Hochschule für Technik und Wirtschaft Dresden (FH)

Fachbereich Informatik/Mathematik

Diplomarbeit

im Studiengang Allgemeine Informatik

Thema: Entwurf und prototypische Implementierung einer Test- und Visualisierungsplattform für dynamische IEEE-802.15.4- und Zigbee-Netzwerke

eingereicht von: Daniel Müller
Studiengruppe 05/041/01
Matrikelnummer 21180

eingereicht am: 15. Dezember 2009

Betreuer: Herr Prof. Dr.-Ing. A. Beck (HTW)
Herr Dipl.-Ing. M.Ludwig (dresden elektronik)

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben.

Zunächst möchte ich mich bei der Firma dresden elektronik bedanken, die es mir ermöglichte, an diesem spannenden Thema zu arbeiten. Ich danke all ihren Mitarbeitern für die Tips und Anregungen während der Bearbeitung, insbesondere aber meinem Betreuer, Mike Ludwig, der mir aufgrund seiner langjährigen Erfahrung eine große Hilfe war und stets für eine Diskussion bereit stand.

Ebenfalls bedanken möchte ich mich bei meinem Hochschulbetreuer, Herrn Prof. Beck, der stets für mich ansprechbar war und mir die Freiheit gelassen hat, die Arbeit nach eigenen Vorstellungen zu entwickeln.

Meinen Eltern danke ich dafür, dass sie mir durch ihre finanzielle Unterstützung das Studium überhaupt erst ermöglicht haben.

Großer Dank gebührt schließlich auch meiner Freundin Manja Schumann für ihre moralische Unterstützung und die vielen Stunden des Korrekturlesens.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis verwendeter Akronyme

Bezeichnung	Beschreibung	Seiten
6LoWPAN	IPv6 over Low power WPAN	26
ACL	Access Control List	24
AES	Advanced Encryption Standard	16, 24
APL	Application Layer	21
APS	Application Support Layer	23
BLOB	Binary Large Object	93
CAP	Contention Access Period	15
CBC-MAC	Cipher Block Chaining Message Authentication Code	24
CCA	Clear Channel Assessment	13
CCM*	Counter Mode with CBC-MAC	24
CFP	Contention Free Period	15
CSMA-CA	Carrier Sense Multiple Access with Collision Avoidance	15
DDoS	Distributed Denial Of Service	16
DOM	Document Objekt Model	50
FCF	Frame Control Field	90
FFD	Full Function Device	9, 14, 21, 32
HVAC	Heating, Ventilating and Air Conditioning	23
IDE	Integrated Development Environment	31
IEEE	Institute of Electrical and Electronics Engineers	8
IETF	Internet Engineering Task Force	26
ISM	Industrial, Scientific and Medical	17
ISO	International Organization for Standardization	11
ISP	In-System Programming	33
JAR	Java Archive	86
LLC	Logical Link Control	11
LQI	Link Quality Indicator	13
LR-WPAN	Low Data Rate Wireless Personal Area Network	8, 9
MAC	Medium Access Control Layer	11–14, 21, 25, 26, 55, 58, 61

Bezeichnung	Beschreibung	Seiten
MCPS	MAC Common Part Sublayer	12
MIC	Message Integrity Check	16
MLME	MAC Layer Management Entity	12
MO	Managed Object	22
MPDU	MAC Protocol Data Unit	69
MTU	Maximum Transfer Unit	26
NIB	Network Layer Information Base	22
NWK	Network Layer	11, 12, 21, 61
OSI	Open Systems Interconnection	11
PAL	Platform Abstraction Layer	32, 33, 55
PAN	Personal Area Network	6, 14, 15
PD	Physical Data	12
PDU	Protocol Data Unit	13, 21, 69
PHY	Physical Layer	11, 12, 21, 25, 55, 61
PIB	PAN Information Base	12, 67
PLME	Physical Layer Management Entity	12
PPDU	PHY Protocol Data Unit	13
RCB	Radio Controller Board	31, 72
RF4CE	Radio Frequency for Consumer Electronics	27
RFC	Request For Comments	26
RFD	Reduced Function Device	9, 14, 21
RUM	Route Under MAC	34
SAP	Service Access Point	12
SAX	Simple API for XML	50
SKKE	Symmetric-Key Key Establishment Protocol	25
SSCS	Service Specific Convergence Sublayer	11, 26
STB	Sensor Terminal Board	54, 72
TCL	Testbed Control Layer	55, 67
USART	Universal Asynchronous Receiver/Transmitter	72
WPAN	Wireless PAN	6, 8, 26, 27, 29
WSN	Wireless Sensor Network	6

Bezeichnung	Beschreibung	Seiten
XML	Extensible Markup Language	50
ZCL	ZigBee Cluster Library	23
ZDO	ZigBee Device Object	23

Glossar

ACK	Acknowledgement (Bestätigung). 14, 25, 69
Application Layer	Anwendungsschicht. vi, 21
Beacon	Sinngemäß: Funkfeuer, Signal, welches das Vorhandensein eines Knotens signalisiert und dessen Parameter propagiert. 14, 15, 19
Beacon Order	Zeitintervall zwischen zwei Beaconaussendungen. 15
Clear Channel Assessment	Zusicherung eines unbelegten Übertragungskanals vor der Datenübertragung. vi, 13
End-Device	Regulärer Netzknoten ohne dedizierte Funktion. 9, 15
Energy Detection	Signaldetektion, sinngemäß: Abhören des Kanals, ob dieser bereits belegt ist. 13
Firefly	Name einer Arbeitsgruppe um Philips Electronics, Vorgänger von ZigBee. 8
Frequency Hopping	Frequenzsprungverfahren. 13
High Band	Von IEEE 802.15.4 genutztes Frequenzband im Bereich 2400-2483,5MHz . 13, 31
IEEE	Der weltweit grösste Berufsverband von Ingenieuren aus den Bereichen Elektrotechnik und Informatik. Er ist hierarchisch gegliedert in Societies, Working- und Taskgroups. vi, 8
In-System Programming	Programmierung der Zielhardware in eingebautem Zustand. vi, 33
Link Quality Indicator	Kanalqualitätsanzeige. vi, 13
Low Band	Von IEEE 802.15.4 genutztes Frequenzband, welches regionalspezifisch entweder im Bereich 868-868,8 MHz (Europa) bzw. 902-928 MHz (Nordamerika, Australien) liegt. 13
Man-in-the-Middle-Attacke	Angriffsart, bei der ein im Kommunikationsweg befindlicher Teilnehmer die über ihn geleiteten Nachrichten manipuliert. 16
Medium Access Control Layer	Medienzugriffsschicht. vi, 11
Mote	Eine alternative Bezeichnung für Sensorknoten. 6
Network Layer	Netzwerkschicht. vii, 11

PAN-Coordinator	Netzknoten der das PAN verwaltet. 9, 10
Payload	Nutzlast eines Datenpakets. 14
Physical Layer	Bitübertragungsschicht. vii, 11
Replay-Angriff	Angriffsart, bei der eine Nachricht erneut versendet wird. 16
SAP	Dienstzugriffspunkt. vii, 12
Slotted Mode	Betriebsart eines PAN, bei welcher der PAN-Coordinator regelmäßig Beacons versendet. 15
Spoofing	Angriffsart bei der die Adressierung verfälscht wird. 16
Superframe Order	Länge der aktiven Phase eines Superframes. 15
Transceiver	Sende-Empfangseinheit. 13
Ubiquituous Computing	Allgegenwart von Rechnernetzen in Form intelligenter Gegenstände, in der kommerziellen Verwendung auch als „pervasive computing“ bezeichnet. 6
Unmarshalling	Erzeugung eines Objektbaumes durch Deserialisierung aus einem XML-Dokument. 75
Unslotted Mode	Betriebsart eines PAN, bei welcher der PAN-Coordinator Beacons nur auf Anfrage versendet. 15, 19

Teil I

Einleitung

Einleitung

Die vorliegende Arbeit entstand im Auftrag von und in Kooperation mit dresden elektronik und behandelt ein bislang ungelöstes Problem aus dem Tätigkeitsfeld der Firma. In diesem Kapitel wird zunächst die Aufgabenstellung benannt und die Motivation ihrer Bearbeitung erläutert. Es folgt ein kurzes Firmenportrait und abschließend wird die Gliederung der Arbeit erläutert.

0.1 Aufgabenstellung

Der Test und die Visualisierung der dynamischen Vorgänge in drahtlosen Sensor-Aktor-Netzwerken ist die Vorraussetzung, um diese Netzwerke beurteil- und beherrschbar zu gestalten. Der IEEE Standard 802.15.4 definiert die unteren Schichten für drahtlose Nahbereichsnetzwerke (LR-WPANs), worauf verschiedene andere Standards – wie z.B. Zigbee – aufbauen, um spezielle Aufgaben zu lösen.

Ziel der Diplomarbeit ist der Entwurf und die Umsetzung eines Testsystems, in dem mehrere Knoten gleichzeitig betrieben, beobachtet und beeinflusst werden können. Dabei muss besonders auf Skalierbarkeit und offene Schnittstellen geachtet werden, um sowohl Netzwerke nach IEEE 802.15.4 als auch darauf aufbauende Standards untersuchen zu können. Als Knoten werden Funkmodule der Firma dresden elektronik ingenieurtechnik verwendet.

Damit die Auswirkungen unterschiedlicher Knotenfunktionalitäten zuverlässig getestet werden können, muss das entstehende System in der Lage sein, bestehende Testfälle automatisch durchzuführen und die dabei gewonnenen Daten aufzuzeichnen sowie zu visualisieren. Mit Hilfe geeigneter Such- und Filteralgorithmen sollen dabei relevante Daten, z.B. Fehlübertragungen oder Störungen, innerhalb der Gesamtdatenmenge gesucht und extrahiert werden können.

0.2 Motivation

Die zunehmends fortschreitende Miniaturisierung elektronischer Bauteile sowie wachsende Integrationsdichten ermöglichen die Herstellung leistungsfähiger Module bei beständig sinkendem Raumbedarf. Bereits seit den 1980er Jahren werden komplette Computer als Einchiplösungen angeboten (sog. Mikrocontroller) und seitdem stetig um Speicher und Schnittstellen erweitert. Fallende Marktpreise – bedingt durch Optimierungen der Fertigungsabläufe – steigern die Attraktivität angebotetener Produkte und ermöglichen so ein immer weiteres Vordringen ihres Einsatzes in den Alltag.

Dieser technologische Wandel betrifft auch die Hochfrequenztechnik, wo inzwischen komplette Funktransceiver auf einem Chip verfügbar sind. Durch Koppelung eines Transceivers an einen Mikrocontroller entstehen so Funkmodule, die neben der klassischen Datenübertragung für Steuerungs- und Regelungsaufgaben eingesetzt werden können. Durch den offensichtlichen Entfall einer Verkabelung ist es möglich, Funkmodule beliebig zu platzieren, leicht nachträglich umzusetzen oder in ad-hoc-Szenarien zu verwenden. Bezüglich ihres Energiebedarfs erlaubt es der mittlerweile erreichte technologische Stand, Funkmodule batteriegestützt zu betreiben, wobei sich Batterie-Wechselintervalle im Bereich mehrerer Jahre erzielen lassen.

Aus logischer Sicht übernimmt ein Funkmodul stets nur einfachste Aufgaben: es dient entweder als Sensor, als Aktor oder zur Datenweiterleitung zwischen anderen Modulen. Erst durch die Vernetzung von Modulen entsteht ein intelligentes System. Zunächst nur Gegenstand der Forschung, hat ob der preislichen Attraktivität inzwischen die Industrie ein verstärktes Interesse an derartigen Sensornetzen entwickelt. Einsatzszenarien, beispielsweise in der Gebäudeautomatisierung, im Gesundheitswesen oder der Logistik, die vor Jahren noch als Utopie galten, können heute realisiert werden.

Diese neue Technologie kann mit herkömmlichen, kabelgebundenen jedoch nur dann konkurrieren, wenn sie deren Zuverlässigkeit und Sicherheit erreicht oder übertrifft. Widrige Einflüsse, wie durch die Gebäudekonstruktion bedingt vorhandene Abschirmungen, konkurrierende Nutzer des Frequenzbands oder die temporäre Abschattung eines Funkmodules dürfen dessen Funktion nicht beeinträchtigen. Aktoren sollen nur aus dem eigenen Netz ansprechbar sein, Sensoren sich nicht unberechtigt darin integrieren lassen.

Bisherige Experimente beschränken sich auf real nachgebildete Punkt-zu-Punkt-Verbindungen. Mit steigender Anzahl von Netzteilnehmern wächst jedoch die Komplexität dermaßen drastisch, dass dazu bislang nur Simulationen durchgeführt wurden. Die Untersuchung großflächig vernetzter Systeme steht indes noch völlig aus. Hierfür existieren bislang weder Simulationsmodelle noch konkrete Versuchsergebnisse, es wird aber international dazu geforscht.

0.3 Firmenkurzportrait dresden elektronik

Die dresden elektronik ingenieurtechnik gmbh [DE09b] – im Folgenden „dresden elektronik“ genannt – ist ein mittelständisches Unternehmen, dessen Sitz sich in Dresden befindet. Seit der Gründung 1990 als Zweimann-Betrieb hat sich der Mitarbeiterstamm ständig vergrößert, derzeit gibt es 55 Beschäftigte. Das Firmen-

portfolio umfasst die Bereiche Elektronikentwicklung und –fertigung, wobei Produkte oftmals beide Phasen in der Firma durchlaufen und sich Kunden in allen Branchen wiederfinden lassen.

Einer der Entwicklungsschwerpunkte ist die Verkehrstechnik. So wurde ein System zur Steuerung von Anzeigetafeln entwickelt, welches von den Dresdner Verkehrsbetrieben (DVB) an vielen Straßenbahnhaltestellen im Stadtgebiet eingesetzt wird. Im Jahr 2006 wurde dieser Bereich in ein Tochterunternehmen ausgegründet.

Der zukunftsträchtigste Tätigkeitsschwerpunkt jedoch ist der Datenfunk. In Kooperation mit Atmel entwickelt, fertigt und vertreibt dresden elektronik ZigBee-konforme Funkmodule und bietet passend dazu Development- und Starter-Kits, Evaluation Boards sowie eine breite Zubehörpalette an. Die Prüfung gefertigter Funkmodule beschränkt sich derzeit auf die Hardwarefunktionen. Tests, bei denen Knoten als Teil eines Netzes agieren, finden bislang nicht statt. Zur Dimensionierung, Konfiguration und Evaluierung kundenspezifischer Funknetze ist jedoch ein skalierbares Emulationssystem auf Panelbasis in Planung, zu dessen Betrieb, Steuerung und Überwachung individuelle Software benötigt wird. Die folgende Arbeit stellt den Entwurf einer solchen Lösung und dessen prototypische Umsetzung vor.

0.4 Gliederung der Arbeit

Mit drahtlosen Nahbereichsnetzen und den sich dahinter verbergenden Technologien beginnend, werden im zweiten Teil der Arbeit die Grundlagen vermittelt, die für ihr Verständnis erforderlich sind. Der praktische Einsatz von Funkmodulen mit Fokus auf Produkten der Firmen Atmel und dresden elektronik ingenieurtechnik wird ebenso umrissen, wie bereits existierende Testplattformen.

Ausgehend von einer Aufnahme der Anforderungen an die Testplattform sowie denkbaren Testszenarien wird im Hauptteil der Arbeit eine mögliche Systemarchitektur dafür vorgestellt. Als Beweis ihrer Umsetzbarkeit werden anschließend die Prototypen einzelner Komponenten sowie ein Beispieldurchlauf erläutert. Nach einer Zusammenfassung des aktuellen Projektstandes schließt diese Arbeit mit einem Ausblick auf zukünftige Entwicklungen.

Teil II

Grundlagen

1 Sensornetze und private Nahbereichsnetze

1.1 Begriffsbestimmung und Motivation

Als *Sensornetz* wird ein Rechnernetz bezeichnet, welches aus autonom arbeitenden *(Sensor-)Knoten*, sogenannten Motes[1], besteht, die ihre Umgebung beobachten [Vgl. Hae08]. Abhängig von den zu erfassenden Messgrößen variieren Aufbau und Größe eines Knotens dabei erheblich[2]. Zur Kommunikation werden entweder bestehende Infrastrukturnetze genutzt oder ad hoc ein dediziertes Netz aufgespannt. Als Übertragungsmedien dienen wahlweise Schall, elektrische Impulse oder elektromagnetische Wellen. Erfolgt die Kommunikation drahtlos, wird ein solches Netz auch als Wireless Sensor Network (WSN) bezeichnet.

Mittlerweile wird implizit davon ausgegangen, dass es sich bei einem Sensornetz um ein „großes, ad hoc, multihop und unpartitioniertes Netzwerk von kleinen, homogenen, unbeweglichen und ressourcenbegrenzten Sensorknoten“ handelt [Vgl. Ten07, S. 1]. Ihren Ursprung haben Sensornetze in militärischen Frühwarnsystemen, nach dem heutigen Forschungsstand können sie überall dort eingesetzt werden, wo großflächig Daten zu erfassen sind[3].

Personal Area Networks (PANs) entsteht durch Zusammenschluss von Kleingeräten, der entweder drahtgebunden[4] oder drahtlos[5] (WPAN[6]) erfolgt. So entstandene Netze bilden die kleinste Stufe in der Netzwerkhierarchie, da sie eine Ausdehnung von nur wenigen Metern aufweisen. Sie können entweder isoliert betrieben oder per Uplink in ein größeres Netz integriert werden. PAN-Knoten sind dabei nicht an eine Funktion gebunden. PANs erweitern in dieser Hinsicht das Konzept der Sensornetze, da der Erhebung und Übertragung von Daten oftmals ihre Speicherung und Auswertung folgt und ggf. eine Aktorik angegliedert ist.

Den beim Entwurf drahtloser Netze anvisierten Zielen – sicherer Kommunikation, preiswerten Knoten und geringer Leistungsaufnahme – stehen Forderungen wie Selbstorganisation und der Einsatz adaptiver Routingalgorithmen entgegen, die sich gegenseitig widersprechen [Vgl. Sik04]. Nach Mark Weisers Vision [Vgl. Wei91] vom Ubiquituous Computing[7] werden Rechnernetze aber in jedem Fall

[1] Eine alternative Bezeichnung für Sensorknoten.
[2] Das derzeitige Minimum liegt bei einem Millimeter Kantenlänge [Vgl. Pis01].
[3] Das derzeit größte zusammenhängende Sensornetz besteht aus 1000 Knoten und erstreckt sich über $0.4km^2$ [Vgl. Exs].
[4] Z.B. per USB, FireWire.
[5] Z.B. per Lichtwellen (IrDA) oder funkbasiert (Bluetooth, Wireless LAN).
[6] Wireless PAN.
[7] Allgegenwart von Rechnernetzen in Form intelligenter Gegenstände, in der kommerziellen Verwendung auch als "'pervasive computing"' bezeichnet.

schrittweise Einzug in den menschlichen Alltag halten und früher oder später gar nicht mehr als solche wahrgenommen werden [Vgl. Mat]. Bereits mögliche und in Zukunft denkbare Einsatzgebiete intelligenter drahtloser Sensor-Aktor-Systeme werden im nächsten Kapitel vorgestellt.

1.2 Einsatzbeispiele

Industrie- und Gebäudeautomatisierung: An rotierenden Teilen, die sich prinzipbedingt nicht verkabeln lassen, können durch Funksensoren trotzdem Messwerte erfasst werden. Aufwändige Verkabelungen sind durch funkbasierte Netze ersetzbar. Unter widrigen Bedingungen[8], arbeiten speziell angepasste Sensornetze dennoch zuverlässig. Der Einsatz einer zentralen Steuereinheit ermöglicht die Realisierung komplexer Steueraufgaben bei maximaler Flexibilität[9].

Intelligentes Wohnen: In einem Haus der Zukunft ist ein funkbasiertes Infrastrukturnetz bereits enthalten. Darin integrierte Helligkeits- und Temperatursensoren halten in Verbindung mit steuerbaren Klima- und Verdunkelungsanlagen die Raumtemperaturen eigenständig konstant[10]. Beim Verlassen des Hauses genügt ein Tastendruck, um sämtliche Beleuchtung abzuschalten, offene Fenster zu erkennen und die Heizung herabzuregeln [Vgl. Haa08, Heg08].

Bridge Condition Monitoring: Lange Hängebrücken werden heute zur Erkennung von Verschleiß oder Resonanzen in regelmäßigen Abständen mit Sensoren versehen. Da deren Verkabelungsaufwand mit zunehmender Länge immens wächst, werden in Asien mittlerweile zur Echtzeitüberwachung drahtlose Netze eingesetzt [Vgl. Cha06].

Erkennung von Waldbränden: Durch Geo-Sensornetze lassen sich Waldbrände frühzeitig erkennen. Das zu überwachende Gebiet wird dazu überflogen und auf Funkbasis arbeitende Temperatursensoren abgeworfen. Diese vernetzen sich spontan und begeben sich danach in einen Ruhezustand. In regelmäßigen Intervallen erwachen sie kurzzeitig, ermitteln die aktuelle Umgebungstemperatur und benachrichtigen einen Leitstand, wenn ein Schwellwert überschritten wurde. Wo sich ein einziger ausgelöster Sensor noch als Fehlalarm interpretieren lässt, deutet eine stetig wachsende Menge eingehender Meldungen unterschiedlicher Sensoren eindeutig auf einen Brandherd hin[11].

---

[8] Bspw. Arbeit mit Giften, Radioaktivität, unter hohen Temperaturen oder Drücken.

[9] Bspw. temporäre Deaktivierung von Sensoren, Gruppierung von Aktoren, Änderung der Bindung zwischen Sensoren und Aktoren.

[10] Sogenanntes Ambient Assisted Living.

[11] Die Lokalisierung des Senders erfolgt entweder durch mitgelieferte Geoinformationen, anhand Laufzeitmessungen oder Triangulation basierend auf der Empfangsfeldstärke.

2 Der IEEE-Standard 802.15.4

2.1 Ursprung und Entstehung

Die Ursprünge privater Nahbereichsfunknetze liegen bei Philips, welche im Jahr 2000 das RF-Lite-Programm vorstellten [Vgl. Hom03]. Durch Zusammenschluss mit weiteren Unternehmen wurde später daraus die Arbeitsgruppe Firefly[12] gegründet, deren Ziel die Entwicklung einer kostengünstigen Sensor-Aktor-Lösung für den Einsatz in Produkten der Konsumelektronik war [Vgl. Ead07, S. 1]. Ebenfalls 2000 gründete die IEEE[13] unter der Society 802[14] die Arbeitsgruppe 15 [Vgl. Gif], um nach Alternativen zum Wireless Standard nach IEEE 802.11 zum Einsatz in Sensornetzen zu suchen. IEEE 802.15.4 benennt dabei einen Teil der Ergebnisse dieser Aufgabengruppe, mit deren Veröffentlichung das Firefly-Projekt eingestellt wurde.

Obwohl gelegentlich – fälschlicherweise – synonym verwendet, sind ZigBee und IEEE 802.15.4 zwei voneinander getrennt zu betrachtende Standards. In diesem Zusammenhang werden oftmals auch Begrifflichkeiten vermischt. ZigBee und weitere auf IEEE 802.15.4 aufsetzende Standards werden im nächsten Kapitel behandelt.

2.2 Einordnung in die Familie der IEEE 802.15-Standards

Die Arbeitsgruppe 802.15 gliedert sich in die sieben Teilbereiche 802.15.1-802.15.7. IEEE 802.15.1 ist auch unter dem Namen Bluetooth bekannt, weitere Arbeitsgruppen befassen sich mit der Koexistenz konkurrierender Technologien, WPANs mit hohen bzw. niedrigen Datenraten, vermaschter Kommunikation, köpernahen Netzen[15] bzw. der Kommunikation mittels Licht[16].

Der IEEE 802.15.4-Standard behandelt in dieser Familie sogenannte LR-WPANs[17]. Die Entwurfsschwerpunkte waren Datenübertragung mit geringen Datenraten bei maximaler Energieeffizienz und mininaler Gerätekomplexität. Die Implementierung eines standardkonformen Gerätes sollte kostengünstig mit einem 8-Bit-Mikrocontroller realisiert werden können, der – batteriebetrieben – über mehrere Jahre lauffähig ist. Bereits die geplante, maximal erreichbare Datenrate von 250 kb/s verdeutlicht, dass der Standard nicht zur Verdrängung bestehender

[12] Name einer Arbeitsgruppe um Philips Electronics, Vorgänger von ZigBee.
[13] Institute of Electrical and Electronics Engineers.
[14] Beschäftigt sich mit der Standardisierung von lokalen Netzen auf den OSI-Schichten 1 und 2.
[15] Body Area Networks (BAN).
[16] Visible Light Communication (VLC).
[17] Low Data Rate Wireless Personal Area Networks.

Technologien, sondern vielmehr ihrer Ergänzung konzipiert wurde. Die geplante Reichweite einer Direktverbindung zwischen zwei Knoten war mit 75 m angegeben[18], größere Distanzen lassen sich durch Multi-Hop-Betrieb überbrücken.

Die Erstveröffentlichung des Standards erfolgte im Jahr 2003 [Vgl. IEE03], 2006 erschien eine überarbeitet Version [Vgl. IEE06], wodurch die Version 2003 praktisch nicht mehr relevant ist. Ergänzend dazu wurden in 2007 ein [Vgl. IEE07] und in 2009 zwei [Vgl. IEE09a], [Vgl. IEE09b] Zusätze veröffentlicht, welche durch Nutzung neuer Frequenzbänder höhere Übertragungsraten erlauben und eine Frequenzagilität vorstellen. Eine Abwärtskompatibilität zu Version 2006 ist hierbei gegeben. Die folgenden Betrachtungen beziehen sich stets auf die Basisversion des Standards von 2006, da die Ergänzungen noch nicht in der Literatur Erwähnung finden und viele Hersteller die Implementierung der neuen Merkmale noch nicht abgeschlossen haben.

2.3 Komponenten eines IEEE-802.15.4 LR-WPAN – Gerätearten und Rollen

Ein LR-WPAN kann aus bis zu 2^{16} Teilnehmern bestehen, die sich einerseits nach ihrem physischen Funktionsumfang, andererseits nach ihrer logischen Rolle differenzieren lassen.

Bezüglich Funktionsumfang unterscheidet der Standard in FFDs[19] und RFDs[20], wobei ein FFD den kompletten Funktionsumfang unterstützt, den der Standard vorsieht, ein RFD hingegen nur eine Minimalimplementierung darstellt, welche mit geringeren Hardwareanforderungen als ein FFD auskommt. Der Standard empfiehlt für FFDs den Netzbetrieb, wohingegen RFDs batteriebetrieben werden und zur Energieeinsparung bei Inaktivität in einen temporären Schlafzustand fallen können. FFDs sind in der Lage sowohl mit anderen FFDs als auch anderen RFDs zu kommunizieren, RFDs können nur mit FFDs kommunizieren.

Als logische Rollen werden PAN-Coordinator[21] und End-Device[22] sowie die damit verbundenen Aufgaben benannt, nicht jedoch *wie* deren konkrete Umsetzung erfolgt. Eine Bindung einer Geräterolle an eine konkrete Einsatzart, bspw. als Sensor oder Aktor wird ebenfalls nicht festgelegt. Ein Coordinator kann dabei nur auf einem FFD implementiert werden, seine Aufgaben umfassen sowohl Kon-

[18] Praktisch lassen sich je nach Ausführung und Richtwirkung der eingesetzten Antennen bei ungestörter Sichtverbindung jedoch mehrere Kilometer erreichen.
[19] Full Function Devices.
[20] Reduced Function Devices.
[21] Netzknoten der das PAN verwaltet.
[22] Regulärer Netzknoten ohne dedizierte Funktion.

trolle als auch Koordination des Netzzugriffs: er erlaubt oder verweigert Endgeräten, dem Netzwerk beizutreten, kann diese gegebenenfalls wieder aus dem Netz entfernen und verwaltet deren logische Adressierung. Optional synchronisiert er die Kommunikation und übernimmt je nach verwendeter Netztopologie zusätzliche Funktionen, wie beispielsweise Nachrichtenweiterleitung (Routing). End-Devices haben keine vorgegebene Funktionalität.

2.4 Netztopologien und Netzaufbau

Grundsätzlich unterscheidet der Standard zwischen zwei verschiedene Netztopologien (Vgl. Abb. 2.1):

- In der *Stern-Topologie* erfolgt die Kommunikation stets über einen zentralen Knoten, den PAN-Coordinator, welcher die Kommunikation zwischen den anderen Netzteilnehmern initiiert und terminiert. Zwischen zwei Endgeräten zu übertragende Nachrichten leitet er weiter, da eine Direktverbindung unter Umgehung des Coordinators nicht möglich ist.
- Bei einer *Peer-to-Peer-Topologie* können Knoten direkt miteinander kommunizieren, sofern sie sich in gegenseitiger Reichweite befinden. Die Kommunikation kann hier über mehrere Zwischenknoten laufen (multi-hop), so dass sich damit große vermaschte Netze aufbauen lassen. Die Aufgaben des auch hier vorhandenen PAN-Coordinators, beschränken sich dann auf die Verwaltung des Netzes.

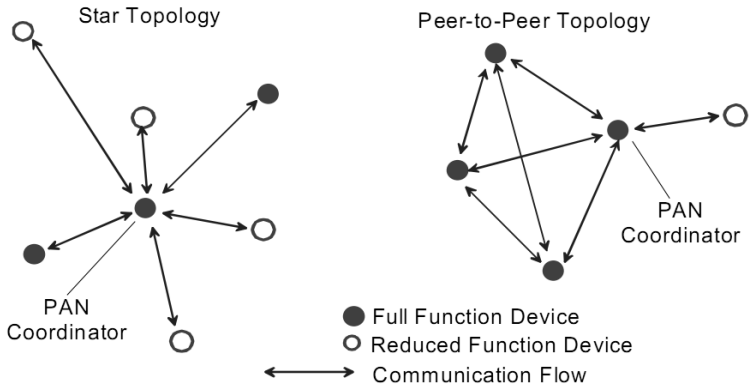


Abbildung 2.1: Nach IEEE 802.15.4 mögliche Netztopologien [Vgl. IEE06, S. 14].

Aufbau und Zusammenschaltung von Netzen werden im Standard nur theoretisch behandelt. Ihre konkrete Umsetzung, wie beispielsweise das Routing von Nachrichten zwischen Teilnetzen, wird in höherliegende Schichten delegiert.

Bevor ein Netz gebildet werden kann, muss ein PAN-Coordinator existieren, wobei jedes Netz genau einen besitzt. Knoten melden sich beim ihm an und gehören

damit zu einem Zeitpunkt immer höchstens zu einem Netz. Der Coordinator gibt eine eindeutige Kennung, die sogenannte *PAN-ID*, vor, wodurch sich Netze gegenseitig physisch überlappen können, logisch jedoch autark arbeiten. Nach Version 2006 darf die selbe PAN-ID auf verschiedenen Kanälen mehrfach verwendet werden, ab 2007 ist dies nicht mehr erlaubt, da ein Netz dort frequenzagil arbeiten kann, d.h. den verwendeten Kanal bedarfsweise wechselt. Kritisch ist in diesem Zusammenhang immer der PAN-Coordinator als Single Point of Failure. Fällt er – aus welchem Grund auch immer – aus, ist das gesamte Netz gestört. Eine Reorganisation des Netzes im Sinne einer Selbstheilung wurde vom Standard in höhere Schichten verlagert.

Ein Verbindungsaufbau zwischen zwei Knoten ist prinzipiell innerhalb weniger Millisekunden möglich. Sind Knoten ausnahmsweise bereits untereinander bekannt, d.h. können sie den Kommunikationspartner explizit adressieren, so können sie auch ohne an einem Netz teilzunehmen gegenseitig Daten austauschen.

Eigenständige Peer-to-Peer Netze können zu sogenannten *Cluster Tree-Netzen* verkoppelt werden. Ein Cluster besteht dabei jeweils aus einem PAN-Coordinator sowie einigen Endgeräten. Ein FFD innerhalb des Clusters (oft der Coordinator) arbeitet als Gateway zu benachbarten Clustern, wodurch sich komplexe Netze aufbauen und Nachrichten cluster-übergreifend versenden lassen. Auch Knoten, die sich nicht in direkter Reichweite befinden, wird es so ermöglicht, miteinander zu kommunizieren.

2.5 Protokollstack

In Anlehnung an das ISO[23]/OSI[24]-Schichten-Referenzmodel definiert der IEEE-Standard 802.15.4 lediglich dessen unterste zwei Schichten: den PHY[25], sowie den darüberliegenden MAC[26], wobei der OSI-Layer 2 damit nicht vollständig abgedeckt ist, da die ebenfalls zur Sicherungsschicht gehörenden beiden Teilschichten LLC[27] und SSCS[28] fehlen. Sämtliche höhere Schichten sind ebenfalls nicht direkter Bestandteil des Standards, jedoch insofern relevant, dass die Schnittstelle zum unmittelbar darüberliegenden NWK[29] beschrieben wird und der Standard partiell Vorschläge für dessen Verhalten trifft.

[23] International Organization for Standardization.
[24] Open Systems Interconnection.
[25] Physical Layer.
[26] Medium Access Control Layer.
[27] Logical Link Control.
[28] Service Specific Convergence Sublayer.
[29] Network Layer.

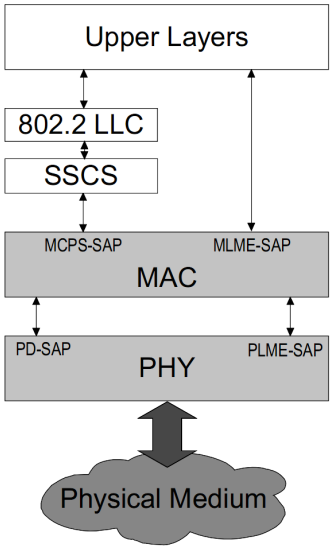


Abbildung 2.2: Der IEEE 802.15.4-Protokollstack [Vgl. IEE06, S. 16].

Für beide direkt definierten Schichten ist jeweils eine sogenannte PIB[30] vorgesehen, über die wesentliche Parameter der jeweiligen Schicht abfragbar und partiell auch änderbar sind. Darüber hinaus werden die Schnittstellen zwischen den Schichten über sogenannte SAPs[31] definiert, welche in Steuerung und Datenübertragung unterteilt werden. Zwischen NWK und MAC gibt es zur Datenübertragung den MCPS[32]-SAP, für Steuerungzwecke wird der MLME[33]-SAP verwendet. Analog existieren zwischen PHY und MAC ebenfalls zwei Zugriffspunkte: zur Datenübertragung der PD[34]-SAP, zur Steuerung der PLME[35]-SAP.

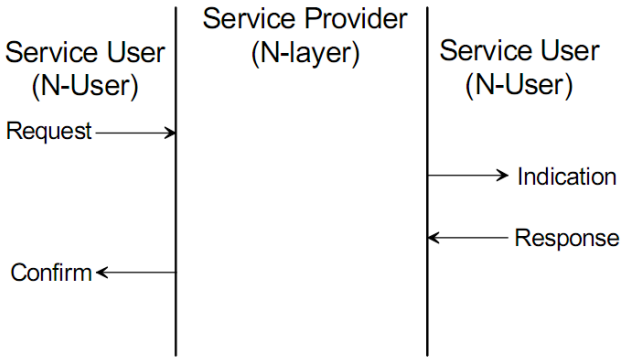


Abbildung 2.3: Von der MAC-Schicht angebotene Dienstprimitive [Vgl. IEE06, S. 25].

Obwohl der MAC-Layer bereits dedizierte Kommunikationskanäle bereitstellt, wird eine (applikationsabhängige) Kommmunikation zwischen Knoten nach OSI-Schichtenmodell erst durch die darüberliegende Netzwerkschicht realisiert. Deren Zugriff auf die MAC-Schicht über die beiden bereitgestellten SAPs wird im

[30] PAN Information Base.
[31] Service Access Points.
[32] MAC Common Part Sublayer.
[33] MAC Layer Management Entity.
[34] Physical Data.
[35] Physical Layer Management Entity.

Standard durch vier Kommunikationsprimitive modelliert (Vgl. Abb. 2.3). Die auf Ebene der MAC Schicht bereitgestellten Kommandos sind orthogonal ausgerichtet, so dass sich deren Anzahl mit 35 in einem überschaubaren Rahmen bewegt. Zum Datenübertragung wurde ein Mechanismus aus dem ISO/OSI-Schichtenmodell übernommen, der zu transportierende Daten paketiert und in PDUs[36] kapselt. Zum Versand werden diese PDUs ebenenweise jeweils mit einem Header versehen und bis zur niedrigsten Schicht an die jeweils darunterliegende weitergereicht. Auf der Empfängerseite läuft dieser Vorgang in umgekehrter Reihenfolge ab.

2.5.1 Physical Layer

Die beiden Hauptaufgaben des Physical Layers sind Versand und Empfang sogenannter PPDUs[37] über die Luftschnittstelle. Unmittelbar damit verbunden sind die Aktivierung und Deaktivierung des Transceivers[38], Energy Detection[39], LQI[40], Kanalselektion und CCA[41]. An dieser Stelle sei besonders darauf hingewiesen, dass es sich hier nur um physische Übertragungskanäle handelt; logische Übertragungskanäle werden erst auf MAC-Schicht realisiert.

Im Frequenzspektrum werden ausschließlich lizenzfreie Frequenzen verwendet, wobei zwei verschiedene Bänder vorgesehen sind: das Low Band[42] und das High Band[43]. Die zur Übertragung eingesetzten Frequenzspreiztechniken und Modulationsarten gelten als äußert effektiv, da sie auch bei geringem Signal-Rausch-Abstand nur geringe Bitfehlerraten zur Folge haben [Vgl. Pet09]. Nach Standardversion 2006 werden insgesamt 27 Kanäle zur Verfügung gestellt (Vgl. Tabelle 2.1). Ein von anderen Funktechnologien bekanntes Frequency Hopping[44] wird nicht unterstützt.

Frequenzbereich	*Anzahl der Kanäle*	*maximale Datenrate*
868MHz	1	20kb/s
915MHz	10	40kb/s
2.4GHz	16	250kb/s

Tabelle 2.1: Durch IEEE 802.15.4 definierte Frequenzen, zugehörige Kanäle und Datenraten [Vgl. IEE06, S. 28].

[36] Protocol Data Units.
[37] PHY Protocol Data Units.
[38] Sende-Empfangseinheit.
[39] Signaldetektion, sinngemäß: Abhören des Kanals, ob dieser bereits belegt ist.
[40] Link Quality Indicator.
[41] Clear Channel Assessment.
[42] Von IEEE 802.15.4 genutztes Frequenzband, welches regionalspezifisch entweder im Bereich 868-868,8 MHz (Europa) bzw. 902-928 MHz (Nordamerika, Australien) liegt.
[43] Von IEEE 802.15.4 genutztes Frequenzband im Bereich 2400-2483,5MHz .
[44] Frequenzsprungverfahren.

2.5.2 Medium Access Control Layer

Der MAC-Layer setzt auf dem Physical Layer auf. Er ist für die Bereitstellung logischer Kanäle verantwortlich und bietet hierfür folgende Funktionen: Beacon[45]-Management, An- und Abmeldung gegenüber einem PAN, Kanalzugriff mit garantierten Zeitschlitzen, Fehlerbehandlung durch Bestätigungen sowie Sicherheitsfunktionen. Für parallel arbeitende Netze stellt er einen Mechanismus zur Behandlung von PAN-ID-Konflikten bereit. Die eingangs erwähnten Geräteart implementieren dabei die hier aufgelisteten Funktionen entweder vollständig (FFD) oder nur partiell (RFD).

Die Adressierung von Netzknoten erfolgt wahlweise über ihre eindeutige 64-Bit-Hardware-Adresse[46] oder eine logische 16-Bit-Short-Adresse[47]. Coordinatoren vergeben sich selbst eine logische Adresse, anderen Knoten wird sie während der Anmeldung am Netz zugewiesen. Knoten können entweder direkt oder indirekt miteinander kommunizieren. Im ersten Fall findet die Zustellung von Nachrichten sofort statt, im letzteren werden sie beim Coordinator hinterlegt und vom Empfänger dort abgeholt. Dies ist besonders dann von Bedeutung, wenn der adressierte Knoten mit Schlafphasen arbeitet.

Zur Kommunikation zwischen Knoten werden sogenannte Frames übertragen, die in vier Typen auftreten: ACK[48], Beacon, Command und Data. Ein Frame besteht generell aus einer Framekennung, welche Frametyp, verschiedene Flags, eine Sequenznummer sowie Quell- und Ziel-Adressierung beinhaltet[49], einer optionalen Payload[50] und einer abschließenden Prüfsumme. Die maximale Länge eines Frames beträgt dabei 127 Byte, wodurch darin bei voller Adressierung höchstens 102 Byte Nutzlast untergebracht werden können.

Datenübertragungen werden als Data-Frames realisiert und können durch Übertragungsbestätigungen abgesichert werden. Ist im Header eines Data-Frames vermerkt, dass eine Bestätigung zu senden ist, muss unmittelbar nach dessen Eingang beim Adressaten ein ACK-Frame zurückgesendet werden. Bleibt dies aus, erfolgt solange eine Wiederholung der Übertragung, bis ein Wiederholungszähler abgelaufen ist.

[45] Sinngemäß: Funkfeuer, Signal, welches das Vorhandensein eines Knotens signalisiert und dessen Parameter propagiert.
[46] I.d.R. abgelegt im EEPROM des Moduls.
[47] veränderlich zur Laufzeit, Zweck: Reduzierung des Adressierung um Platz für Nutzdaten zu gewinnen
[48] Acknowledgement (Bestätigung).
[49] Nicht bei ACK-Frames.
[50] Nutzlast eines Datenpakets.

Command-Frames übertragen Steuerinformationen, Beacon Frames werden nur vom Coordinator versendet und enthalten dessen PAN-ID, Kanalnummer und Short-Adresse, eine sogenannte *Superframe-Struktur* sowie optional eine Liste von Knoten, für die der Koordinator Nachrichten gepuffert bereithält. Durch die Superframe-Struktur wird der Zugriff auf das Netz in der Zeitebene in Zeitschlitze partitioniert, die entweder individuell an Geräte vergeben werden oder für alle Netzteilnehmer zur Verfügung stehen (Vgl. Abb. 2.4). Die Aussendung von Beacon-Frames und das Format des darin enthaltenen Superframes stehen in direktem Zusammenhang mit den beiden Modi, in denen ein PAN betrieben werden kann:

- Im Unslotted Mode[51] werden Beacons nur auf Anfrage durch den Coordinator versandt. Durch den Superframe werden generell keine Zeitschlitze verteilt. Der Netzzugriff ist jederzeit möglich und erfolgt per unslotted CSMA-CA[52]-Verfahren. Sendewillige Netzteilnehmer versuchen dabei Kanalübersprechen zu verhindern, indem sie den Kanal vor Beginn der Übertragung abhören und erst dann zu senden beginnen, wenn der Kanal unbelegt erscheint. Vor Prüfung auf einen freien Kanal sowie im Falle seiner Belegung werden zufällige Wartezeiten (Backoff-Periods) eingefügt.
- Beim Slotted Mode[53] strahlt der Coordinator Beacons in regelmäßigen Zeitintervallen gemäß Beacon Order[54] aus, auf die sich alle anderen Netzknoten synchronisieren. Der Superframe beinhaltet hier zwei Aktivitäts-Intervalle, die zur *active period* zusammengefasst werden. In der CAP[55] werden Knoten explizit Zeitschlitze zugewiesen, in der CFP[56] ist der Kanal ist durch alle Geräte nutzbar. Optional kann sich daran eine *inactive period* anschließen, in der der Coordinator in einen Schlafmodus zurückfällt. Die Länge der active period wird durch die sogenannte Superframe Order[57] definiert, ihr Verhältnis zur Beacon Order entscheidet darüber, ob es Inaktivitätsphasen gibt oder nicht. Hat ein End-Device keinen Kommunikationsbedarf, kann es ebenfalls Schlafperioden einlegen, um Energie zu sparen. Kurz vor der erwarteten Beaconausstrahlung wacht es wieder auf, um den Beacon zu empfangen und dessen Inhalt zu parsen. Enthält der Beacon eine Ankündigung, dass für den Knoten Nachrichten beim Coordinator hinterlegt wurden, versucht der Knoten diese zunächst abzuholen, andernfalls verfällt er unmittelbar wieder in den Ruhezustand.

[51] Betriebsart eines PAN, bei welcher der PAN-Coordinator Beacons nur auf Anfrage versendet.
[52] Carrier Sense Multiple Access with Collision Avoidance.
[53] Betriebsart eines PAN, bei welcher der PAN-Coordinator regelmäßig Beacons versendet.
[54] Zeitintervall zwischen zwei Beaconaussendungen.
[55] Contention Access Period.
[56] Contention Free Period.
[57] Länge der aktiven Phase eines Superframes.

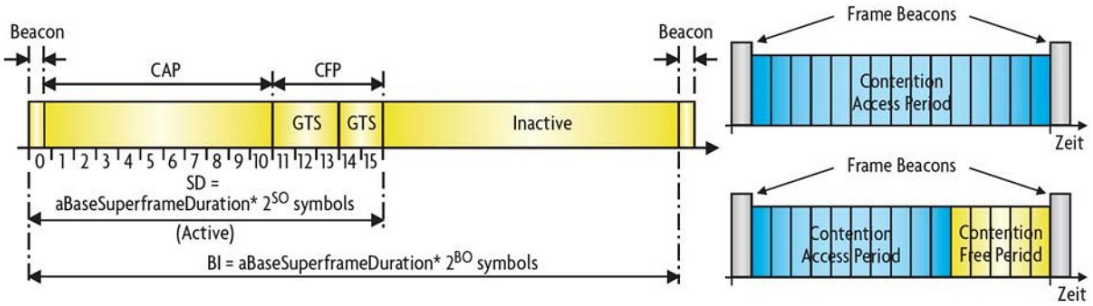


Abbildung 2.4: Superframe Struktur mit und ohne zugesicherte Zeitschlitze [Vgl. Sik04, S. 7].

Abschließend sei erwähnt, dass ACK-Frames stets ohne Verwendung des CSMA/-CA-Mechanismus' gesendet werden.

2.6 Sicherheitsbetrachtungen

Der Sicherheit in Funknetzen muss besondere Aufmerksamkeit gewidmet werden. Da sich Funkverbindungen schlecht abschirmen lassen, ist es prinzipiell möglich, dass Angreifer unbemerkt Übertragungen mithören, Pakete verändern oder zusätzlich einspielen können. Mögliche Angriffsarten sind dabei DDoS[58], Man-in-the-Middle-Attacke[59], Replay-Angriff[60] und Spoofing[61]. Mit den folgenden zwei Mechanismen lässt sich dies vermeiden: durch *Verschlüsselung* werden Inhalten von Nachrichten vor unbefugtem Zugriff verborgen, durch *Authentifizierung* wird erreicht, dass nur dazu berechtigte Kommunikationspartner Nachrichten austauschen.

Um den Datenverkehr gegenüber möglichen Angriffen zu schützen, sieht der IEEE 802.15.4-Standard daher Sicherheitsmechanismen, wie beispielsweise eine Integritätssicherung per MIC[62] oder eine symmetrische Verschlüsselung unter Verwendung von AES[63] mit einer Schlüssellänge von 128 Bit vor.

Prinzipiell wäre eine Verschlüsselung sowohl auf PHY- als auch auf MAC- oder Netzwerkschicht vorstellbar, aber nicht überall ist sie sinnvoll. Wird sie in der PHY-Schicht angesiedelt, muss ein Paket auf allen Knoten, die an dessen Weiterleitung beteiligt sind, zunächst entschlüsselt werden, um den Empfänger zu ermitteln. Ist der Empfänger nicht mit dem aktuellen Knoten identisch, muss das Paket vor seiner Weiterleitung erneut verschlüsselt werden und so fort. Hinsichtlich der verwendeten Schlüssel muss differenziert werden: Existiert ein Netzwerk-

[58] Distributed Denial Of Service.
[59] Angriffsart, bei der ein im Kommunikationsweg befindlicher Teilnehmer die über ihn geleiteten Nachrichten manipuliert.
[60] Angriffsart, bei der eine Nachricht erneut versendet wird.
[61] Angriffsart bei der die Adressierung verfälscht wird.
[62] Message Integrity Check.
[63] Advanced Encryption Standard.

globaler Schlüssel? Wird Hop-weise unterschiedlich verschlüsselt? Sind in strukturierten Netzen hierarchisch unterschiedliche Schlüssel möglich? Werden Schlüssel fix im Knoten hinterlegt, oder können sie zur Laufzeit dynamisch verteilt werden. Wie wird im Falle eines Angriffes verfahren - sind Schlüssel zurückziehbar?

Der IEEE 802.15.4-Standard platziert die Verschlüsselung daher auf MAC-Schicht, wobei nur die Payload innerhalb der Frames verschlüsselt wird und der Frame Header zur Umgehung des o.g. Problems der sonst unbekannten Adressierung unverschlüsselt bleibt. Zusätzlich übernimmt der MAC-Layer die Schlüsselverwaltung, Festlegung und Verteilung der Schlüssel wurden jedoch in höhere Schichten verschoben.

2.7 Koexistenz mit anderen Funktechnologien

Das verwendete 2.4 GHz-Band ist Teil des lizenzfreien ISM[64]-Bandes. Es wird konkurrierend mit einer Reihe weiterer Kommunikationsstandards benutzt, wovon die bekanntesten IEEE 802.15.1 (Bluetooth) und IEEE 802.11[65] (Wireless LAN) sind, Aber auch proprietäre Funklösungen oder Mikrowellenherde strahlen in diesem Frequenzband. Es kann daher zu Interferenzen kommen, die bereits Gegenstand zahlreicher Untersuchungen waren. Abhängig von deren Versuchsbedingungen, Interessenlage und Auftraggeber schwanken die Interpretationen der Ergebnisse jedoch stark und sprechen entweder für einen unbedenklichen Einsatz von IEEE 802.15.4 oder raten grundsätzlich davon ab. An dieser Stelle werden daher lediglich unwiderlegbare Fakten genannt und Ergebnisse universitärer Studien zusammengefasst.

Der Einfluss von Bluetooth auf IEEE 802.15.4 spielt dabei nur eine untergeordnete Rolle, da dort ein Frequenzsprungverfahren eingesetzt wird, bei dem sich die Übertragungsfrequenz permanent ändert. Bei Wireless LAN werden fixe Frequenzen benutzt, die Kanäle 1, 6 und 11 überlappen sich direkt mit einer Vielzahl von IEEE 802.15.4-Kanälen, so dass nur noch die IEEE 802.15.4-Kanäle 15,20,25 und 26 davon unbetroffen sind (Vgl. Abb. 2.5). Untersuchungen von IEEE 802.15.4 Übertragungen, bei denen der Übertragungskanal von einem zeitgleich übertragenden Wireless LAN benutzt wurde [Vgl. Pet09], ergaben, dass:

- Bei kleinen Paketen eine geringere Paketfehlerrate erzielt wird als bei großen Paketen. Dies ist durch das Verhältnis zwischen Nutz- und Fehlerkorrektur-

[64] Industrial, Scientific and Medical.

[65] Die Standardfamilile IEEE 802.11 unterteilt sich in die Normen a-j sowie die geplanten Erweiterungen n und p. Das 2,4GHz ISM-Band wird dabei nicht in allen Normen verwendet, jedoch aber in den heute üblichen Normen 802.11b und 802.11g.

daten während der Datenübertragung begründbar, das bei kurzen Paketen günstiger ausfällt.

- Die Paketfehlerrate abhängig von der Kanalauslastung ist. Bei voller Auslastung ist die Wahrscheinlichkeit von Kollisionen höher.
- Bei Abständen von mehr als acht Metern zwischen IEEE 802.15.4-Knoten und IEEE 802.11b-Sendern Koexistenz beider Technologien möglich ist.

Umgekehrt wird der Einfluss von Knoten nach IEEE 802.15.4 auf andere Funktechnologien im ISM-Band als unbedeutend angesehen, da zu erwarten ist, dass der Übertragungskanal nicht permanent ausgeschöpft wird und der Standard festlegt, dass Geräte stets die niedrigst-mögliche Sendeleistung[66] benutzen sollten, um Interferenzen zu vermeiden [Vgl. IEE03, 6.9.5].

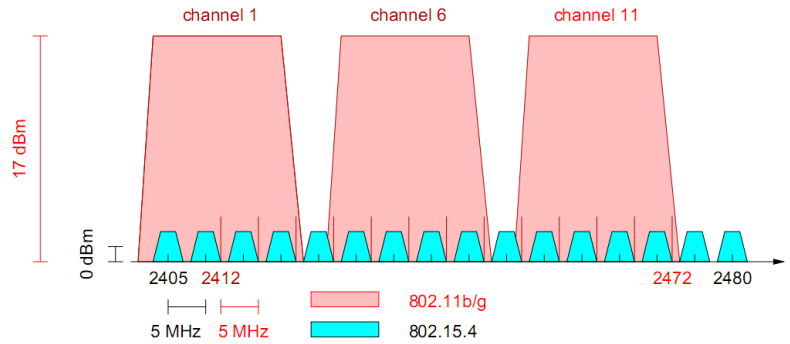


Abbildung 2.5: Überdeckung der IEEE 802.11 b/g- und IEEE 802.15.4-Kanäle [Vgl. Pet09, S. 6].

Zur Kompensation der Störeinflüsse stellt IEEE 802.15.4 einerseits das Clear Channel Assessment zur Verfügung, bei dem nur gesendet wird, wenn der Kanal auch wirklich frei ist, andererseits kann durch die Verwendung von Bestätigungen eine gestörte Übertragung erkannt und ggf. wiederholt werden. Hinsichtlich des stetig wachsenden Einsatzes von Wireless-LAN sollte daher in kritischen Anwendungen auf die freien Kanäle ausgewichen oder besser gleich das Low-Band benutzt werden, das zwar eine niedrigere Datenrate, aber eine größere Reichweite bietet. Aufgrund des hohen Störpotenzials wurden daher für den industriellen Einsatz Erweiterungen des IEEE 802.15.4-Standards oder gar komplette Alternativen entwickelt [Vgl. Kar06, Gis08], die im nächsten Kapitel kurz vorgestellt werden.

[66] Die konkrete Sendeleistung wird durch nationale Behörden festgelegt.

3 Höherschichtige Aufsätze und Alternativen zu IEEE 802.15.4

Während der IEEE 802.15.4-Standard mit den beiden untersten Netzwerkschichten die Infrastruktur zum Betrieb eines WPANs bereitstellt, wird die eigentliche „Intelligenz“ erst durch höhere Schichten realisiert, die Funktionen wie Routing, Zusammenschaltung von Netzen, Suche nach bestimmten Knoten oder Diensten implementieren. Dafür kann wahlweise entweder eine proprietäre Lösung eingesetzt oder auf bereits vorhandene Standards zurückgegriffen werden. Da im Rahmen dieser Arbeit eine proprietär implementierte Netzwerkschicht verwendet wird, werden standardisierte Ansätze nur kurz vorgestellt, um einen Einblick in mögliche Erweiterungen zu geben.

3.1 ZigBee

Die derzeit bedeutendste Erweiterung von IEEE 802.15.4 ist ZigBee[67], eine Entwicklung der im Jahr 2002 gegründeten ZigBee Alliance. In diesem Industriekonsortium haben sich mehr als 200 Unternehmen zusammengeschlossen [Vgl. ZA09b], um einen offenen Standard zur Interoperabilität von Produkten für den Heim- und Industriegebrauch zu etablieren. Pate für die Namensgebung waren Honigbienen, die in großen Schwärmen zusammenleben und durch Körperbewegungen miteinander kommunizieren. Entdeckt eine Biene eine neue Nahrungsquelle, propagiert sie diese durch einen ausgeklügelten Tanz, mit dem sie Richtung und Entfernung vom Bienenstock anzeigt [Vgl. Wik09, MN09b, SSS08].

Die erste Version der ZigBee-Spezifikation wurde im Jahr 2004 veröffentlicht, 2006 folgte eine Überarbeitung. Die ZigBee Versionen 2004 und 2006 sind zueinander inkompatibel, wodurch Zigbee-2004 praktisch nicht mehr in Gebrauch ist. Die aktuelle Version datiert auf 2007 (*ZigBee-2007*) und bietet bei voller Abwärtskompatibilität zu ZigBee-2006 erstmals zwei Stackprofile an: das schlicht „ZigBee“ genannte Basisprofil ist mit limitierten Hardwareresourcen implementierbar, wodurch es für einfache Anwendungen, beispielsweise in der Heimautomatisierung prädestiniert ist. Auf den Einsatz von Beacons wurde verzichtet, so dass das Netz nur im Unslotted Mode betrieben werden kann. Das zweite Profil, „ZigBee Pro“, bietet gegenüber dem Basisprofil eine Vielzahl optionaler Merkmale, bswp. stochastische Adressierung oder dynamische Kanalwechsel. Da ZigBee ein Produkt mehrerer Hersteller ist, arbeiten die Geräte zwar herstellerübergreifend zusammen, es existiert aber keine einheitliche API.

[67] Slogan: „Wireless Control that simply works“.

3.1.1 Rollen und Gerätearten

Basierend auf den beiden in IEEE 802.15.4 vorgeschlagenen Rollen beschreibt ZigBee drei *ZigBee-Rollen* (Vgl. Tabelle 3.1), die im Folgenden ohne den Präfix „ZigBee“ verwendet werden. Die neu hinzugekommenen *ZigBee Router* dienen dabei der Paketweiterleitung im Netz.

Rolle	*Kürzel*	*implementiert als*	*kommuniziert mit*
ZigBee Coordinator	ZC	FFD	ZR
ZigBee Router	ZR	FFD	ZC, ZR, ZED
ZigBee End Device	ZED	RFD	ZR

Tabelle 3.1: ZigBee Rollen

3.1.2 Netztopologien und Netzaufbau

ZigBee unterstützt drei verschiedene Netzwerktopologien (Vgl. Abb. 3.1), wovon die einfachste, ein *sternförmiges Netz* mit single-hop Kommunikation, direkt aus dem IEEE 802.15.4-Standard übernommen wurde. In IEEE 802.15.4 vorgeschlagene Peer-to-Peer Netze, bei denen zur Reichweitenvergrößerung Zwischenknoten (Router) in den Kommunikationspfad eingefügt werden können, realisiert ZigBee auf zwei verschiedene Arten: Die *Cluster Tree-Topologie* implementiert eine baumartige Verkettung sogenannter Cluster, welche entweder aus einzelnen Knoten oder wiederum Netzen bestehen können. Bei der *Mesh-Topologie* entsteht durch Querverbindungen zwischen den Knoten eine Vermaschung mit redundanten Pfaden, die wesentlich zur Erhöhung der Ausfallsicherheit bzw. einer Selbstheilung im Fehlerfall beitragen.

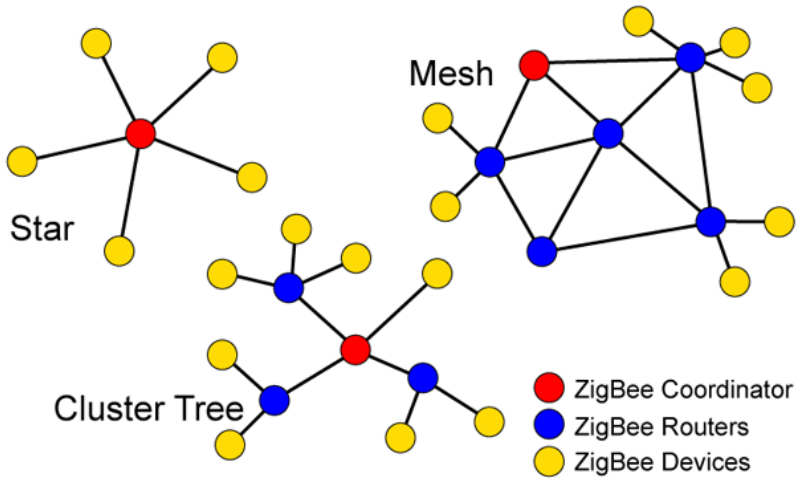


Abbildung 3.1: Topologien von ZigBee-Netzen [Vgl. MN09d].

Wie in IEEE 802.15.4 vorgeschlagen, erfolgt die Eröffnung eines Netzes durch den Coordinator auf einem bestimmten Kanal. Der Coordinator besitzt dazu im Re-

gelfall eine Liste möglicher Betriebskanäle, woraus er eine Auswahl treffen muss. Nicht verpflichtend, aber gängige Praxis, ist es, den Kanal mit den geringsten zu erwartenden Interferenzen (Störungen oder andere bereits in Betrieb befindliche PANs) auszuwählen. Um eine Menge von Kanälen nach Aktivität abzusuchen, gibt es drei verschiedene Möglichkeiten, die auch kombiniert eingesetzt werden können:

- Ein *Energy Scan* ermittelt die Feldstärke auf den Kanälen.
- Durch einen *Active Scan* wird überprüft, ob auf den Kanälen bereits PANs existieren. Dazu werden sogenannte *Beacon Requests* versendet, bei deren Empfang andere Coordinatoren antworten[68]. Bei erfolgloser Anfrage kann davon ausgegangen werden, dass auf dem Kanal kein anderes PAN betrieben wird.
- Per *Passiv Scan* wird nur nach Beacon Frames gelauscht.

Nach erfolgreichem Start vergibt der Coordinator eine PAN-ID. Deren Länge unterscheidet sich je nach verwendetem Stackprofil: bei *ZigBee* werden herkömmliche 16 Bit lange PAN-IDs verwendet, *ZigBee Pro* nutzt dazu 64 Bit[69].

3.1.3 Der Zigbee-Protokollstack

Direkt auf den beiden durch IEEE 802.15.4 definierten Schichten PHY und MAC aufsetzend, spezifiziert der ZigBee Protokollstack zwei neue Schichten: den NWK sowie den APL[70] (Vgl. Abb. 3.2). Analog IEEE 802.15.4 gibt es auch hier zwischen den Schichten stets zwei Dienstzugriffspunkte, jeweils einen für Datenaustausch und einen für Managementfunktionen. Die bereits in IEEE 802.15.4 verwendete Kapselung zu transportierender Daten durch PDUs wird auf die beiden neuen Schichten ausgedehnt. Gemäß Spezifikation hat ein vollständig implementierter ZigBee Stack (FFD) eine Binärgröße von weniger als 32 kB, ein RFD kommt mit etwa 6 kB aus.

3.1.4 Network Layer

Der Network Layer definiert die bereits eingangs erläuterten Netzwerktopologien. Darauf aufbauend legt er Funktionen zum Topologiemanagement fest und beschreibt, wie die Paketweiterleitung (Routing) innerhalb der gewählten Topologie erfolgt. Dazu benennt er Mechanismen zur Routensuche, zur Etablierung von Backup-Routen bei Verwendung vermaschten Routings und zur Entfernung von Paketduplikaten. Werden auf MAC-Ebene Sicherheitsfunktionen eingesetzt,

[68] Durch einen Beacon Frame.
[69] Extended PAN-ID.
[70] Application Layer.

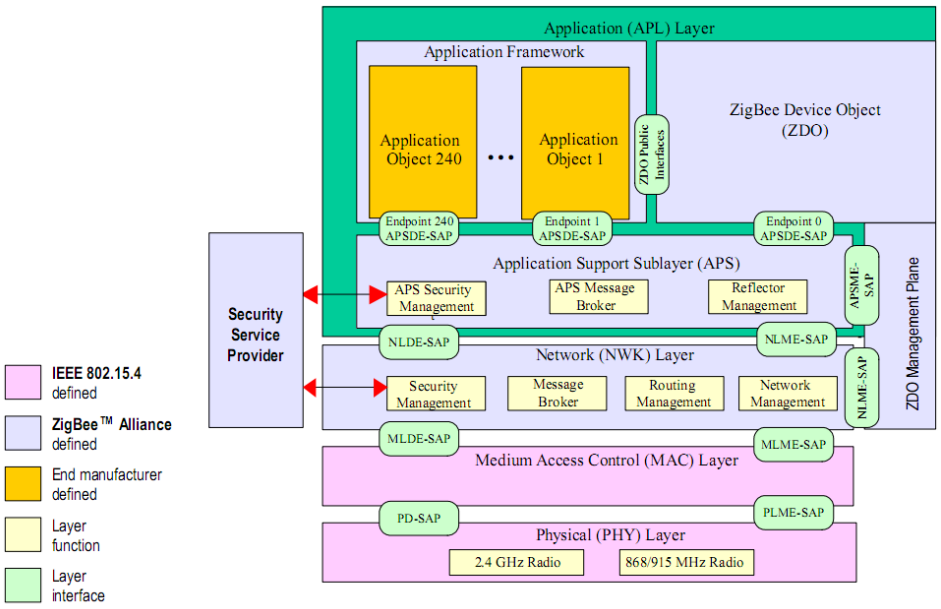


Abbildung 3.2: Der ZigBee Protokollstack [Vgl. ZA09c].

übernimmt er die dazu erforderliche Schlüsselverteilung. Alle die Arbeitsweise des Network Layers betreffenden Eigenschaften werden innerhalb der Network Layer Information Base (NIB) in sogenannten Managed Objects (MOs) gekapselt. Die Adressierung von Knoten erfolgt über logische Netzwerkadressen. Als Adressierungsmodi stehen Unicast, Multicast oder Broadcast zur Verfügung, wobei sich die letzten beiden auf Gruppen von Knoten einschränken lassen[71]. Da in ZigBee-basierten Netzen eine Multi-Hop-Kommunikation möglich ist, ermöglicht der Network Layer die *Ende-zu-Ende Bestätigung* versandter Nachrichten. In Broadcasts kann zur Reduzierung der Netzlast die maximale Anzahl zu durchlaufender Hops angegeben werden. Bei bekannter Route zwischen zwei Knoten und unter der Annahme, dass keine Übertragungswiederholungen erforderlich sind, beträgt die Latenz der Datenübertragung etwa 10 ms/Hop, wobei die maximale Reichweite auf 10 Hops limitiert wurde.

3.1.5 Application Layer

Die Applikationen, die das eigentliche Verhalten des Netzes widerspiegeln – „Das Netz ist die Anwendung“ –, sind im Application Layer angesiedelt, der dazu nochmals unterteilt ist:

- Das *Application Framework* beinhaltet eine Ablaufumgebung für Applikationen, die in sogenannten *Application Objects* vorliegen. Davon können auf einen Knoten mehrere vorhanden sein. Applikationen bieten letztlich den Zugriff auf die Hardware des Knotens.

[71] Beispielsweise auf alle derzeit nicht schlafenden Knoten.

- Der Application Support Layer (APS) dient der Anbindung der Applikationsobjekte an den Network Layer sowie der Adressierung und Bindung zueinander passender Applikationen zwischen Endgeräten.
- Das ZigBee Device Object (ZDO) ist eine spezielle Anwendung, die den logischen Status des Knotens innerhalb des Netzes verwaltet. Sie ist verantwortlich für die Suche nach, den Beitritt in oder den Wiederaustritt aus einem Netz. Bei Inbetriebnahme des Knotens initialisiert sie alle anderen Schichten, abhängig von der Zugehörigkeit zu einem Netz und dessen Parametern startet oder beendet sie verfügbare Anwendungen.
- Das *ZDO Management* bietet Funktionen zum Registrieren von Anwendungen und der Suche von Geräten mit speziellen Eigenschaften (Device-/Service Discovery).

3.1.6 Endpunkte, Cluster und Profile

Da auf einem Knoten zeitgleich potentiell mehrere Anwendungen parallel verfügbar sein können, bedarf es einer Möglichkeit, diese über die Knotenadresse hinaus adressieren zu können. Dazu werden *Endpunkte*[72] aus dem Intervall [0, 255] verwendet[73], die dynamisch auf Applikationsobjekte abgebildet werden.

Die Realisierung der Applikationsschicht ist nicht direkter Bestandteil der ZigBee-Spezifikation, sondern wurde in sogenannte *Profile* ausgelagert, die durch das ZDO lediglich innerhalb der Anwendungsschicht verwaltet werden. Profile gruppieren verwandte Applikationen, in dem sie aufgabenbezogen Aufbau, Funktionalität, Verhalten und Interaktion der einzusetzenden Geräte beschreiben und diese Gruppen an definierte Endpunkte koppeln. Profile werden von der ZigBee Alliance vergeben und beständig erweitert, ihr Spektrum unterteilt sich in proprietäre und allgemein nutzbare Bereiche, wobei durch letztere eine herstellerübergreifende Interaktion von Knoten ermöglicht werden soll.

In sogenannten *Clustern* werden die in speziellen Einsatzbereichen gebräuchlichen Datentypen und Kommandos zusammengefasst. Cluster werden in der ZCL[74] verwaltet und zur Spezifikation der Anwendungsschnittstelle von Profilen eingesetzt.

Als komplexes Beispiel sei an dieser Stelle das HVAC[75]-Profil genannt, unter dem Funktionen zum Betrieb von Klimaanlagen gebündelt werden. Mögliche Funk-

[72] vergleichbar mit TCP/IP, wo unter Ports Dienste zur Verfügung gestellt werden.
[73] der Endpunkt 0 ist dabei für Managementfunktionen (ZDO) reserviert, 255 für Broadcast und 241-254 für zukünftigen Einsatz, wodurch letztendlich nur noch die Endpunkte 1-240 für Applikationen zur Verfügung stehen.
[74] ZigBee Cluster Library.
[75] Heating, Ventilating and Air Conditioning.

tionen einer Klimaanlage, wie „an“, „aus“, „wärmer“, „kälter“, „Regelung auf Temperatur x innerhalb Zeitspannne y“ werden dabei in Clustern gruppiert. Ein dazu passendes minimalistisches Programmierbeispiel wird in [Aue09] vorgestellt.

3.1.7 Sicherheit

Wie bereits in Kapitel 2.6 beschrieben, muss bei Funknetzen besonders auf Vertraulichkeit und Authentikation übertragener Daten Wert gelegt werden. ZigBee bietet hierzu drei grundsätzliche Mechanismen an:

- ACLs[76] werden im Coordinator vorgehalten und beinhalten pro vorhandenem Netzknoten einen Eintrag mit der zugeordneten Sicherheitsstufe. Die Sicherheitsstufen sind dabei orthogonal wählbar aus 'Vertraulichkeit' und 'Authentikation', wodurch sich vier mögliche Kombinationen ergeben: keine Sicherheit, nur Vertraulichkeit, nur Authentikation oder beides.
- Der in allen Knoten vorhandene *Packet Freshness Timer* wird genutzt, um den Frame Counter eintreffender Paketen zu überprüfen. Eine dabei entdeckte abgelaufene Framenummer führt zum Verwerfen des zugehörigen Pakets. Dieses Vorgehen versucht Replay-Attacken zu verhindern.
- Mittels kryptografischer Verfahren auf Basis von AES lassen sich Nachrichten signieren oder vollständig verschlüsseln.

Da in Zig-Bee Netzen vorrangig Steuerbefehle transportiert werden und ihr Effekt ohnehin meist sichtbar wird, ist eine Verschlüsselung nicht besonders wirkungsvoll [Vgl. Wob09]. Um dennoch zu gewährleisten, dass Steuerbefehle von einem dazu berechtigten Netzteilnehmer versendet wurden, bietet ZigBee Mechanismen zur Absenderauthentifikation: Über der auf Netzwerkschicht im Nachrichtenheader verwendeten 32-Bit-Folgenummer, den restlichen Feldern des Header, der eigentlichen Nachricht und eines nur beiden Kommunikationspartnern bekannten Schlüssels wird eine kryptografisch gesicherte Prüfsumme gebildet (CBC-MAC[77])und mit der Nachricht versandt. Auf der Empfängerseite erfolgt diese Berechnung erneut und wird mit der übertragenen Prüfsumme verglichen, wodurch sich praktisch alle relevanten Angriffsarten verhindern lassen.

Soll der volle Schutz durch eine komplette Verschlüsselung der Nachrichten erreicht werden, verwendet ZigBee eine Stromchiffrierung, die basierend auf AES mittels des CCM*[78]-Verfahrens vorgenommen und zusätzlich durch eine CRC-

[76] Access Control Lists.
[77] Cipher Block Chaining Message Authentication Code.
[78] Counter Mode with CBC-MAC.

Prüfsumme während der Übertragung ergänzt wird. Zur Umgehung der ebenfalls in Kapitel 2.6 beschriebenen unbekannten Adressierung, die in potentiell zwischenliegenden Knoten zum Routing erforderlich ist, bleibt der Nachrichtenheader dabei stets unverschlüsselt.

Generell vertraut ZigBee darauf, dass Schlüssel geheim gehalten werden. Es arbeitet nach *Open Trust*-Prinzip, wonach sich einzelne Schichten und Anwendungsobjekte eines Endgerätes gegenseitig vertrauen. Dies vereinfacht die Ver- und Entschlüsselung, da sie nur einmal erfolgen muss. Die Verwaltung und Verteilung von Schlüsseln erfolgt durch ein *Trust-Center*, das meist auf dem Coordinator realisiert wird.

Bezüglich verwendeter Schlüssel sieht ZigBee einen *Master-Key* vor, der bei der Initialisierung des Knoten an ihn übertragen wird. Im einfachsten Fall gibt es weiterhin einen *Network-Key*, der austauschbar ist (z.B. in regelmäßigen Intervallen oder bei Kompromittierung). Dieser Network-Key ist gleichzeitig der Gruppenschlüssel und wird bspw. zur Verschlüsselung von Broadcast-Nachrichten, wie Routinganfragen, verwendet. Optional gibt es noch *Link-Keys*, die zur individuellen Verschlüsselung von Unicast-Verbindungen zufällig erzeugt werden. Der Schlüsselaustausch erfolgt dabei stets durch das sogenannte Symmetric-Key Key Establishment Protocol (SKKE), bei dem der zu übertragende Schlüssel durch den Master Key verschlüsselt wird. Schläft ein Gerät während des Schlüsselwechsels, wird der neue Schlüssel erst dann verwendet, wenn ihn alle Geräte erhalten haben. Selbst wenn es also einem Angreifer gelingen sollte, in Besitz des Master-Keys zu gelangen, so muss er immer noch die Schlüsselübertragung abpassen.

ACK-Frames werden nicht verschlüsselt, da diese nur auf MAC-Ebene angesiedelt sind. Dies stellt immer noch eine Möglichkeit für einen DoS-Angriff dar.

Die in ZigBee implementierten Sicherheitsmerkmale sind keineswegs perfekt, stellen aber angesichts der einerseits angestrebten geringen Komplexität des Protokolls und andererseits einer möglichst hohen Sicherheitsstufe einen guten Kompromiss dar. Ihre Spezifizierung ist sogar so ausführlich, dass auch herstellerübergreifend Schlüssel problemlos ausgetauscht werden können. Da kryptografische Operationen aufwändig sind, wird zunehmends durch die Modulhersteller eine Hardwareunterstützung der Chiffrierung und zur Erzeugung von Zufallszahlen angeboten.

3.1.8 praktischer Einsatz

Da ZigBee zur Funkübertragung auf den in IEEE 802.15.4 definierten PHY zurückgreift, gelten die bereits in Kapitel 2.7 getroffenen Aussagen hier ebenfalls.

Die ZigBee Alliance zweifelt diese Ergebnisse jedoch an [Vgl. ZA09a]. Durch die bereits in IEEE 802.15.4 definierte Frequenzagilität können ZigBee-Knoten jedoch auf einen ungestörten Übertragungskanal ausweichen. Pro übertragenem Paket wird auf MAC-Ebene die Verbindungsqualität ermittelt, wodurch eine tendenziell demnächst abbrechende Verbindung abgeleitet werden kann. In kritischen Anwendungen stellt daher ein vermaschtes Netz eine zusätzliche Sicherheitsebene dar, da dort bedarfsweise auf alternative Routen zurückgegriffen werden kann. Jeder zusätzliche Hop wird jedoch durch eine weitere Übertragungslatenz erkauft. Für zeitkritische Übertragungen lassen sich zumindest innerhalb eines Clusters garantierte Übertragungszeiten durch eingeräumte Zeitschlitze erzielen, die sich jedoch nicht clusterübergreifend fortsetzen.

Bezüglich der logischen Netztopologie ist der Coordinator immer noch der Single Point of Failure, jedoch lassen sich auf Ebene des Network Layer Funktionen implementieren, bei denen Backup-Coordinatoren vorgehalten werden, die den Coordinator bei Ausfall ersetzen. Derartige Szenarien erfordern allerdings einen beträchtlichen Aufwand, da permanent der Zustand des Coordinators auf dem Backup Coordinator zu replizieren ist.

Die herstellerübergreifende Interoperabilität von ZigBee-Knoten wird durch die Vermischung von ZigBee mit dem IEEE 802.15.4 Standard sowie die teilweise inkompatiblen Versionen der ZigBee-Spezifikation erschwert.

Zur Koppelung an fremde Netztechnologien lassen sich Gateways installieren. Auf dem Markt verfügbare ZigBee-Knoten mit USB-Anbindung ermöglichen so neben der Überwachung dedizierter Knoten die Integration eines kompletten ZigBee-Netzes in ein LAN.

3.2 6LoWPAN

Mit dem Akronym 6LoWPAN[79] wird ein Verfahren zur Integration von WPAN-Knoten in IPv6-Netzen durch Adaption eines TCP/IP-Stacks auf die beiden durch 802.15.4 definierten unteren OSI-Schichten beschrieben. Die Betreuung erfolgt dabei durch die IETF[80], formell handelt es sich hierbei nur um eine RFC[81].

Da die IEEE 802.15.4-Spezifikation den im OSI-Layer 2 zur Anpassung unterschiedlicher Technologien verwendeten SSCS-Sublayer aber ausklammert, muss dieser hier selbst implementiert werden. Von besonderer Bedeutung ist hier die Fragmentierung der 1280 Byte umfassenden MTUs[82] auf Vermittlungsschicht in

[79] IPv6 over Low power WPAN.
[80] Internet Engineering Task Force.
[81] Request For Comments.
[82] Maximum Transfer Units.

maximal 127 Byte große Frames in der Sicherungsschicht, verbunden mit einer geeigneten Headerkompression. Ein weiterer Unterschied betrifft die Adressierung, die bei IPv6 mit 128 Bit, bei WPANs mit 64 Bit erfolgt.

3.3 Sonstige Lösungen

Neben den bisher erläuterten existiert eine Reihe weiterer, aber weniger populärer Lösungen, die entweder ebenfalls auf dem IEEE 802.15.4 Protokollstack basieren (⋆) oder proprietäre Implementierungen (◦) darstellen. Aus Platzgründen kann hier keine detaillierte Beschreibung erfolgen; die folgende Auflistung erhebt keinen Anspruch auf Vollständigkeit:

- ⋆ *WirelessHART* ist ein System für drahtlose Feldbusse, welches ein Kanalsprungverfahren für IEEE 802.15.4 realisiert, dies aber (noch) nicht basierend auf detektierten Interferenzen.
- ◦ *ISA100.11a*, ebenfalls ein drahtloses Feldbussystem, steht in direkter Konkurrenz zu WirelessHART.
- ◦ *Wireless Profibus* ist die drahlose Version des Feldbussystems Profibus.
- ◦ *Z-Wave* ist ein zu ZigBee konkurrierender Standard und momentan der Marktführer im Bereich der Heimautomatisierung. Auch hier verbirgt sich dahinter ein Industriekonsortium, die Z-Wave Alliance.
- ◦ *Wibree* ist eine Ergänzung zu Bluetooth, die dessen Energiebedarf nochmals senkt und daher auch unter dem Namen *ultra low power Bluetooth* firmiert. Dadurch, dass sich Bluetooth-Knoten unverändert weiterverwenden lassen, stellt es ebenfalls eine ernstzunehmende Alternative zu ZigBee dar.
- ⋆ *SimpliciTI* ist eine partielle Implementierung des IEEE 802.15.4-Standards von Texas Instruments.
- ◦ *Synkro* ist ein Protokoll zur Steuerung von Konsumelektronik-Geräten.
- ◦ *MiWi* stellt eine Implementierung eines Peer-to-Peer Mesh-Protokolls der Firma MicroChip dar, die speziell auf deren Transceiver optimiert wurde.
- ◦ *EC-Net* von Freescale kommt in Funkfernbedienungen zum Einsatz.
- ⋆ RF4CE[83] beschreibt einen Standard für Funkfernbedienungen und ist mittlerweile Teil der ZigBee-Spezifikation.

[83] Radio Frequency for Consumer Electronics.

4 Funkmodule in der Praxis

4.1 Marktüberblick

IEEE 802.15.4-konforme und ZigBee-zertifizierte Funkmodule sind in einer breiten Vielfalt verfügbar. Überdies werden teilweise komplette Entwicklungsumgebungen oder aus mehreren Modulen bestehende Starter-Kits angeboten. Der Markt ist sehr schnelllebig, beinahe wöchentlich verändert sich das Angebot. Unbekannte Firmen drängen auf den Markt, bestehende werden durch Konkurrenten übernommen, Kooperationen oder Insolvenzen bekanntgegeben. Aus genannten, aber vor allem aus Platzgründen kann an dieser Stelle lediglich ein genereller Überblick des vielfältigen Angebots gegeben werden, eine ausführliche Marktübersicht mit Stand 2006 findet sich in [Ead07]. Beispielhaft werden jedoch im nächsten Kapitel die Hard- und Software-Lösungen der Firma Atmel vorgestellt, da diese zur der Bearbeitung der Aufgabe eingesetzt wurden.

Wie eingangs beschrieben, bestehen Funkmodule aus den zwei Hauptkomponenten *Mikrocontroller* und *Funktransceiver*[84], deren bekannteste Anbieter Atmel, MicroChip, Motorola/Freescale, Texas Instrument/Chipcon und ZMD sind. Der Trend geht dahin, beide Komponenenten auf einem Chip zu vereinigen, erste Lösungen dazu sind bereits erhältlich [Vgl. FS09].

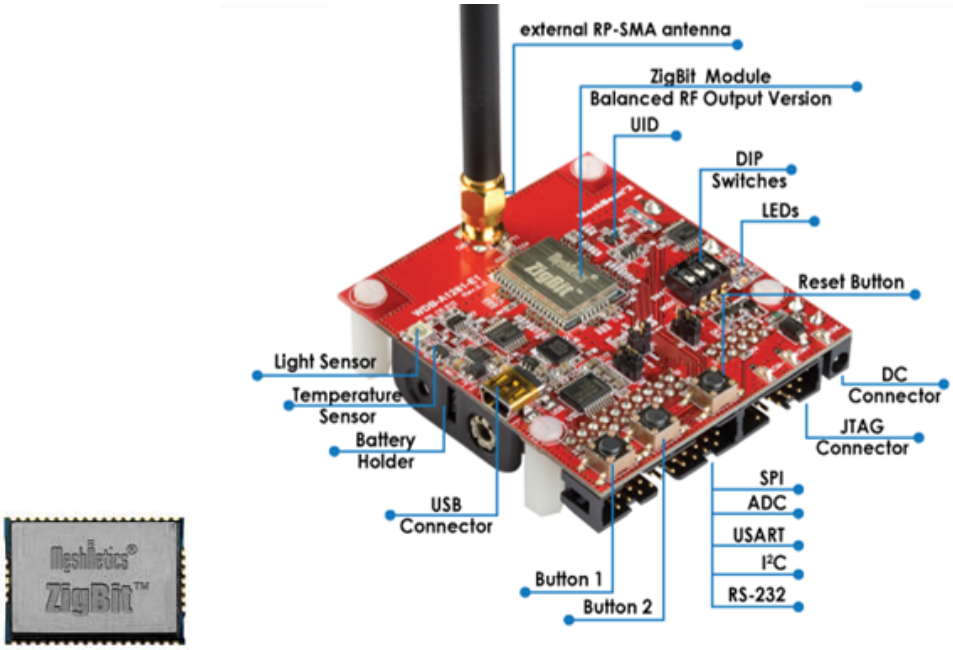


Abbildung 4.1: ZigBit-Modul [MN09c] und MeshBean Development Board [MN09a].

[84] Die Ankopplung des Transceivers an den Mikrocontroller erfolgt dabei meist per SPI (Serial Peripheral Interface). Als mögliche Paarungen kommen Produkte eines Herstellers in Frage, es gibt aber auch Firmen, die nur Funktransceiver anbieten, wobei dann Mikrocontroller eines Fremdherstellers eingesetzt werden.

Im Herstellungsprozess werden entweder nur Mikrocontroller und Transceiver auf einen Träger aufgebracht, welcher als Baustein für weitere Entwicklungen verwendet werden kann (Vgl. Abb. 4.1 links) oder aber auf eine Platine aufgesetzt, die als voll einsatzfähiges Modul verwendet werden kann (Vgl. Abb. 4.1 rechts) und dafür eine breite Palette an Schnittstellen bietet. Zur Hochfrequenzankopplung kommen entweder Leiterschleifen oder externe Antennen in Betracht. Entgegen dem eigentlichen Designziel, dass WPAN-Knoten immer energiesparsam und klein sein sollen, gibt es auch Module, welche die Größe einer Pralinenschachtel erreichen, einen Leistungsverstärker[85] beinhalten und deshalb nur noch per Stromnetz gespeist werden können [Vgl. Ead07, S. 167],[Vgl. MN09a]. Zur Produktion und Distribution der Module werden oft externe Firmen herangezogen.

Selbst hinsichtlich des Lieferumfanges unterscheiden sich die Module enorm. Während einige Hersteller sich lediglich auf Hardware beschränken, geben andere einen Protokollstack oder sogar Beispielapplikationen mit, die wiederum selbst oder extern entwickelt wurden. In diesem Zusammenhang sei erwähnt, dass gleiche Module unter verschiedenen Bezeichnungen mit unterschiedlicher Ausstattung durch Wiederverkäufer angeboten werden. Der Einsatz einer herstellerübergreifenden Lösung ist insofern problematisch, dass sich einzelne Hersteller gegenseitig für Bugs verantwortlich machen könnten. Kommt jedoch eine Lösung „aus einer Hand“ (Transceiver, Mikrocontroller und Software) zum Einsatz, so ergeben sich klare Vorteile für den Kunden: eine bessere Abstimmung der Einzelkomponenten aufeinander, kurze Entwicklungszyklen, und im Bedarfsfall ein allumfassender Support.

Bezüglich der Implementierung der Knotenfunktionalität kann zwischen zwei Extremen unterschieden werden:

- Eine weitgehende Implementierung in Hardware erfordert einen hohen Entwicklungsaufwand, nachträglich erkannte Fehler können nur in der nächsten Release berücksichtigt werden, nach Änderungen am Protokoll lässt sich ein Modul nicht mehr verwenden.
- Die umgekehrte Möglichkeit, Implementierung vollständig in Software[86], bietet hingegen extreme Flexibilität. Dadurch, dass hier jedoch selbst die gesamte Signalverarbeitung per Software gelöst wird, ist ein immenser Rechenaufwand erforderlich, der dem Konzept energiesparsamer und preiswerter Knoten nicht gerecht wird.

[85] Sowohl IEEE 802.15.4 als auch die ZigBee-Spezifikation geben keine maximale Sendeleistung der Module vor, üblich sind jedoch 1mW.

[86] sogenanntes „Software-Defined Radio“.

Bei IEEE 802.15.4-konformen Funkmodulen wurde daher ein Kompromiss gewählt: Der Transceiver ist ein speziell optimierter Schaltkreis, der große Bereiche des Physical Layers implementiert und höheren Schichten Hardwareunterstützung bietet, beispielsweise bei der Verschlüsselung oder der Übertragungssicherung. Aufgrund der unterschiedlichen Charakteristiken der beiden PHY-Layer[87] wird in der Regel nur einer der beiden durch den Chip unterstützt.

So breitgefächtert die Möglichkeiten der Hardware-Realisierung sind, so unterschiedlich können auch eingesetzte Software und Programmierparadigmen ausfallen: Zum Betrieb benötigt ein Knoten einen Protokollstack, darauf aufsetzend eine eigene Applikation sowie eine Ablaufumgebung. Für letzteres kommen proprietäre Eigenentwicklungen oder der Einsatz eines vollwertigen (Echtzeit-)Betriebssystems[88] in Frage. Der Protokollstack kann quelloffen sein oder lediglich als Bibliothek vorliegen, die Schnittstelle zu eigenen Anwendungen wahlweise per low-level Zugriff oder über Frameworks realisiert werden. Einige Hersteller bieten gar keine Möglichkeit zur Integration eigener Anwendungen, sondern liefern nur vorkompilierte Firmware mit, die erst durch einen angeschlossenen PC gesteuert werden kann [Vgl. Ead07, S.170,178]. Obwohl die Schnittstellen aller Protokollschichten in den zugrunde liegenden Normen beschrieben sind, gibt es derzeit keine gemeinsame API. Eine Anwendung kann daher nicht einfach zwischen Produkten unterschiedlicher Hersteller portiert werden. Dies ist vermutlich so beabsichtigt, um den Kunden an eine Hardwareplattform zu binden.

4.2 Der Atmel-Weg

Die Atmel Corporation [Vgl. Atm09g] ist ein weltweit agierender Hersteller von integrierten Schaltkreisen, der vor allem für seine AVR-Mikrocontrollerfamilie bekannt ist. Die Produktpalette umfasst aber weit mehr, so gehören beispielsweise auch Speicherschaltkreise, Netzwerk- und Funkkomponenten dazu, aus denen Atmel komplexe Systeme zusammenstellt [Vgl. Atm09f].

Im Funksektor bietet Atmel drei verschiedene Transceiver an, die in Kombination mit einem Atmel Mikrocontroller auch als Bundle erhältlich sind [Vgl. Atm09c]. Seit der Übernahme der Firma MeshNetics sind die bereits im vorigen Kapitel vorgestellten ZigBit Module Teil der Produktpalette [Vgl. Atm09e]. Als Demonstrations- und Entwicklungsplattformen bietet Atmel die *Z-Link*- und die *RAVEN*-Familie an. Zu letzterer gehört ein *RZ USB-Stick*, durch den eine einfache Anbindung eines PC an einen Funkknoten realisiert werden kann. Mit dem *AVR*-

[87] Frequenzband, Modulation.

[88] Bspw. *TinyOS*, ein event-basiertes open-source Betriebssystem für embedded Hardware, welches das ZigBee-Protokoll unterstützt.

Studio steht kostenfrei eine IDE[89] zur Verfügung. Für IEEE 802.15.4, ZigBee und 6LoWPAN sind Referenzimplementierungen erhältlich [Vgl. Atm09d], die in den folgenden Kapiteln näher beschrieben werden. Letztere stehen zwar ebenfalls kostenfrei zur Verfügung, ihre Lizenzbestimmungen erlauben jedoch nur den Einsatz auf Atmel-basierter Hardware.

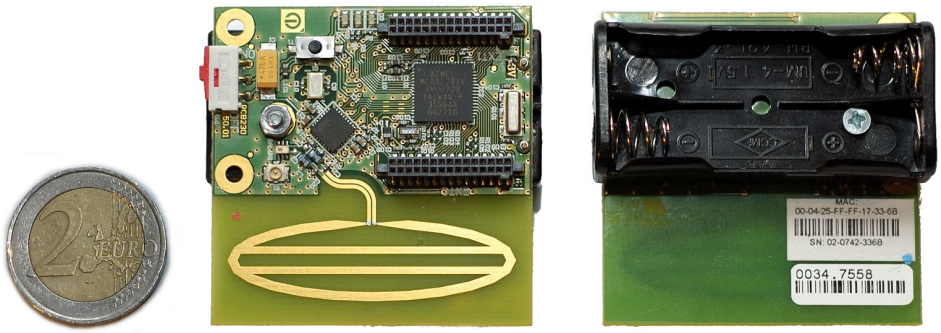


Abbildung 4.2: dresden elektronik Radio Controller Board, Revision 3.1.

dresden elektronik tritt als Drittanbieter auf und produziert basierend auf den Atmel-Chips eigene Funkmodule [Vgl. DE09a]. Während der Entwicklung eingesetzt wurden sogenannte RCBs[90](Vgl. Abb. 4.2). Der darauf befindliche Mikrocontroller, ein ATMega1281, bietet 128 kB Flash-Speicher, 8 kB SRAM und kann unter der gegebenen Betriebsspannung von 3,3 V mit zu 8 MHz getaktet werden. Der eingesetzte ATRF230-Funktransceiver bedient das High Band und tritt in zwei Revisionen auf[91], wobei letzterer optional im sogenannten *Extended-Mode*[92] Hardwareunterstützung für zwei Merkmale des IEEE 802.15.4 MAC-Layers bietet. Zur Stromversorgung des Funkmoduls, dem Programmieren der Firmware sowie zur Verbindung mit dem PC wurde ein eigens von Dresden-Elektronik entwickeltes *Sensor-Terminal-Board* verwendet.

4.2.1 IEEE 802.15.4-Protokollstack-Referenzimplementierung

Atmel stellt mit dem *IEEE 802.15.4 MAC Software Package for AVR Z-Link* [Vgl. Atm09b] eine Referenzimplementierung eines IEEE 802.15.4-konformen Prokollstacks bereit, die komplett selbst in der Programmiersprache C entwickelt wurde und neben den vollständigen Quellen Beispielapplikationen und eine ausführli-

[89] Integrated Development Environment.
[90] Radio Controller Boards.
[91] ATRF230A und ATRF230B, wobei erstere nicht mehr hergestellt wird.
[92] *Auto-ACK* versendet automatisch ggf. erforderliche ACK-Frames, *Auto-Retry* ist das Pendant dazu - es versendet Frames solange, bis entweder ein entsprechender ACK-Frame empfangen wurde oder ein Wiederholungzähler abgelaufen ist (Vgl. Kapitel 2.5.2).

che Entwicklerdokumentation enthält. In dieser Arbeit wurde die Version 2.2.0 verwendet, mittlerweile liegt die Version 2.3.1 vor [Vgl. Atm09b]. Im Folgenden wird das Softwarepaket der Einfachheit halber als „Stack“ bezeichnet. Die Implementierung ist dabei nicht völlig frei von Fehlern, gilt aber im Vergleich zu anderen Implementierungen als eine der besten am Markt verfügbaren.

Angesichts der Vielfalt an Atmel-Produkten[93] bestand das Designziel darin, sämtliche daraus bildbare Kombinationen durch nur einen Stack zu unterstützen. Das Softwarepaket ist daher extrem flexibel konfigurierbar, aus Sicht des IEEE-Standards unterstützt es dabei den Funktionsumfang eines FFD.

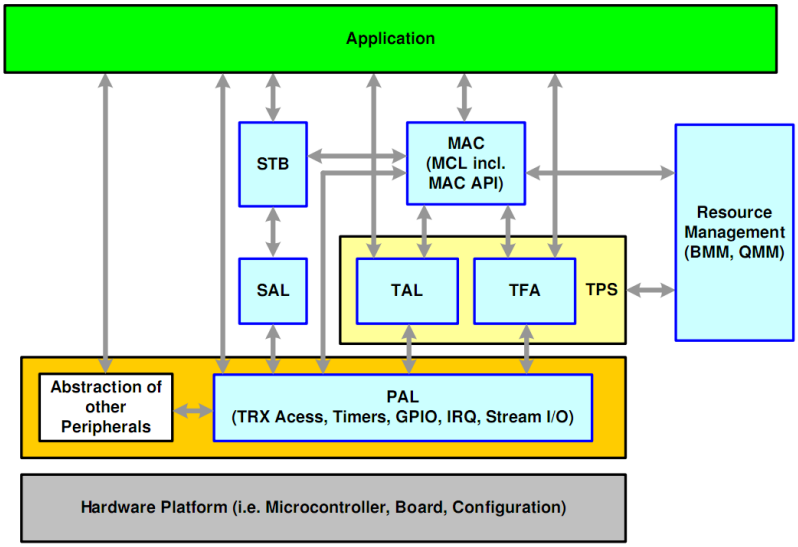


Abbildung 4.3: Architektur der Atmel IEEE 802.15.4-Protokollstack-Referenzimplementierung.

Konzeptionell gliedert es sich in Module, die einerseits die durch den 802.15.4-Standard festgelegten Funktionen der PHY- und MAC-Schicht übernehmen, andererseits in Form von Abstraktionsschichten der Anpassung an die konkret verwendete Hardware dienen (Vgl. Abb. 4.3). Wichtige funktionale Blöcke sind dabei:

- *MCL* (MAC Core Layer) - implementiert die Funktionen nach IEEE 802.15.4,
- PAL (Platform Abstraction Layer) - abstrahiert den generellen Zugriff auf die Zielhardware wie I/O, Interrupts und Timer,
- *TAL* (Transceiver Abstraction Layer),
- *SAL* (Security Abstraction Layer) und *STB* (Security Toolbox) - bieten Low-Level-Zugriff auf Sicherheitsfunktionen wie AES-Verschlüsselung,
- *BMM* (Buffer Management) und *QMM* (Queue Management) - Verwaltung von Puffern und Warteschlangen.

[93] Transceiver, Mikrocontroller, Boards.

In Abhängigkeit von der zu erstellenden Applikation müssen dabei nicht immer alle Module genutzt werden. Auch wenn es für ein Funkmodul nicht sinnvoll erscheint, kann eine Applikation bspw. lediglich aufbauend auf dem PAL arbeiten – dann kann das Modul zwar nicht funken, man kann aber die GPIO-Funktionalität verwenden.

Den begrenzten Hardwareresourcen eines Mikrocontrollers ist es geschuldet, dass Speicher nicht in beliebigen Mengen zur Verfügung steht. Bei der Initialisierung des Stack wird daher einmalig eine Menge von Puffern verschiedener Größen angelegt und in einer Liste verkettet. Aus dieser Liste freier Puffereinträge können sie bedarfsweise entnommen werden, um nach Abschluss der Verwendung wieder darin aufgenommen zu werden. Puffereinträge werden in Zusammenhang mit Queues verwendet, die zur Entkoppelung der Kommunikation eingebaut wurden.

Die Initialisierung des Stack erfolgt durch eine Menge von Funktionsrufen aus der `main()`-Funktion heraus, anschließend können benutzerdefinierte Initialisierungen vorgenommen werden. Den letzten Funktionsblock der `main()`-Funktion bildet die Ablaufsteuerung. In einer Endlosschleife wird praktisch nur eine Funktion aufgerufen, die in weitere Unterfunktionen absteigt. All diese Funktionsaufrufe haben die alleinige Aufgabe, die Queues, welche logisch betrachtet zwischen den Modulen platziert sind, auf hinzugekommene Einträge zu überprüfen und ggf. zu bearbeiten. Eine gewisse Asynchronität wird durch Hardware-Interrupts erreicht, welche die eigentliche Ablaufsteuerung in unregelmäßigen Abständen unterbrechen können. Wiederkehrende Aufgaben lassen sich durch Timer realisieren. Die Interaktion des Stack mit Knotenapplikationen wird später anhand eines Beispieles erläutert.

Der Buildvorgang setzt unter Windows die Installation der winavr-Umgebung [Vgl. Win09] voraus und basiert auf Makefiles, durch die neben der eingesetzten Hardwarebasis weitere Parameter des Stack vorgegeben werden, beispielsweise, ob Beacons zu unterstützen sind. Die Binärgrößen einer übersetzten Anwendung bewegen sich je nach Umfang des genutzten Funktionsumfangs und eigenen Implementierungen zwischen 20 und 40 kB. Die eigentliche Übertragung auf das Funkmodul[94] erfolgt schließlich per ISP[95].

4.2.2 BitCloud – der Atmel Zigbee-Stack

BitCloud wurde ursprünglich von MeshNetics entwickelt, mittlerweile erfolgt der Vertrieb durch Atmel. Der Stack bietet Unterstützung für beide Stack-Profile.

[94] genauer: den darauf befindlichen Mikrocontroller.
[95] In-System Programming.

Im Gegensatz zum IEEE 802.15.4-Protokollstack steht BitCloud nur als Bibliothek zur Verfügung, eine Offenlegung der Quellen ist jedoch geplant. Der Zugriff erfolgt über eine C-API, die Hardwareunterstützung umfasst alle derzeit angebotenen Funkmodule von Atmel. Wie auch bei der IEEE 802.15.4-Referenzimplementierung werden Beispiele mitgeliefert. Da in der vorliegenden Arbeit ZigBee nur theoretisch betrachtet wird, soll an dieser Stelle keine weitere Vertiefung erfolgen.

4.2.3 RUM – Die Atmel 6LoWPAN-Implementierung

Zur Implementierung des 6LoWPAN-Verfahrens stellt Atmel RUM[96] bereit [Vgl. Atm09a]. Dabei kommt ein *modifizierter Stack* zum Einsatz, bei dem die MAC-Schicht um Routingfunktionen ergänzt wurde, die laut dem IEEE 802.15.4-Standard erst auf Netzwerkschicht angesiedelt sind. Bei RUM wird auch nicht die soeben beschriebene IEEE 802.15.4-Referenzimplementierung eingesetzt, sondern nur eine auf die benötigten Funktionen reduzierte Variante.

Alternative Implementierungen erfolgen durch *Archrock* [Vgl. IES08] oder *Contiki* [Vgl. Dun09]. Contiki ist ein embedded Echtzeit-Betriebssystem unter Open-Source-Lizenz. Es unterstützt hardwareseitig mehrere Mikrocontroller-Plattformen, Atmel setzt es auf der firmeneigenen AVR-Architektur ein. Weiterhin bietet es Netzwerkunterstützung für IP in den Versionen 4 und 6 sowie nach 802.15.4 WPAN.

4.3 Sniffer

Extrem hilfreich bei der Netzwerkanalyse[97] sind sogenannte *Sniffer*[98]. In WPANs nach IEEE 802.15.4 werden dazu reguläre Funkknoten eingesetzt, die mit einer speziellen Firmware versehen und an einen PC angeschlossen werden. Prinzipbedingt lauscht ein Funkknoten während des Betriebs das Übertragungsmedium permanent ab und empfängt so *alle* gesendeten Nachrichten, blendet aber nicht an ihn adressierte wieder aus. Die hier eingesetzte Knotenfirmware unterbindet dies und leitet alle Pakete an den PC weiter. Beispielhaft seien hier drei derzeit gängige Möglichkeiten für Sniffer beschrieben:

- Für den frei erhältlichen, universellen Netzwerk-Protokollanalyzer *Wireshark* [Vgl. WI] gibt es ein Plug-In, mit dem sich IEEE 802.15.4-Frames auswerten lassen[99]. Für Atmel Z-Link-Knoten wird vom *µracoli-Projekt* [Vgl. Wac09] eine

[96] Route Under MAC.
[97] Allgemeine Beobachtung der Netzwerkverkehrs, Fehlerdiagnose.
[98] Unterstützen Aufzeichnung und Auswertung.
[99] Entweder zur Laufzeit oder aus gespeicherten Mitschnitten.

Firmware bereitgestellt, die alle empfangenen Frames über die serielle Verbindung an einen angeschlossenen PC weiterleitet, wo sie zur Laufzeit über ein Python-Script in das benötigte Format konvertiert und per Pipe an Wireshark übergeben werden können.

- Der *Daintree Sensor Network Analyzer (SNA)* [Vgl. DN09] ist ein kommerzielles Produkt, welches Knoten verschiedener Anbieter unterstützt. Erforderliche Knotenfirmware befindet sich im Lieferumfang, wobei die Schnittstelle nicht offengelegt ist. Der Funktionsumfang geht über den von Wireshark hinaus, da nicht nur Pakete zerlegt und interpretiert werden können, sondern auch eine Visualisierung des Netzwerkverkehrs und der Netztopologien möglich ist und neben IEEE 802.15.4/ZigBee eine Reihe weiterer Netzwerkprotokolle unterstützt werden[100].
- *ZENA* [Vgl. ZEN09] ist ein mit dem Daintree SNA vergleichbares Analysewerkzeug, welches speziell für MicroChip-Plattformen entwickelt wurde. Neben IEEE 802.15.4 können ZigBee- und MiWi-Pakete dekodiert werden, eine grafische Aufbereitung der Netzwerktopologie nebst Anzeige des Paketflusses ist ebenso möglich.

[100] RF4CE, 6LoWPAN, JenNet, SimpliciTI und Synkro.

5 Verwandte Arbeiten

Bereits im Kapitel 2.7 wurde der Einsatz von IEEE 802.15.4 unter verschiedenen Betriebsbedingungen und Störungen diskutiert. Die dazu durchgeführten Untersuchungen betrafen den Test der Knotenfunktionen auf niedrigen Netzwerkschichten[101]. Sie bilden die erste Kategorie verwandter Arbeiten.

In Theorie und Simulation zeigen Funknetze bedingt durch Idealisierung bei der Modellbildung kaum Schwachstellen. Um jedoch das Verhalten kollaborierender Knoten in selbstorganisierenden Netzen zu erforschen, sind komplexe Testsysteme erforderlich. Sie bieten dazu Werkzeugunterstützung für den Entwurf und die Feinabstimmung von Netzen und Anwendungen sowie die Durchführung von Experimenten mit Knoten, die Anwendungen unter realen Bedingungen ausführen.

5.1 Vorhandene Testplattformen

Dass einerseits Hersteller von Funkmodulen eigene Testumgebungen betreiben, kann nur vermutet werden[102], andererseits gibt es im universitären Umfeld dazu zahlreiche Forschungsprojekte. Aus Platzgründen kann auch hier keine detaillierte Gegenüberstellung vorgenommen werden, so dass an dieser Stelle lediglich die bekanntesten Plattformen aufgelistet werden:

- Das *Senslab - Very large scale open wireless sensor network testbed* [Vgl. Bou09] ist ein französisches Kooperationsprojekt zwischen Universitäten und Industrieunternehmen, in dem 1024 fest installierte und mobile Knoten verwendet werden, die sich über vier Standorte erstrecken.
- Beim *BigNet - Sensor Network Testbed* [Vgl. Pal09] werden unter der Leitung der Universität Melbourne Knoten verschiedener Hersteller eingesetzt, die ebenfalls auf mehrere Standorte verteilt vorliegen.
- Das *Harvard Mote Lab* [Vgl. Mot09] besteht aus 190 TI[103]-basierten Knoten, die in einem Fakultätsgebäude der Universität Harvard platziert wurden.
- Mit *TKN Wireless Indoor Sensor network Testbed (TWIST)* [Vgl. Twi08] wird ein Testsystem der TU Berlin bezeichnet, das - wie auch in Harvard - in einem Bürogebäude untergebracht wurde und so eine Fläche von $1500m^2$ abdeckt. Eingesetzt werden je 102 Knoten von zwei verschiedenen Herstellern.
- Das an der FU Berlin betriebene *Scatterweb* [Vgl. Sca08] ist ein heterogenes, verteiltes Sensornetzwerk, das neben Lehre und Forschung auch für industri-

[101] Paketfehlerrate, Durchsatz.
[102] Dies ist jedoch nicht durch Quellen belegbar.
[103] Texas Instruments MSP430.

elle Projekte zur Verfügung steht. Dazu wurde es ausführlich dokumentiert, zur Unterstützung der Entwicklung eigener Knotenapplikationen existieren eine API zur Hardware-Abstraktion und verschiedene Tools.

- Unter dem Namen *SeNeTs* [Vgl. SeN09] werden alle an der Universität Rostock durchgeführten Forschungsprojekte zu drahtlosen Sensornetzen gebündelt. Eine grafische Oberfläche zur Konfiguration, Steuerung und Beobachtung des Netzes steht mit *EnviSense* bereit.

Der Funktionsumfang dieser Testumgebungen zeichnet sich in der Verallgemeinerung dadurch aus, dass:

- eine große Menge (ggf. heterogener) Knoten unterstützt wird,
- das Netz partitioniert werden kann und sich so unterschiedliche Anwendungen parallel ausführen lassen,
- Test automatisiert ablaufen[104], aufgezeichnet werden und auswertbar sind,
- sich einzelne Netzknoten beobachten lassen[105],
- der Aufbau der Netztopologie nachvollzogen werden kann,
- wahlweise Unterstützung für Simulation und Ausführung auf echten Knoten angeboten wird,
- sich gezielte Störungen provozieren lassen oder das Netz - um gleich unter realen Bedingungen zu arbeiten - in der Öffentlichkeit platziert wurde[106],
- die Netze vorrangig für Forschungsprojekte verwendet werden und der Industrie nur teilweise zur Verfügung stehen.

Ihre Systemarchitektur orientiert sich an einer Trennung der Kommunikationswege: Um die Kommunikation zwischen den Knoten per Funk nicht zu beeinflussen, basiert die Steuerungs- und Überwachungschnittstelle oft auf einer zusätzlichen Verkabelung. Die Hierarchisierung von Knoten ist nicht unüblich. Testfälle werden durch einen Server verwaltet, zur komfortablen Bedienung werden Oberflächen eingesetzt.

Aus all diesen Projekten wurden Anregungen für die eigene Testplattform gewonnen, welche in den nächsten Abschnitten vorgestellt wird.

[104] Inklusive automatischer Verbreitung der Knotenfirmware.
[105] Bspw. Energieverbrauch, Funkaktivität oder Status der Knotenapplikation.
[106] Bspw. innerhalb von Verwaltungsgebäuden oder in Innenstädten [Vgl. Cit08].

Teil III

Umsetzung

6 Vorgehensweise

Die Bearbeitung der Aufgabe begann mit einer Literaturrecherche. Dabei stellte sich heraus, dass es zwar viele Bücher über Funknetze und -technologien gibt, davon aber nur wenige auf den Themenkomplex IEEE 802.15.4 und ZigBee detailliert eingehen. Der Schwerpunkt der Recherche lag daher auf Online-Quellen (Fachartikel, wissenschaftliche Publikationen, vorhandene Testsysteme). Parallel dazu wurden der IEEE 802.15.4-Standard und die ZigBee-Spezifikation beschafft und studiert. Die Ergebnisse dessen sind im Grundlagenabschnitt zusammengefasst.

Im weiteren Verlauf folgte die Einarbeitung in den Atmel Protokollstack. Für die praktische Vertiefung wurden die mitgelieferten Beispiele auf Funkknoten ausprobiert und um Debugausgaben ergänzt. Zur Aufzeichnung der Knotenkommunikation wurde ein einfaches Serverprogramm entwickelt.

Mit der an dieser Stelle gewonnenen Erfahrung begann die Analyse der Anforderungen an das zu entwickelnde Testsystem. Da es sich nur durch Zerlegung in Teilprobleme beherrschen lässt, schloss sich im darauf folgenden Entwurf die Suche nach einer geeigneten Aufteilung der Funktionalität auf Komponenten an. Während der Entwicklung von Komponentenprototypen ergaben sich zusätzliche Anforderungen, die weitere Entwurfszyklen bedingten und den Entwurf schrittweise verfeinerten. Der finale Stand der Anforderungen und die daraus resultierende Systemarchitektur werden in den nächsten Kapiteln vorgestellt.

7 Anforderungsanalyse

Obwohl es bereits eine Reihe von Testplattformen gibt (Vgl. Kapitel 5), plant dresden elektronik den Aufbau einer eigenen. Motiviert wird dies dadurch, dass die Firma vorrangig mit Atmel-Produkten arbeitet, es derzeit aber keine darauf basierende Testplattform gibt. Andererseits wäre die Nutzung einer fremden Testumgebung mit Kosten verbunden, nur nach Abstimmung mit dem Betreiber möglich und ein physischer Zugriff oder gar Umbauten nur schwer vorstellbar. Nicht zuletzt setzt der Einsatz von Funkmodulen und -protokollen erfahrene Mitarbeiter voraus, deren Wissen sich durch die Eigenentwicklung einer Testplattform weiter anreichern lässt.

In diesem Kapitel werden die Anforderungen an die zu entwickelnde Testumgebung zusammengetragen. Als Basis dafür diente die vorgegebene Aufgabenstellung, zur Präzisierung wurden Rücksprachen mit dem Betreuer gehalten und denkbare Testfälle entworfen. Da das Projekt Teil einer Kooperation mit anderen Firmen und Forschungseinrichtungen ist, die Aufgabenteilung jedoch noch aussteht, stellen die genannten Anforderungen nur eine Momentanaufnahme dar. Primäres Ziel der Arbeit ist die Konzipierung und Spezifikation einer Testumgebung sowie eine schwerpunktorientierte Implementierung. Der dabei entstehende Prototyp soll als Diskussionsgrundlage dienen und wird die Basis für das spätere Testsystem bilden.

7.1 Funktionale Anforderungen

Pflichtkriterien:

- Abfrage unterstützter Eigenschaften eines Knotens
- Änderung der Betriebsparameter einzelner Knoten („Commissioning“)
- Beobachtung von Knoten zur Laufzeit, Abfrage aktueller Knotenparameter
- Beeinflussung der Knoten zur Laufzeit (LogLevel, (De-)Aktivierung)
- automatische Abarbeitung von Testfällen, Wiederholbarkeit der Ausführung
- parallele Ausführbarkeit von Testfällen
- Aufzeichnung der während der Abarbeitung des Testfalles anfallenden Daten
- Visualisierung unter Einsatz von Filtern
- Auswertung unter Erkennung von Fehlersituationen
- Wiedergabe bereits ausgeführter Testfälle („Replay“)

Wunschkriterien:

- Schedulingmechanismus für Testfälle
- Ermittlung der Verbindungsqualität
- Abfragbarkeit der Knotenbeziehungen
- Kompression redundanter Daten während der Persistierung

Abgrenzungskriterien:

- keine Softwarevalidierung der Knotenfirmware
- keine Simulation
- prototypische Implementierung ausgewählter Schwerpunkte
- keine Benutzerverwaltung

7.2 Nichtfunktionale Anforderungen

Betrieb und Umgebungsbedingungen:

- Einsatz von dresden elektronik-Funkmodulen (RCB) nach IEEE 802.15.4
- Funkmodule werden an einen Steuerrechner (PC) angebunden

Umsetzung:

- Bereitstellung einer Testfirmware für Funkknoten basierend auf der Atmel IEEE 802.15.4 Protokollstack-Referenzimplementierung
- Verteiltes System - Steuerrechner und Benutzerschnittstelle voneinander getrennt

Skalierbarkeit und Flexibilität:

- Unterstützung mehrerer Knoten
- Betriebssystem- und Plattformneutrale Umsetzung
- erweiterbare Schnittstellen zum Einsatz
  - von auf IEEE 802.15.4 aufbauenden Protokollen
  - Einsatz von Nicht-Atmel-Modulen
  - Unterstützung Ethernet-basierter Knotenanbindung

Zuverlässigkeit, Korrektheit und Fehlertoleranz:

- Integrität übertragener Daten
- synchronisierte Uhren

7.3 Denkbare Testszenarien

An dieser Stelle folgt eine Auswahl *möglicher* Testszenarien. Diese haben nicht den Charakter von Abnahmekriterien, sondern geben nur wieder, was mit dem System testbar sein könnte. Teilweise sind sie widersprüchlich und erfordern unterschiedliche Umsetzungen.

Vernetzung der Knoten:

- Einfluss unterschiedlicher Einschaltreihenfolgen der Knoten
- simultaner Start vieler Knoten
- Ermittlung der Dauer, bis alle Knoten mit Netz assoziiert sind

Leistungsparameter:

- Durchsatz (Pakete/Datenrate)
- maximale Reichweite ohne Dämpfung
- maximale Anzahl Hops zwischen Knoten
- Batterielebensdauer in Abhängigkeit von Kommunikationshäufigkeit und Länge der Schlaf-/Wachphasen
- Verhalten des Netzes bei hoher Auslastung, maximale Anzahl zeitgleich möglicher Übertragungen
- Übertragungswahrscheinlichkeit bei nichtbestätigter Übertragung

Übertragungsprobleme:

- Störsender
- Hidden-Node-Problem
- Kanalkapazität erschöpft
- Ausfall eines Knotens (Zunahme der Dämpfung, komplette Deaktivierung)
- Selbstheilung durch alternatives Routing (generelle Möglichkeit, zur Veränderung der Routingtabelle benötigte Zeit)

Verschiedenes:

- überlagerte Netze
- Interoperabilität von Knoten unterschiedlicher Hersteller
- Test verteilter Anwendungen

8 Systemarchitektur

8.1 Überblick

Im Testbed wird eine Menge von Funkmodulen verwendet, die das *Knotennetz* bilden. Zentrale Komponente ist jedoch der *Testserver*, der das Knotennetz kapselt und zur Steuerung der Experimente eingesetzt wird. Die Benutzerschnittstelle der Testumgebung wird durch *Clients* realisiert, zur Datenablage wird eine *Persistierungskomponente* verwendet. Daraus ergibt sich folgender Vorschlag einer Systemarchitektur:

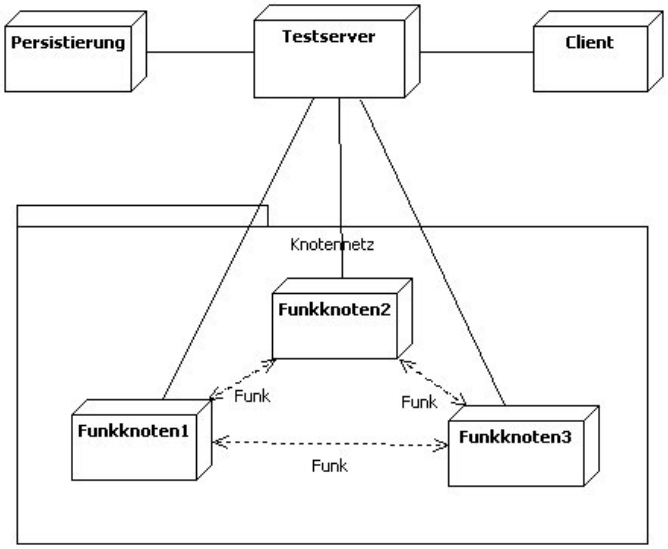


Abbildung 8.1: Systemarchitektur des Testbed.

8.2 Komponenten und deren Funktionsumfang

Der *Testserver* wird durch einen handelsüblichen PC realisiert, auf dem eine spezielle Testsoftware läuft. Seine Aufgaben umfassen:

- Verwaltung des Knotennetzes:
  - Hinzufügen/Entfernen von Knoten,
  - exklusive Zuweisung von Knoten an Testfälle,
- Verwaltung und Durchführung von Testfällen:
  - Bereitstellung von Knotenapplikationen,
  - Steuerung der Knoten während der Testfallausführung,
  - Protokollierung der Testfalldurchführung.

Um Applikationen auf die Funkmodule laden zu können, sind sie direkt mit dem Testserver verkabelt. Während der Experimente kommunizieren die Funkknoten untereinander jedoch über die Luftschnittstelle. Um den dabei genutzten Kanal nicht zusätzlich zu belegen, erfolgen Steuerung und Überwachung der Funkknoten ebenfalls kabelgebunden.

Da das Testbed räumlich getrennt von seinen Bedienern betrieben können werden soll, sind *Clients* vorgesehen. Diese werden über einen Socket angebunden und ermöglichen folgende Funktionen:

- Steuerung des Testservers (beenden, neu starten),
- Steuerung des Knotennetzes (stoppen, (re)initialisieren),
- Abfrage des Knotennetzes (verfügbare Knoten, unterstützte Parameter, aktuelle Parameter, derzeitige Verwendung in Testfällen),
- gezielte Beeinflussung einzelner Netzknoten, auch ohne zugehörigen Testfall
- Übermitteln, Validieren, Starten, Schedulen und Unterbrechen von Testfällen,
- Abfrage existierender Testfälle und deren Status (wartend, laufend, beendet),
- Beobachten momentan ausgeführter Testfälle (Status einzelner Knoten, Netztopologie),
- Auswertung von Testfällen, nochmalige Wiedergabe,
- Anzeige der vom Server gemeldeten Ereignisse, bspw. Veränderungen des Knotennetzes (entfernte/hinzugekommene Knoten), Statuswechsel von Testfällen, geplanter Halt des Servers,
- (optional: Sicherheitsfunktionen, bspw. Übertragung von Schlüsseln).

Die *Persistierungskomponente* ermöglicht die dauerhafte Ablage von:

- Testfallbeschreibungen,
- Knotenfirmware,
- während der Ausführung von Testfällen angefallener Daten.

Alle genannten Komponenten werden in den folgenden Kapitel detaillierter beschrieben, wobei vom Allgemeinen zum Speziellen vorgegangen wird. Im nächsten Kapitel werden zunächst die Schnittstellen zwischen den Komponenten beschrieben, da dies die Basis der ihrer Interaktion darstellt. Weil es zyklische Abhängigkeiten gibt, war es in dieser Arbeit nicht möglich, ohne Vorwärtsreferenzen zu arbeiten. An den betreffenden Stellen wird versucht, das Problem für das Verständnis ausreichend zu erläutern und nur die Stelle angegeben, an der es wieder aufgegriffen wird.

9 Kommunikationsprotokolle und Befehlsformate

9.1 Kommunikationsprotokoll Server-Netzknoten

Zur Kommunikation zwischen Testserver und Netzknoten wird ein Binärprotokoll eingesetzt, wobei zwei Fälle zu unterscheiden sind:

1. Netzknoten sind passiv, d.h. sie warten auf eingehende Befehle, die sie beantworten[107]. Das Protokoll arbeitet hier nach dem *stop&wait*-Prinzip[108].
2. Ereignisse werden von den Knoten selbständig gemeldet[109], beispielsweise ein erfolgter Reset.

Zur Übertragung der Nachrichten wird das folgende Paketformat verwendet:

Feld	TYPE	MAC	TS*	CMD	ATTR	LEN	DATA	CRC*	TERM
Länge/Bytes	1	8	8	1	1	1	0-255	1	1

Abbildung 9.1: Paketformat zur Kommunikation zwischen Knoten und Server.

Die mit * gekennzeichneten Felder sind optional[110]. Obwohl Pakete eine variable Länge aufweisen, müssen stets alle Felder vorhanden sein[111]. Dies vereinfacht die Nachrichtenerstellung und -decodierung. Der Aufbau des Paketes ist hierarchisch, d.h. von links beginnend wird es nach rechts immer spezieller. Es folgt eine kurze Beschreibung der verwendeten Felder:

- *TYPE* definiert die Nachrichtenart und dadurch implizit ihre Laufrichtung.
- *MAC* beinhaltet die MAC-Adresse eines Netzknotens[112]. Knoten akzeptieren nur Nachrichten, die an die eigene Adresse oder die „Broadcast-Adresse“ `0xFF-FF-FF-FF-FF-FF-FF-FF` gerichtet sind.
- Im *TS*-Feld kann *optional* ein Zeitstempel eingetragen werden, um den Absendezeitpunkt der Nachricht zu speichern[113].
- Unter *CMD* wird ein Kommando abgelegt.
- *ATTR* spezialisiert das vorige Kommando (Attribut oder Subkommando).

[107] Die Anforderungen bestehen immer nur aus einer Nachricht, zugehörige Antworten bestehen nur in Ausnahmefällen aus mehreren Nachrichten.
[108] Erst nachdem eine Anforderung durch den Knoten beantwortet wurde, kann die nächste gesendet werden. Aus der Antwort geht hervor, ob der Befehl erfolgreich ausgeführt werden konnte oder nicht. Die Wartezeit auf die Antwort wird mit einem Timeout abgesichert.
[109] Pollen wäre Resourcenverschwendung.
[110] (De-)Aktivierung bei der Erstellung der Knotenfirmware bzw. Schalter im Server. Verwendung jedoch stets einheitlich für alle Knoten. Im aktuellen Stand werden beide Felder verwendet.
[111] Unbenötigte Felder können speziell gekennzeichnet werden.
[112] Je nach Nachrichtentyp ist dies entweder der Empfänger oder der Absender der Nachricht.
[113] Dies ist wichtig, da die Nachrichtenübertragung einer unbestimmten Latenz unterliegt. Erst so lassen sich Abstände zwischen einzelnen Ereignissen ermitteln.

- *ACK* beinhaltet einen Statuscode[114].
- *LEN* gibt die Länge des folgenden Datenfelds an[115].
- *DATA* ist nur Bestandteil der Nachricht, wenn der Wert des *LEN*-Felds > 0 ist. Darin werden einzelne Befehlsparameter oder komplette Datenstrukturen übertragen, deren Semantik sich implizit aus den vorigen Feldern ableitet.
- Das *CRC*-Feld beinhaltet eine Prüfsumme, die Byte-weise per XOR über alle Nachrichtenfelder berechnet wird[116]. Das CRC-Feld ist *optional*.
- *TERM* kennzeichnet das Paketende[117].

Die Felder *TS* und *CRC* werden bedarfsweise erst zum Zeitpunkt des Nachrichtenversands generiert. Für alle anderen Felder sind in einer Schnittstellenbeschreibung[118] mögliche Belegungen vorgegeben (Vgl. Listing 9.1). Dadurch werden einerseits ungültige Feldbelegungen vermieden, andererseits kann über symbolische Namen darauf zugegriffen werden[119].

```
1  ...
2  #define TCL_MSGTYPE_REQUEST                  (0x00)
3  #define TCL_MSGTYPE_RESPONSE                 (0x01)
4  #define TCL_MSGTYPE_WPAN_INVOCATION          (0x02)
5  #define TCL_MSGTYPE_WPAN_CALLBACK            (0x03)
6  #define TCL_MSGTYPE_STATUS                   (0x04)
7  #define TCL_MSGTYPE_FRAME_CALLBACK           (0x05)
8  ...
9  #define TCL_ACKSTATUS_NACK                   (0x00)
10 #define TCL_ACKSTATUS_ACK                    (0x01)
11 #define TCL_ACKSTATUS_DONTCARE               (0xFE)
12 #define TCL_ACKSTATUS_ERROR                  (0xFF)
13 ...
```

Listing 9.1: Beschreibung gültiger Belegungen für Nachrichtentyp- und Statusfeld im Paketformat der Server-Knoten-Kommunikation (Vgl. `stack/TCL/Inc/commands.h`).

Pakete werden als Bytestrom übertragen, bei dessen Zerlegung in einzelne Nachrichten es vorkommen kann, dass das *Endesymbol* inmitten der Nachricht auftaucht[120]. Daher wird anhand des Längenfeldes die exakte Position des Nachrichtenendes berechnet. Alternativ könnte jedes außerplanmäßige Vorkommen des Endesymbols durch ein zusätzliches festgelegtes Zeichen maskiert werden, was jedoch einen erhöhten Aufwand bei der Zerlegung der Nachricht erfordert.

[114] Anhand dieses Feldes lassen sich in einer Menge von Nachrichten diejenigen herausfiltern, die möglicherweise zu Fehlersituationen geführt haben.
[115] Durch die Beschränkung auf ein Byte sind maximal 255 Byte lange Datenfelder möglich, wodurch selbst komplette MAC-Frames in ein Datenfeld passen.
[116] Damit lassen sich Übertragungsfehler erkennen, aber nicht beheben.
[117] Als Endesymbol wird derzeit `0x0A` verwendet.
[118] Header-Datei bzw. statische Klasse.
[119] Dadurch, dass sich trotz Gültigkeit einzelner Feldbelegungen daraus ungültige Kombinationen bilden lassen, findet zur Laufzeit eine Validierung der verwendeten Feldkombinationen statt.
[120] Dies ist unvermeidbar, da der in den Nachrichtenfeldern genutzte Zeichenvorrat alle möglichen Werte aus `[0x00;0xFF]` einnehmen kann.

Nachrichten werden eingesetzt zur

- Abfrage der von Knoten unterstützten Eigenschaften,
- Abfrage und Änderung von Knotenparametern,
- Anforderung von Diensten des Protokollstack,
- Meldung von Ereignissen während der Ausführung der Knotenapplikation.

Für *an Knoten gesendete Befehle* sind nur die Nachrichtentypen `TCL_MSGTYPE_REQUEST` bzw. `TCL_MSGTYPE_WPAN_INVOCATION` zulässig, das ACK-Feld kann nur mit dem Wert `TCL_ACKSTATUS_DONTCARE` belegt werden. Das *zugehörige Antwortpaket* hat bei *Erfolg* dementsprechend als Nachrichtentyp `TCL_MSGTYPE_RESPONSE` bzw. `TCL_MSGTYPE_WPAN_CALLBACK` gesetzt, das ACK-Feld enthält `TCL_ACKSTATUS_ACK`. Im *Fehlerfall* sind im Antwortpaket Nachrichtentyp, Kommando und Attribut mit denen der Anfrage identisch, das ACK-Feld indiziert den Fehler mittels `TCL_ACKSTATUS_ERROR` und im Datenfeld ist die Fehlerursache[121] vermerkt. *Nachrichten zur Ereignismeldung* enthalten im ACK-Feld entweder `TCL_ACKSTATUS_DONTCARE`, wenn das Ereignis nicht mit einem Status verbunden ist, andernfalls `TCL_ACKSTATUS_ACK` bzw. `TCL_ACKSTATUS_NACK` bei Erfolg bzw. Fehlern[122].

9.2 Kommunikationsprotokoll Server-Client

Die Clients-Server-Kommunikation wurde nicht implementiert. An dieser Stelle wird lediglich eine *mögliche Schnittstelle* in Verbindung mit einer Variantendiskussion vorgestellt. Zur Nachrichtenübertragung wird ein Paketformat vergleichbar mit dem der Kommunikation zwischen Server und Knoten vorgeschlagen. Die Felder für MAC-Adresse und Zeitstempel entfallen, eine Prüfsumme ebenfalls, da durch die auf TCP/IP-Sockets basierende Kommunikation bereits auf IP-Schicht die Integrität übertragener Daten sichergestellt wird. Um große Nutzdaten wie Testfallbeschreibungen übertragen zu können, müssen diese entweder fragmengtiert oder das Längenfeld auf mehrere Bytes vergrößert werden, so dass sich das in Abbildung 9.2 dargestellte Paketformat ergibt.

Feld	TYPE	CMD	ATTR	LEN	DATA	TERM
Länge/Bytes	1	1	1	$\geq$1	$\geq$0	1

Abbildung 9.2: Paketformat zur Kommunikation zwischen Server und Client.

[121] Bspw. nicht unterstützter Befehl oder Länge des Datenfelds nicht zum Befehl passend.

[122] Das Datenfeld enthält dann eine komplette Datenstruktur innerhalb deren die Fehlerursache eingetragen ist. Zur Abgrenzung gegenüber Datenfeldern die nur den Fehlercode enthalten wurde `TCL_ACKSTATUS_ERROR` hier nicht verwendet.

Sobald zur Übertragung einer Nachricht mehrere Pakete eingesetzt werden oder Nachrichtenfolgen zum Erreichen eines gewünschten Verhaltens notwendig sind, ist eine zustandsbehaftete Kommunikation erforderlich. Da das System nur firmenintern verwendet wird, sind Angriffe unwahrscheinlich. Es werden überdies keine schutzbedürftigen Daten ausgetauscht und ohne Kenntnis des Paketformats lassen sich keinerlei Rückschlüsse auf dessen Semantik ziehen. Soll trotzdem eine gesicherte Datenübertragung erfolgen, wird empfohlen auf bereits dafür existierende Möglichkeiten[123] zurückzugreifen.

Falls eine direkte Ansteuerung von Funkknoten erforderlich ist, könnte das bereits zwischen Server und Knoten eingesetzte Paketformat eingesetzt und als Nutzlast im neuen Paketformat gekapselt werden. Problematisch ist in diesem Zusammenhang die semantische Auswertung der Antworten des Knotens auf Steuerbefehle. Sie kann entweder bereits im Server erfolgen, oder wiederum erst in den Clients. Im Sinne einer funktionalen Trennung zwischen Server und Client sowie der Reduzierung der Komplexität der Clients (Thin-Client) wird vorgeschlagen, die Interpretation im Server vorzunehmen und in einem Zwischenformat an den Client zu übertragen.

Eine Trennung in aktive und passive Rollen ist auch hier nicht möglich: Einerseits sendet der Client Steuerbefehle an den Server, andererseits meldet der Server selbständig Ereignisse[124]. Für das Kommunikationsprotokoll wird daher eine Vorgehensweise analog Knoten-Server vorgeschlagen: Anforderungen des Clients an den Server werden vom Server bestätigt, Ereignismeldungen des Server an den Client jedoch nicht. Auch hier ist die Nutzung von Timeouts in Verbindung mit einer begrenzten Anzahl an Übertragswiederholungen angebracht, um unendliches Warten zu verhindern. Die Kommunikation könnte auch auf getrennte Verbindungen verteilt werden, wobei eine davon zur Übermittlung von Steuerdaten und die andere lediglich für die Ereignisbenachrichtigung verwendet wird.

Völlig andere Ansätze stellen die Kommunikation mittels Objekttransfer oder gleich der Einsatz einer objektorientierten Middleware dar. [125].

9.3 Kommunikationsprotokoll Server-Datenbank

Die Kommunikation zwischen Server und Datenbank erfolgt über SQL-Statements, die keiner weiteren Erläuterung bedürfen.

[123] Aufbau eines VPN, Einsatz von IPSec.
[124] Bspw. Beendigung von Testfällen.
[125] Bspw. CORBA (Common Object Request Broker Architecture).

10 Beschreibung automatisierbarer Tests

Das Testsystem soll in der Lage sein, Tests automatisch abzuarbeiten. Dazu müssen zunächst Testfälle vorbereitet werden, deren Bestandteile im nächsten Abschnitt erläutert werden. Weiterhin ist ein geeignetes Darstellungsformat erforderlich, um Testfälle einerseits speichern und andererseits vor der Ausführung wieder laden zu können. Dies wird im darauf folgenden Kapitel beschrieben. Die Abarbeitung eines Testfalles wird später in einem gesonderten Kapitel behandelt.

10.1 Elemente einer Testfallbeschreibung

Die zur Ausführung eines Testfalles benötigten Parameter umfassen mindestens:

- *Zu verwendende Knoten*. Es wird davon ausgegangen, dass im Testbed eine Menge von Knoten verfügbar ist, von denen nicht alle simultan verwendet werden müssen. Die Auswahl der Knoten kann dabei anhand ihrer MAC-Adresse erfolgen, da sie einen eindeutigen Schlüssel darstellt[126]. Pro Knoten können dabei spezifische Initialisierungsparameter angegeben werden, beispielsweise dessen Rolle[127].
- *Netzwerkparameter*, die von allen Knoten gleichermaßen benutzt werden:
  - PAN-ID, Channel und Channel Page[128],
  - Beacon- und Superframe-Order,
  - Zu bildende Netztopologie,
  - Bei Einsatz von Sicherheitsfunktionen: zu verwendende Schlüssel.
- *Die Beschreibung der Testfalldurchführung*:
  - Testfalldauer,
  - Aktivitätsintervalle von Knoten[129],
  - während des Testfalls zu simulierende Störungen mit Angabe von Zeitintervallen,
  - regelmäßig auszuführende Aktionen[130] mit Angabe von Startzeiten und Wiederholungen,
  - Aktionen, die bei Eintreten bestimmter Ereignisse auszuführen sind.

  Sämtliche Zeitangaben erfolgen dabei relativ zum Start der Testfallausführung, als kleinste Auflösung können Millisekunden dienen.

[126] Sie ist bereits durch die Knotenhardware vorgegeben und entweder im EEPROM oder auf einen speziellen ID-Chip (Vgl. ZigBit Modul) abgelegt.
[127] Möglich wären u.a. Coordinator, End-Device, Router oder Sensor, Aktor. Die Angabe ist abhängig von der im konkreten Testfall verwendeten Applikation, da IEEE 802.15.4 keine Rollen vorsieht.
[128] Eine zukünftige Erweiterung mit dem sich Kanäle gruppieren lassen.
[129] Knoten können zeitverzögert zu- oder abgeschaltet werden.
[130] Bspw. Datenübertragungen.

10.2 Testfall-Steuerdateien

Zur Abbildung aller Testfallparameter werden im Testbed sogenannte *Testfall-Steuerdateien* verwendet. Als Dateiformat wurde XML[131] gewählt, da es eine strukturierte Ablage von Daten erlaubt und sich als Datenaustauschformat, insbesondere in der Maschine-Maschine-Kommunikation etabliert hat. Gegenüber einem proprietären Dokumentenformat weist XML mehrere Vorteile auf:

- Die Verwendung von Schemata ermöglicht die Definition individueller Sprachen, zugehöriger Sprachelemente und einschränkender Bedingungen. Anhand eines zugeordneten Schemas kann ein Dokument einfach auf Wohlgeformtheit und Gültigkeit seiner Elemente hin untersucht werden.
- Zur maschinellen Verarbeitung von XML-Dokumenten existieren bereits XML-Prozessoren (XML-Parser), die partiell auch validierend arbeiten können. Die Implementierung eigener Parser und Validierer entfällt damit. Der Zugriff auf die im Dokument abgelegten Daten ist abhängig vom Verarbeitungsmodell[132]. Spezielle Erweiterungen (Data Bindings) kapseln den Zugriff weiter, so dass direkt mit nativen Objekten gearbeitet werden kann.

Hinsichtlich der Definition der Eigenschaften von Knoten in XML-Dokumenten sind zwei grundlegend verschiedene Paradigmen möglich: Die *Verwendung von Knotenattributen* ist intuitiver und einfacher im Parser zu verarbeiten, Eigenschaften lassen sich jedoch nur textuell angeben und es ist keine Reihenfolge festlegbar. Durch *Verwendung von Unterknoten* werden Wiederholungen von Elementen bei beliebiger Verschachtelungstiefe erlaubt, Änderungen am Datenmodell sind einfacher übertragbar. In dieser Arbeit werden hauptsächlich Unterknoten verwendet.

Inhaltlich lehnt sich der Aufbau der Testfall-Steuerdateien an die Angaben im vorigen Kapitel an und soll an einem Auszug verdeutlicht werden:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tns:testcase
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:tns="http://www.dresden-elektronik.de/testcases"
5     xsi:schemaLocation="http://www.dresden-elektronik.de/testcases testcase.xsd">
6
7     <networkDefinition>
8      <networkParameters>
9        <beaconOrder>10</beaconOrder>
10       <superframeOrder>9</superframeOrder>
11       <channel>21</channel>
12       <panId>CAFE</panId>
13        ...
14     </networkParameters>
```

[131] Extensible Markup Language.

[132] Bspw. erlauben nach dem Document Objekt Model (DOM) arbeitende XML-Prozessoren einen wahlfreien Zugriff auf den Syntaxbaum. In der Simple API for XML (SAX)-Implementierung ist nur ein sequentieller Zugriff möglich.

```
15
16      <requiredNetworkMember>
17        <description>PAN-Coordinator</description>
18        <addr>00-04-25-FF-FF-17-33-73</addr>
19        <type>
20          <regularNode>
21            <role>coordinator</role>
22          </regularNode>
23        </type>
24      </requiredNetworkMember>
25
26      <requiredNetworkMember>
27         ...
28      </requiredNetworkMember>
29    </networkDefinition>
30
31    <testcaseDefinition>
32      <testcaseParameters>
33        <description>Ermittlung Dauer bis zur Assoziation aller Knoten</description>
34        <startOffset>5</startOffset>
35        <duration>30</duration>
36        <loglevel>full</loglevel>
37        <executionLaterAllowed>false</executionLaterAllowed>
38        <rescheduleOnServerShutdown>false</rescheduleOnServerShutdown>
39        <nodeAvailabilityCheck>
40          <retryCounter>3</retryCounter>
41          <waitDuration>2</waitDuration>
42        </nodeAvailabilityCheck>
43      </testcaseParameters>
44
45      <nodeStateChange>
46        <nodeAddress>00-04-25-FF-FF-17-33-73</nodeAddress>
47        <timeOffset>0</timeOffset>
48        <newState>on</newState>
49      </nodeStateChange>
50
51      <dataPacket id="1">
52        <description>request echo state</description>
53        <message>
54          <msg_t>00</msg_t>
55          <addr>00-04-25-FF-FF-17-33-73</addr>
56          <cmd>01</cmd>
57          <attr>A1</attr>
58          <ack>FE</ack>
59          <data></data>
60        </message>
61      </dataPacket>
62
63      <packetToSend id="1">
64        <timeOffset>5</timeOffset>
65        <iterationCount>2</iterationCount>
66        <iterationInterval>3</iterationInterval>
67      </packetToSend>
68
69    </testcaseDefinition>
70
71  </tns:testcase>
```

Listing 10.1: Auszug aus einer Testfall-Steuerdatei

Da inhaltlich zusammengehörend, werden Netzparameter und zu verwendende Netzknoten unter dem XML-Knoten `<networkDefinition>` erfasst. Der zweite XML-Knoten, `<testcaseDefinition>`, beschreibt die Testfalldurchführung. Im ersten Abschnitt werden darin Ausführungsbeginn und Dauer des Testfalles definiert sowie ein Detaillierungsgrad, mit dem dessen Ausführung zu protokollieren ist (Vgl. Zeilen 34-36). Es folgen zwei Scheduler-bezogener Parameter (Vgl. Zeilen 37,38). Sie spezifizieren, ob der Testfall generell zeitverzögert ausführbar ist und ob er nach einem Neustart des Testservers wieder in den Scheduler auf-

genommen werden kann. Den abschließenden Block bilden Parameter, die angeben, wie oft und in welchen Abständen vor der Ausführung eines Testfalles versucht werden soll, die Verfügbarkeit erforderlicher Knoten zu überprüfen (Vgl. Zeilen 39-42).

Der Knoten `<nodeStateChange>` legt die Ein- und Ausschaltzeitpunkte der Netzknoten fest (Vgl. Zeilen 45-49), die beiden letzten Abschnitte Datenpakete, die während der Ausführung des Testfalles in das Netz auszuliefern sind. Diese sind zunächst zu spezifizieren (Vgl. `<dataPacket>`, Zeilen 51-61), danach deren Sendezeitpunkt, Anzahl der Wiederholungen und zwischenliegende Abstände anzugegeben (Vgl. `<packetToSend>`, Zeilen 63-67).

Zur Schemadefinition wird in dieser Arbeit XML-Schema verwendet. Damit wurde ein eigener Namensraum (Vgl. Listing 10.2, Zeilen 4,5) definiert und die Dokumentenstruktur beschrieben. Zur Erhöhung der Übersichtlichkeit wurden Unterknoten des Wurzelknotens in anonyme innere Typen ausgelagert (Vgl. Zeilen 17-23) und zur Wertebereichseinschränkung eigene Datentypen eingeführt (Vgl. Zeilen 9-13). Die referenzielle Integrität zwischen definierten und zu sendenden Datenpaketen wird durch eine Fremdschlüsselbeziehung zwischen ihren IDs zugesichert (Vgl. Zeilen 51-58). Analog sind Datenpaketen nur Knoten adressierbar, die auch in der Testfallbeschreibung verwendet werden. Um sicherzustellen, dass Knoten nur einmal an und ausgeschaltet werden können, bilden Knotenadresse und Knotenzustand einen eindeutigen Schlüssel (Vgl. Zeilen 60-64).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema
3     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4     targetNamespace="http://www.dresden-elektronik.de/testcases"
5     xmlns:tns="http://www.dresden-elektronik.de/testcases"   >
6
7     ...
8
9     <xsd:simpleType name="macAddress">
10       <xsd:restriction base="xsd:string">
11          <xsd:pattern value="([0-9,a-f,A-F]{2}-){7}[0-9,a-f,A-F]{2}"></xsd:pattern>
12       </xsd:restriction>
13    </xsd:simpleType>
14
15   ...
16
17    <xsd:complexType name="requiredNetworkMemberType">
18     <xsd:all>
19       <xsd:element name="description" type="xsd:string" minOccurs="0" />
20       <xsd:element name="addr" type="tns:macAddress" />
21       ...
22     </xsd:all>
23   </xsd:complexType>
24
25    ...
26
27    <xsd:element name="testcase">
28       <xsd:complexType>
29          <xsd:sequence>
30             <xsd:element name="networkDefinition">
31                <xsd:complexType>
32                   <xsd:sequence>
```

```
33                      <xsd:element name="networkParameters" type="
                            tns:networkParametersType" />
34                      <xsd:element name="requiredNetworkMember" type="
                            tns:requiredNetworkMemberType" maxOccurs="unbounded"/>
35                   </xsd:sequence>
36                </xsd:complexType>
37             </xsd:element>
38             <xsd:element name="testcaseDefinition">
39                <xsd:complexType>
40                   <xsd:sequence>
41                      <xsd:element name="testcaseParameters" type="
                            tns:testcaseParametersType" />
42                      <xsd:element name="nodeStateChange" type="
                            tns:nodeStateChangeType" minOccurs="0" maxOccurs="unbounded"
                             />
43                      <xsd:element name="dataPacket" type="tns:dataPacketsType"
                            minOccurs="0" maxOccurs="unbounded" />
44                      <xsd:element name="packetToSend" type="tns:packetsToSendType"
                            minOccurs="0" maxOccurs="unbounded" />
45                   </xsd:sequence>
46                </xsd:complexType>
47             </xsd:element>
48          </xsd:sequence>
49       </xsd:complexType>
50
51       <xsd:key name="packetId">
52          <xsd:selector xpath="./testcaseDefinition/dataPacket" />
53          <xsd:field xpath="@id" />
54       </xsd:key>
55       <xsd:keyref name="keyIdref" refer="tns:packetId">
56          <xsd:selector xpath="./testcaseDefinition/packetToSend" />
57          <xsd:field xpath="@id" />
58       </xsd:keyref>
59
60       <xsd:unique name="nstUnique">
61       <xsd:selector xpath="./testcaseDefinition/nodeStateChange"/>
62       <xsd:field xpath="nodeAddress"/>
63       <xsd:field xpath="newState"/>
64     </xsd:unique>
65
66    </xsd:element>
67 </xsd:schema>
```

Listing 10.2: Auszug aus der Schemadefinition für Testfall-Steuerdateien

Die Beschreibungsmächtigkeit von XML-Schemata ist aber auch limitiert. Einseitige Abhängigkeiten lassen sich – wie am Beispiel der Fremdschlüsselbeziehung in der Steuerdatei – noch angeben. Existenzielle Abhängigkeiten zwischen einzelnen Elementen[133] oder ihren Attributen[134] lassen sich jedoch nicht definieren, sondern müssen bedarfsweise manuell validiert werden. Im Beispiel der Steuerdatei ist dies für die Knoten `<beaconOrder>` und `<superframeOrder>` wichtig: wenn eine Beacon-Order angegeben ist, muss auch eine Superframe-Order vorhanden sein. Zusätzlich muss *Beacon Order* $\leq$ *Superframe Order* gelten. Eine Beispiel einer Steuerdatei und das vollständige Schema sind im Anhang zu finden.

[133] Bspw. wenn Knoten A vorhanden ist, muss auch Knoten B vorhanden sein; wenn A fehlt, dann muss auch B fehlen.

[134] Attribut X von Knoten A $\leq$ Attribut Y des Knotens B.

11 Funkknoten

Die folgenden Betrachtungen beziehen sich sowohl auf Atmel-basierte Hardware[135] als auch auf von Atmel gelieferte Software. Dabei wurde nur der IEEE 802.15.4-Stack verwendet. Der ZigBee-Protokollstack wurde aus Zeitgründen, aber auch, weil er nicht quelloffen ist, nicht eingesetzt.

11.1 Anbindung an den Testserver

Die Anbindung der Funkmodule an den Testserver erfolgt derzeit per Sensor Terminal Board (STB) (Vgl. Abb. 11.1), auf dem ein FTDI245RL-Chip [Vgl. FTD09] als USB-FIFO arbeitet[136]. Über einen von FTDI bereitgestellten Treiber ist der Zugriff darauf per virtuellem COM-Port möglich[137]. Der Treiber implementiert einen bidirektionalen Gerätezugriff per USB-Bulktransfer in Blöcken von 64 Byte mit Datenraten bis zu 300 kB/s. FTDI empfiehlt stets mit der vollen Blockgröße zu schreiben [Vgl. FTD08], andernfalls wird nur noch Byte-weise gesendet, was die Performanz deutlich einbrechen lässt. Empfangene Daten werden entweder an den Host-PC übertragen, sobald der Eingangspuffer komplett gefüllt oder der Latenz-Timer abgelaufen ist.

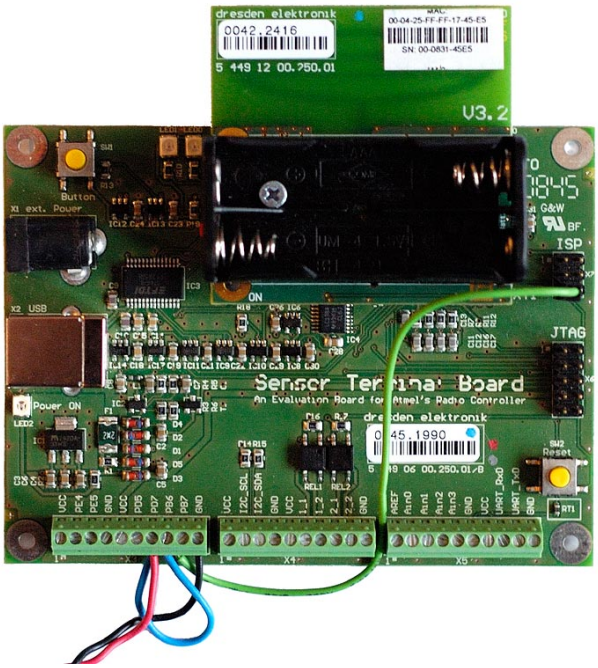


Abbildung 11.1: dresden elektronik Sensor-Terminal-Board.

[135] Eingesetzt wurden dresden elektronik RCBs.

[136] Dabei handelt es sich um ein Full-Speed-Gerät, welches 256 Byte Empfangs- und 128 Byte Sendepuffer besitzt.

[137] Treiber sind für Linux, Mac OS und Windows verfügbar. Die maximale Anzahl von COM-Ports ist betriebssystemspezifisch, unter Windows gibt es maximal 256.

Kritisch zu bewerten ist in diesem Zusammenhang die träge Übermittlung von Steuerbefehlen an den Knoten, da die dazu eingesetzten Datenpakete i.d.R. kürzer als 64 Byte sind. Umgekehrt werden vom Knoten verschickte Nachrichten bereits vorab mit einem Zeitstempel versehen, so dass die Standard-Latenz von 16 ms, nach der das entsprechende Datenpaket spätestens beim Betriebssystem des Servers vorliegt, nicht ins Gewicht fällt. Einen wesentlich größeren Einfluss auf Empfangsverzögerungen haben vielmehr die Dimensionierung der Zeitscheiben im Betriebssystem und die Systemlast, die indirekt bestimmt, wann der Serverprozess das Datenpaket letztendlich erhält.

Die verfügbare USB-Bandbreite ist selbst beim Betrieb eines Snifferknotens im 2.4 GHz-Band ausreichend dimensioniert. Bei einem ausgelasteten Übertragungskanal wird dort auf Ebene des Physical Layer eine Datenrate von 250 kb/s erreicht, was etwa einem Zehntel der verfügbaren Datenrate der USB-Verbindung entspricht. Jeder empfangene PHY-Frame beinhaltet dabei einen nur 6 Byte kleineren MAC-Frame, welcher als Nachricht gekapselt und über die serielle Verbindung ausgeben wird. Aufgrund des dabei eingesetzten Paketformats wird jede Nachricht 13-28 Byte länger[138] als der MAC Frame, wodurch sich der Protokoll-Overhead selbst bei kleinen Frames nicht so dramatisch auswirken sollte, dass die USB-Bandbreite nicht mehr ausreicht. Die maximale Datenrate wird außerdem nur theoretisch erreicht, da der Kanal je nach Betriebsart gar nicht dauerhaft belegt werden kann, sondern konkurrierend benutzt wird und Geräte vor dem Zugriff per CSMA/CA-Verfahren zusätzliche Wartezeiten einlegen.

Einen nur theoretisch vorhandenen „Flaschenhals“ stellt der Mikrocontroller dar, der bei 8 MHz Takt durch seine RISC-Architektur beinahe 8 MIPS erreicht, wenn sich die Befehlsoperanden in Registern befinden. Zugriffe auf RAM und I/O, zu denen Zugriffe auf den USB-FIFO zählen, benötigen zwar mehr als einen Takt pro Operation, jedoch sollte sich damit die verfügbare Bandbreite der USB-Anbindung übertreffen lassen, wenn der Mikrocontroller außer der Datenübertragung keine weiteren Aufgaben hat.

11.2 Modifikationen und Erweiterungen des Atmel-Stack

Der Atmel Protokollstack wurde bereits im Kapitel 4.2 vorgestellt. Das eingesetzte proprietäre Sensor-Terminal-Board wird von ihm jedoch nicht unterstützt, so dass zunächst ein eigener PAL dafür entwickelt wurde. Den Schwerpunkt der Anpassung bildet jedoch ein zusätzliches Modul, welches Testbed Control Layer (TCL) genannt wurde (Vgl. Abb. 11.2). Es ermöglicht die Kommunikation des

[138] Je nachdem, ob Zeitstempel, Prüfsumme und/oder ergänzende Frameinformationen wie Empfangsfeldstärke aktiviert sind.

Knotens mit dem Testserver über das in Kapitel 9.1 vorgestellte Protokoll und erweitert die Knotenfirmware um einen Zustandsautomaten. In Fortführung der Aufteilung der Stack-Module auf separate Verzeichnisse befindet sich der TCL im gleichnamigen Verzeichnis. Mit Ausnahme von internen Hilfsroutinen beginnen die Namen aller Funktionen mit dem Präfix „`tcl_`“. Sämtliche ab hier referenzierten Quellcode-Dateien sind auf der beiliegende CD enthalten.

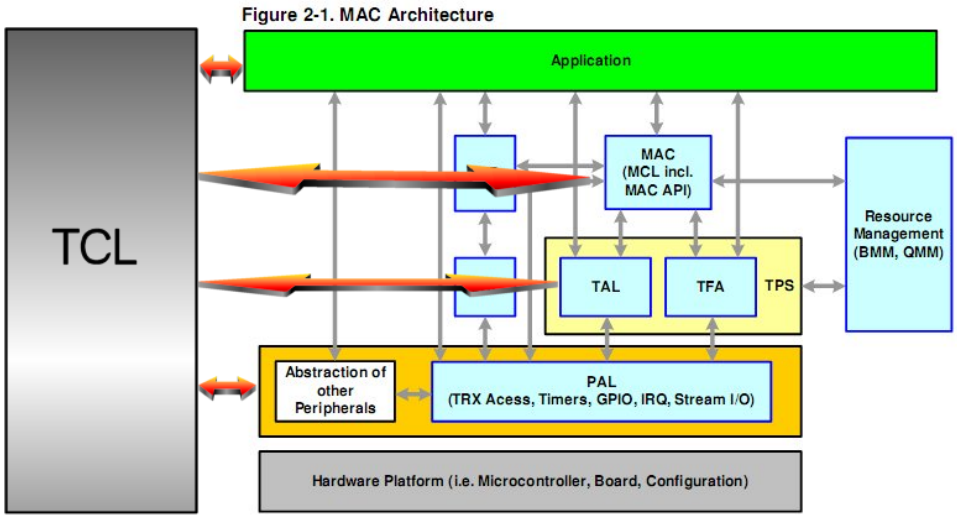


Abbildung 11.2: Stackerweiterung um Testbed Control Layer.

Während der Initialisierung des TCL (`tcl_init()`) wird die serielle Verbindung eingerichtet und ein Eingangspuffer angelegt. Es werden Datenstrukturen für Nachrichten erzeugt; um diese mit Zeitstempeln versehen zu können, wird ein Timer konfiguriert. Als Zeitgeber hierfür bietet sich der Transceiver an, da dieser einerseits auf dem RCB durch einen externen Quarz getaktet wird und sich andererseits durch Setzen eines Transceiver-Registers dessen 1MHz-Taktsignal auf ein Pin (CLKM) ausgeben lässt, welches mit dem Mikrocontroller verbunden ist. Der Timer arbeitet als 16-Bit-Zähler, bei Überlauf wird eine lokale Variable inkrementiert. Für den Fall, dass auf einer anderen Hardware-Plattform kein externes Taktsignal verfügbar ist, wurden Initialisierungs- und Interrupt-Service-Routinen für prozessorgetaktete Timer vorgesehen, welche durch einen Schalter während der Übersetzung einfach austauschbar sind.

Regelmäßig auszuführende Aktionen werden im Stack durch die Funktion `wpan_task()` realisiert. Darin integriert wurde der `tcl_task()`, welcher den Eingangspuffer der seriellen Verbindung ausliest und dabei versucht den Datenstrom wieder in Nachrichten zu zerlegen. Vereinfacht zusammengefasst wird anschließend jedes erkannte Paket auf passenden Typ und gültige Prüfsumme hin untersucht und entsprechend des Befehls eine weitere Handlerroutine aufgerufen (Vgl. Abb. 11.3).

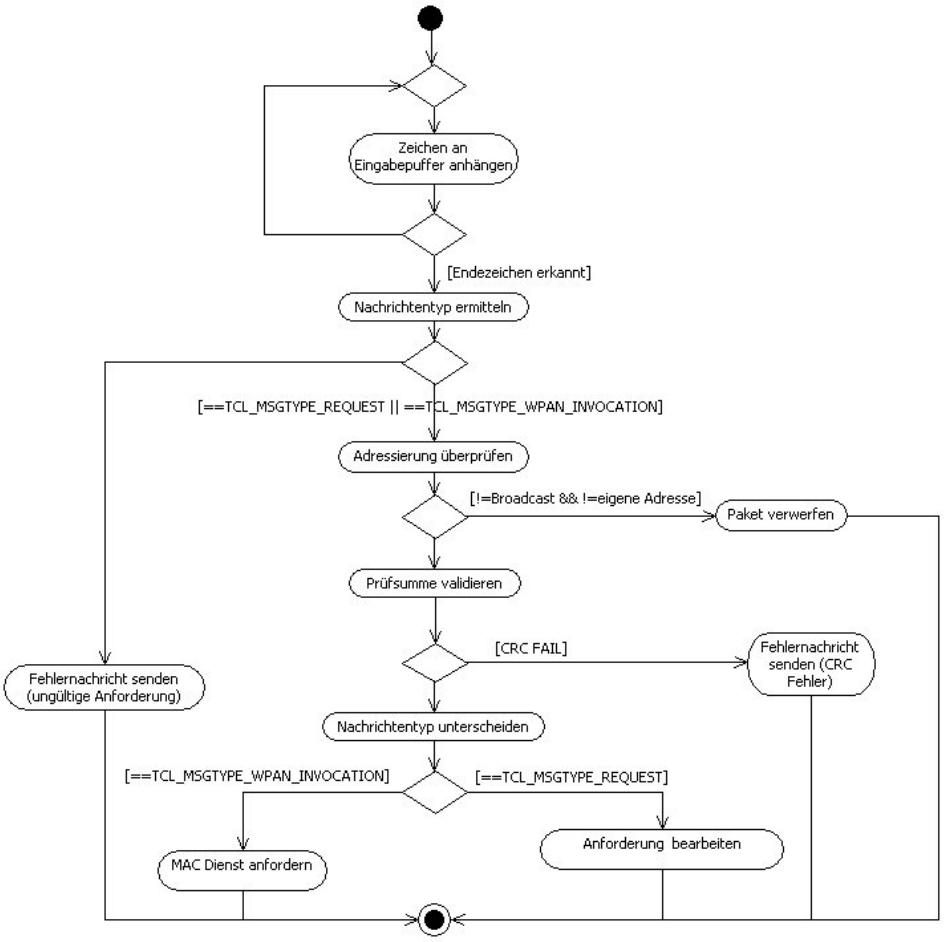


Abbildung 11.3: Behandlung eingehender Zeichen im TCL.

Für den Versand von Nachrichten wurde ebenfalls eine eigene Routine entwickelt (`tcl_gen_rsp_wrapper()`). Diese fügt zuerst den aktuellen Zeitstempel in die Nachricht ein, der aus einer Variable und dem Wert des Timerregisters zusammengesetzt wird. Anschließend wird die Nachricht zeichenweise in einen Ausgabepuffer kopiert. Zur Vermeidung des oben angesprochenen Problems der ineffizienten Byte-weisen Übertragung wird der Puffer dabei entweder sofort geschrieben, wenn er vollgelaufen ist oder erst nach Ablauf eines Timeout, was wiederum im `wpan_task()` abgeprüft wird.

Um den Start der Knotenapplikation gezielt verzögern zu können, beinhaltet der TCL einen Zustandsautomaten (Vgl. Abb. 11.4). Der aus der `main()`-Funktion erfolgte Aufruf von `tcl_init()` blockiert solange in einer Schleife, bis der zur Ausführung der Applikation erforderliche Zustand erreicht ist. Zustandsübergänge werden durch Steuerbefehle realisiert, mit einer Ausnahme, bei der ein ausgelöster Interrupt erforderlich ist. Dies wird an späterer Stelle wieder aufgegriffen.

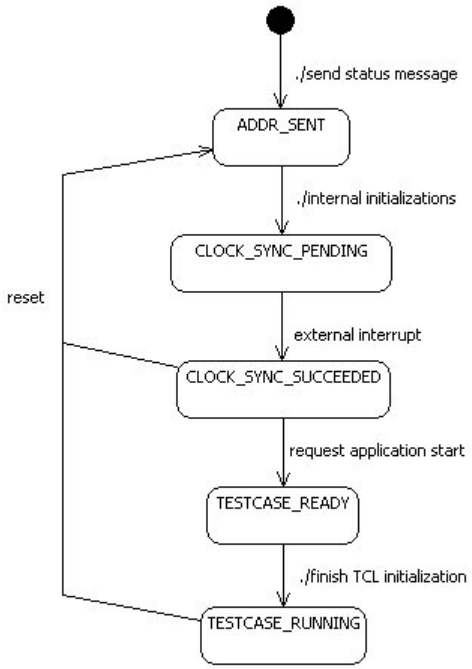


Abbildung 11.4: Zustandsautomat des TCL.

Von besonderem Interesse beim Einsatz von Knotenapplikationen ist deren Interaktion mit dem Stack. Dessen generelles Funktionsprinzip orientiert sich an den Kommunikationsprimitiven des IEEE 802.15.4-Standards: Dienste des MAC-Layer können dabei angefordert werden (*Request*), durchlaufen dann den Protokollstack und resultieren in einer Bestätigung (*Confirmation*). Umgekehrt zeigt der Stack die externe Anforderung eines Dienstes mittels einer *Indication* an, worauf eine Antwort (*Response*) zurückgesendet werden könnte (Vgl. Abb. 11.5).

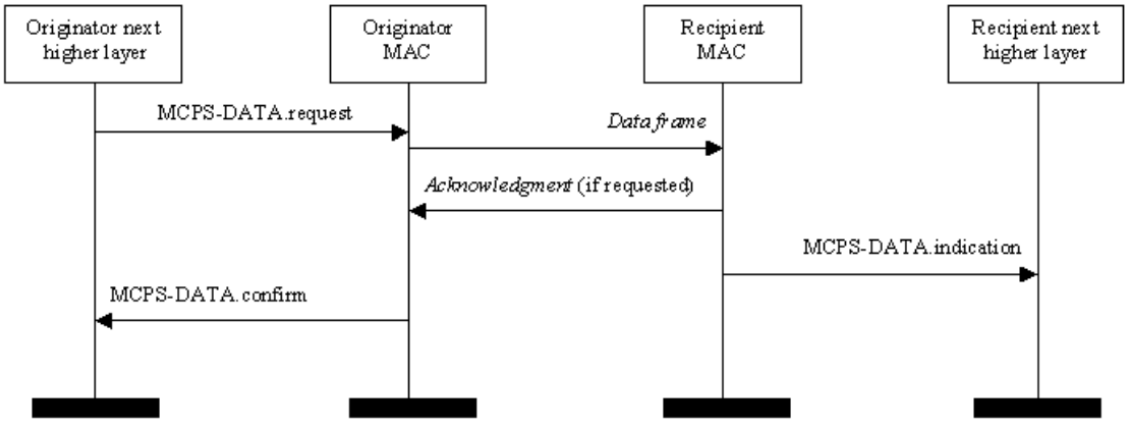


Abbildung 11.5: Datenübertragung zwischen zwei Knoten [Vgl. IEE06, S. 95].

Die vier genannten Kommunikationsprimitive werden im Stack durch Funktionen abgebildet, über die der MAC-Layer mit der Applikation interagiert. Requests und Responses sind dabei Bestandteil des Stack und aktiv rufbar (Präfix „`wpan_`“), Confirmations und Indications werden durch Callback-Funktionen

realisiert (Präfix „usr_“), die auf Applikationsebene – ggf. mit leerem Funktionsrumpf – vorhanden sein müssen. Um bei Aufruf solcher Funktionen nicht für jede Anwendung individuell Debugmeldungen erzeugen zu müssen, wurden im Protokollstack an den entsprechenden Stellen Callbacks eingefügt, bei deren Aufruf Nachrichten erzeugt werden.

Verkompliziert wird die Anbindung der Callback-Funktionen dadurch, dass im Stack keine schichtenübergreifenden Funktionsaufrufe möglich sind. Stattdessen werden Events erzeugt und über eine Warteschlange ausgetauscht. Dieses komplexe Verhalten soll am Beispiel einer Datenübertragung verdeutlicht werden (Vgl. Abb. 11.6). Der von der Anwendung initiierte Aufruf zum Versand eines Datenpakets (`wpan_mcps_data_req()`) wird zunächst in eine Queue eingefügt, später wieder daraus extrahiert und im MAC-Modul verarbeitet (`mcps_data_request()`). Die dort angebundene Callbackfunktion `tcl_mcps_data_req()` kopiert sämtliche Parameter des initialen Funktionaufrufes in eine Nachricht, die anschließend versendet wird (Vgl. Listing 11.1). Die Funktionen, die die Schnittstelle zur Anwendung bereitstellen, befinden sich dabei in der Datei `MAC/Src/mac_api.c`, die daraus gerufenen Funktionen mit den Präfixen „`mcps_`“ und „`mlme_`“ sind auf das Verzeichnis `MAC/Src/` verteilt. Die gerufenen Callbackfunktionen befinden sich in `TCL/Src/tcl_callbacks.c`.

```
1  void mcps_data_request(uint8_t *msg)
2  {
3      mcps_data_req_t mdr; // see mac_msg_types.h
4
5      // copy only the part of the buffer that contains the message header
6      memcpy(&mdr, BMM_BUFFER_POINTER((buffer_t *)msg), sizeof(mcps_data_req_t));
7
8  #ifdef TCL_USE_MAC_CALLBACKS
9
10     // the payload is contained in this buffer entry (but at a different position,
11     // which is not accessible via the mcps_data_req_t, see how the payload is
12     // copied to a different buffer-location in wpan_mlme_data_req()@mac_api.c)
13
14     // we need to create a temporary pointer like this is done in mcps_data_req()
15
16     // reference to the start of the buffer
17     uint8_t* pBufStart = BMM_BUFFER_POINTER((buffer_t*)msg);
18
19     // reference to the payload
20     uint8_t* pPayload = pBufStart + (LARGE_BUFFER_SIZE - FCF_SIZE - mdr.msduLength);
21
22     tcl_mcps_data_req(  mdr.SrcAddrMode,
23                         mdr.DstAddrMode,
24                         mdr.DstPANId,
25                         mdr.DstAddr,
26                         mdr.msduHandle,
27                         mdr.TxOptions,
28                         mdr.msduLength,
29                         pPayload
30                         );
31 #endif
32 ...
33 }
```

Listing 11.1: Aufruf einer Callbackfunktion im MAC Layer (Vgl. `MAC/Src/mac_mcps_data.c`).

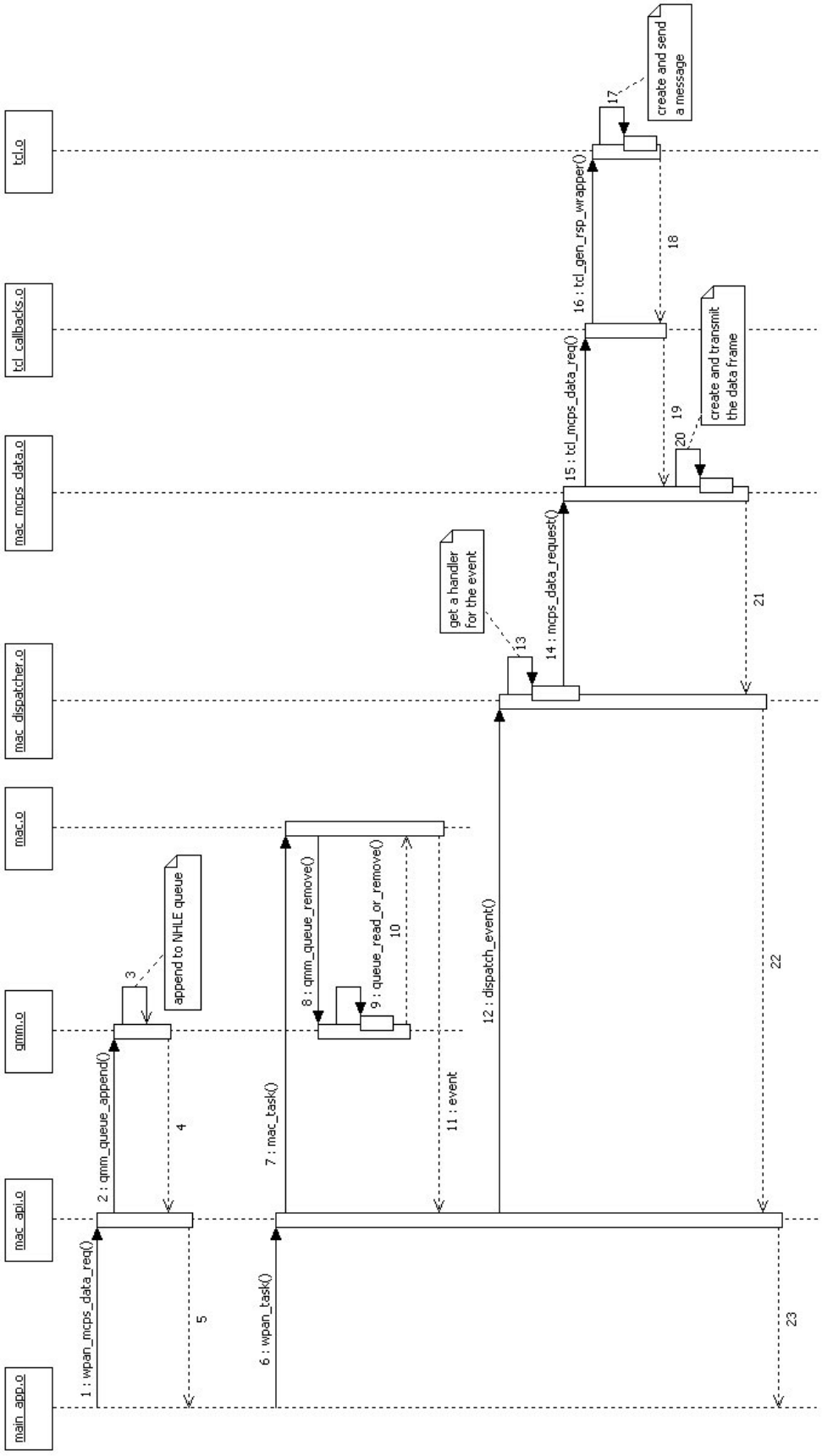


Abbildung 11.6: Anbindung der Callback-Nachrichten an Dienstanforderungen des Stack.

```
1  void tcl_mcps_data_req( uint8_t SrcAddrMode,
2                          uint8_t DstAddrMode,
3                          uint16_t DstPANId,
4                          uint64_t DstAddr,
5                          uint8_t msduHandle,
6                          uint8_t TxOptions,
7                          uint8_t msduLength,
8                          uint8_t* data           )
9  {
10    if(macCallbacksEnabled)
11    {
12       pMTcl->command   = TCL_CMD_WPAN_REQ;
13       pMTcl->attrib    = TCL_ATTR_MCPS_DATA;
14       pMTcl->len       = 1+1+2+8+1+1+1+msduLength;
15       memcpy(&pMTcl->data[ 0],&SrcAddrMode,1);
16       memcpy(&pMTcl->data[ 1],&DstAddrMode,1);
17       memcpy(&pMTcl->data[ 2],&DstPANId,2);
18       memcpy(&pMTcl->data[ 4],&DstAddr,8);
19       memcpy(&pMTcl->data[12],&msduHandle,1);
20       memcpy(&pMTcl->data[13],&TxOptions,1);
21       memcpy(&pMTcl->data[14],&msduLength,1);
22       memcpy(&pMTcl->data[15],data,msduLength);
23       tcl_gen_rsp_wrapper(pMTcl);
24    }
25 }
```

Listing 11.2: Erzeugung von Nachrichten durch die Callback-Funktion im TCL-Layer (Vgl. `TCL/Src/tcl_callbacks.c`).

Mit dem soeben beschriebenen Mechanismus lässt sich die Interaktion zwischen MAC- und NWK-Schicht beobachten. Werden bspw. Daten zwischen Knoten ausgetauscht, so lässt sich damit feststellen, ob die Übertragung erfolgreich war, nicht jedoch, ob dabei Störungen auftraten und die Datenübertragung wiederholt werden musste. Um dies ermitteln zu können, muss die Interaktion zwischen MAC- und PHY-Schicht ebenfalls mit Nachrichten erzeugenden Callback-Funktionen versehen werden. Über diese Schnittstelle werden MAC-Frames ausgetauscht, was wiederum über Queues realisiert wird.

Die Behandlung von Frames ist Transceiver-spezifisch implementiert, selbst für die zum Test eingesetzten ATRF230A und ATRF230B unterscheidet sie sich wesentlich. Der Versand eines Frames über einen ATRF230A ist dabei in Abbildung 11.7 vereinfacht dargestellt. Das Sequenzdiagramm beinhaltet den in Abbildung 11.6 ausgelassenen Teil. Durch die Funktion `send_frame()` wird der Frame letztlich in den Transceiver kopiert und physisch übertragen.

Die Platzierung der Callback-Funktion, welche den versandten Frame in eine Nachricht dupliziert, ist dabei besonders kritisch. War die Übertragung unbestätigt, kann die Nachricht sofort erzeugt werden. Andernfalls muss auf Empfang des zugehörigen ACK-Frames (bzw. dessen Ausbleiben) gewartet werden, da erst dann der Übertragungsstatus bekannt ist. Übertragungsbestätigungen müssen innerhalb einer fixen Zeitspanne eingehen [Vgl. IEE06, Abschnitt 7.5.6.4.2].

Generell signalisiert der Transceiver Ereignisse dem Mikrocontroller gegenüber durch einen Interrupt, in der zugehörigen Interrupt-Service-Routine wird die

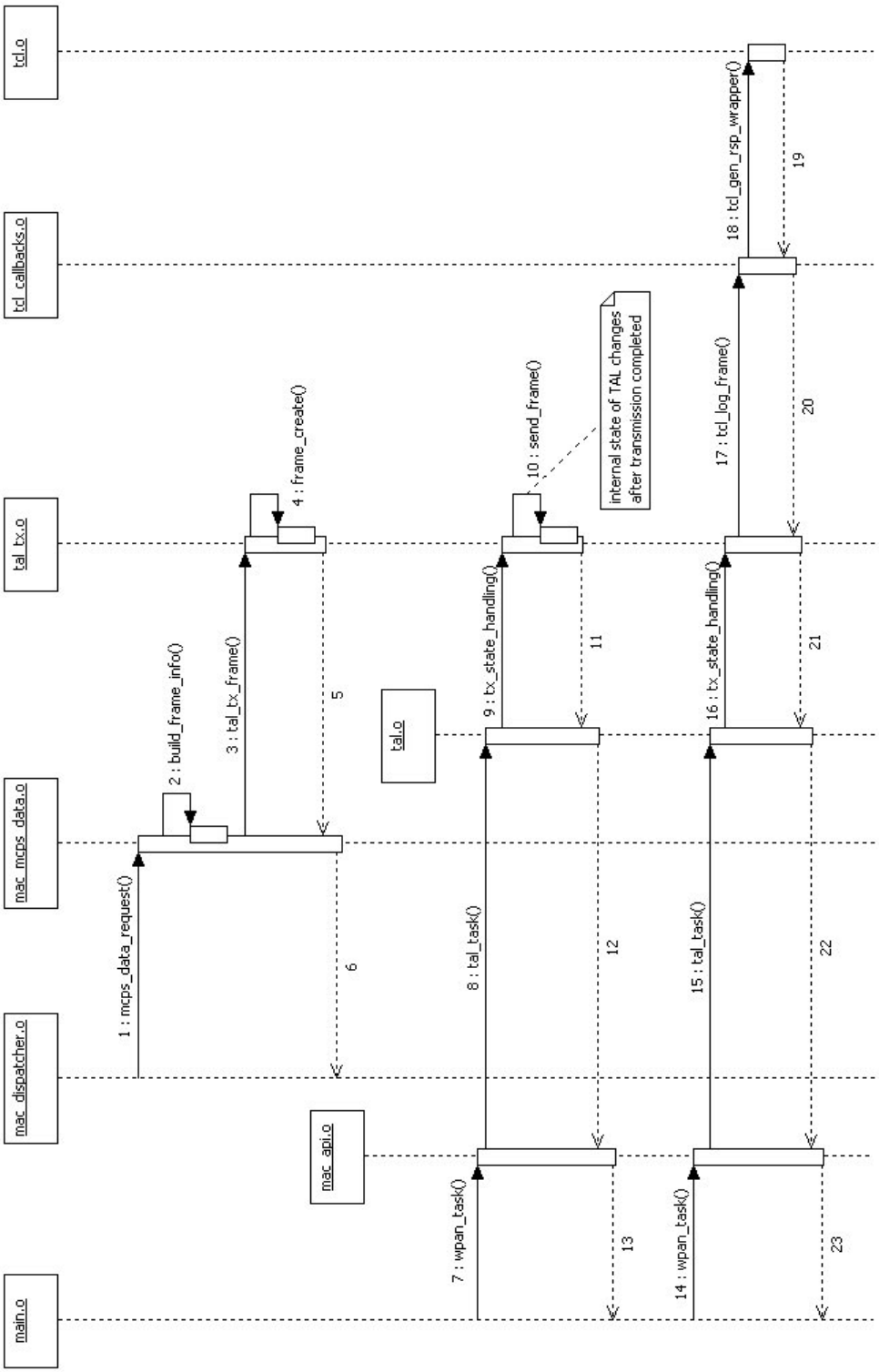


Abbildung 11.7: Vereinfachte Darstellung des Frame-Versand bei Verwendung eines ATRF230A.

Ereignisursache ermittelt. War dies ein empfangener Frame, muss er schnellstmöglich in den Mikrocontroller heruntergeladen werden, da der Transceiver nur über einen Eingangspuffer für einen Frame verfügt und der Frame sonst verloren geht[139].

Bezüglich einer ggf. ausstehenden Übertragungsbestätigung muss hier nach verwendetem Transceiver unterschieden werden: Der ATRF230B wird durch den Stack für den Extended Mode konfiguriert, wodurch er das ACK-Frame selbst erkennt. Der ATRF230A besitzt dieses Merkmal nicht, so dass hier bei Start einer Übertragung ein dekrementierender Timer gestartet wird, welcher entweder durch das eingehende ACK-Frame wieder gestoppt wird oder die Signalisierung der fehlgeschlagenen Übertragung bewirkt.

In jedem Fall ist die asynchrone Meldung empfangener Frames problematisch, da sie einerseits unmittelbar in Nachrichten überführt werden müssen, andererseits so schnell hintereinander eingehen können, dass dies gar nicht mehr möglich ist. Die zweite unangenehme Eigenschaft der Frame-Behandlung ist, dass während sämtlicher Datenübertragungen zwischen Mikrocontroller und Transceiver Interruptsperren eingesetzt werden, welche den zur Erzeugung von Zeitstempeln verwendeten Timer beeinflussen können. Wird durch die Sperre mehr als ein Überlauf des Timers verpasst, äußert sich das in einem nicht behebbaren Nachlauf der lokalen Uhr.

Stack und Knotenapplikation sind nicht eigenständig lauffähig und müssen immer zusammen übersetzt werden. Über einen Schalter wird während der Übersetzung entschieden, ob der TCL dazugebunden wird. Ist er aktiviert, kann durch zwei weitere Schalter beeinflusst werden, ob während der Behandlung von MAC-Funktionsrufen bzw. MAC-Frames Nachrichten erzeugt werden. Zu Testzwecken lässt sich somit eine Applikation mit Callback-Rufen übersetzen, die im Produktiveinsatz nicht mehr vorhanden sind. Einmal einkompiliert lassen sich zur Laufzeit die Aufrufe der Callbackfunktionen zwar nicht mehr verhindern, aber die Nachrichtenerzeugung über Schalter deaktivieren.

11.3 Implementierung einer Knotenfirmware

Zur Demonstration der Möglichkeiten, die der Atmel IEEE 802.15.4-Stack bietet, aber vor allem auch als „Proof-of-concept“ der Testplattform selbst wurde eine Beispielanwendung entwickelt. Diese implementiert eine proprietäre Netzwerkschicht und orientiert sich an den Beispielen, die beim Stack mitgeliefert werden.

[139] Empfangene Frames werden ebenfalls als Nachrichten ausgegeben, wobei ergänzende Informationen, wie die Empfangsfeldstärke, mit übertragen werden.

Die Anwendung unterstützt die zwei Rollen *Coordinator* und *End-Device*. An das vom Coordinator eröffnete Netz können sich End-Devices anmelden. Bei Erfolg senden sie in regelmäßigen Intervallen Datenpakete an den Coordinator, wobei die Übertragung bestätigt wird.

Designziel war außerdem, die Anwendung generisch zu gestalten, damit auf allen Knoten eine einheitliche Firmware eingesetzt werden kann, die erst zur Laufzeit spezialisiert wird. In der Applikation sind folgende Parameter änderbar, die anfangs mit Standardeinstellungen belegt sind:

- Rolle (Coordinator, End-Device)
- Kanal, Page, zu benutzende PAN-ID
- Beacon- und Superframeorder
- Short-Adresse des Coordinators
- Länge des zu übertragenden Datenpaketes, Übertragungsintervall
- zu scannende Kanäle (Coordinator: Prüfung auf existierende PANs, End-Device: Suche nach dem Coordinator)
- Länge der Scan-Intervalle, rollenbezogen

Die Abarbeitung der Knotenapplikation beginnt mit einem Initialisierungblock. Danach wird ein Reset des Stack angefordert, der wie bereits im vorigen Kapitel beschrieben, lediglich in die Befehlswarteschlange einfügt wird. Den Abschluss der `main()`-Funktion bildet eine Endlosschleife, welche den `wpan_task()` ausführt, in dem u.a. die Befehlswarteschlange bearbeitet wird. Beim ersten Aufruf befindet sich bereits ein *Reset Request* darin, bei dessen Bearbeitung eine *Reset Confirmation* ausgelöst wird. Dies resultiert im Ruf der Callbackfunktion `usr_mlme_reset_conf()`, worin sogleich die nächste Anforderung gestellt wird:

```
1  void usr_mlme_reset_conf(uint8_t status)
2  {
3     if (status == MAC_SUCCESS)
4     {  if(runtime_role==COORD)
5        {
6           // Set the short address of this node.
7           wpan_mlme_set_req(macShortAddress,
8                     (void*)(&runtime_coord_shortAddr));
9        }
10       else // role == ED
11       {
12          // Initiate an active scan over all channels to
13          // determine which channel is used by the coordinator.
14          wpan_mlme_scan_req(
15                         runtime_scantype,
16                         runtime_scan_channelmask,
17                         runtime_scaninterval_ed_short,
18                         runtime_channelPage);
19       }
20    } // status==MAC_SUCCESS
21    else
22    {
23       // something went wrong; restart
24       wpan_mlme_reset_req(true);
```

```
25    }
26 }
27
28 void usr_mlme_scan_conf(uint8_t status,
29                         uint8_t ScanType,
30                         uint8_t ChannelPage,
31                         uint32_t UnscannedChannels,
32                         uint8_t ResultListSize,
33                         void *ResultList)
34 {
35    if(runtime_role==COORD)
36    {
37       if( false==wpan_mlme_start_req(runtime_panId,
38                       runtime_channel,
39                       runtime_channelPage,
40                       runtime_beaconOrder,
41                       runtime_superframeOrder,
42                       true, false, false) )
43        {
44          // something went wrong, restart
45          wpan_mlme_reset_req(true);
46        }
47    }
48    ...
49 }
```

Listing 11.3: Abwechselnder Aufruf von Callbackfunktionen und Funktionen die Dienste des MAC Layer anfordern (Vgl. `Applications/App_2_parametrizable/Src/main_common.c`).

Auf diesem Wechselspiel zwischen Anforderungen und Bestätigungen baut die gesamte Applikation auf. In der gleichen Callbackfunktion ergibt sich durch die beiden möglichen Rollen ein unterschiedliches Verhalten. Ihre aktuellen Funktionsparameter und die Rückgabewerte aufgerufener Requests beeinflussen ebenfalls, welche Anforderung als nächstes gestellt wird. Unter den Annahmen, dass

1. der Coordinator das Netz bereits gestartet hat, wenn sich ein End-Decive assoziieren möchte,
2. alle Funktionsrufe fehlerfrei ausgeführt werden konnten (d.h. `wpan_xxx() == true`) und
3. alle Bestätigungen Erfolg anzeigen (d.h. in den Callbackfunktionen `usr_xxx()` gilt für den Parameter `status == MAC_SUCCESS`),

ergibt sich für die in Abbildung 11.8 dargestellte Abarbeitungsfolge.

Die Übersetzung erfolgt zusammen mit dem Stack anhand eines Makefiles. Wird zusätzlich der optionale TCL-Layer eingebunden, lässt sich das Verhalten der Knotenapplikation beobachten. Um die Knotenapplikation zur Laufzeit parametrieren zu können, ist er jedoch zwingend erforderlich. Zur Parametrierung der Anwendung zur Laufzeit wird ein separates Modul verwendet, welches im nächsten Kapitel vorgestellt wird. Der Quellcode dieser Anwendung ist auf der beiliegenden CD im Verzeichnis `stack/Applications/App_2_parametrizable/` zu finden.

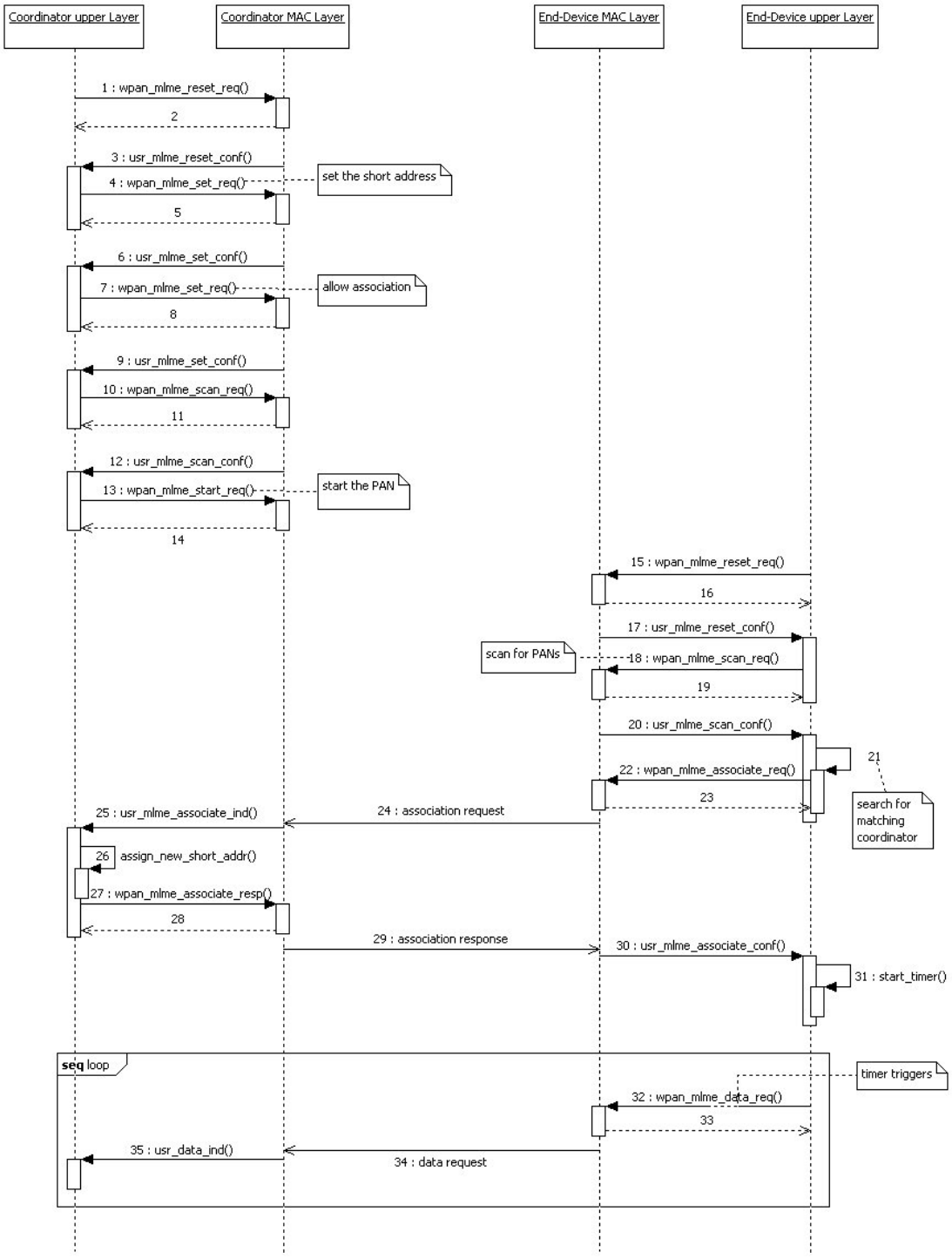


Abbildung 11.8: Abarbeitungssequenz der Beispielapplikation.

11.4 Laufzeitparametrierung der Knotenfirmware

In diesem Kapitel wird die Änderung von Parametern der Knotenfirmware diskutiert, wobei damit alle Parameter gemeint sind, die nicht den Stack direkt betreffen[140]. Dies sind einerseits Parameter des TCL-Layers[141], andererseits Parameter der Knotenapplikation. Die Parameter des TCL sind nur Schalter, ihr Umfang ändert sich nicht. Knotenapplikationen können sich jedoch stark voneinander unterscheiden, woraus sich eine Reihe von Problemen ergibt:

- Der adressierte Parameter ist in der Knotenfirmware gar nicht vorhanden.
- Der Parameter ist zwar vorhanden, aber nicht änderbar oder nur vor Start der Knotenapplikation änderbar und nicht mehr zur Laufzeit.
- Der betreffende Parameter soll auf einen Wert außerhalb seines Wertebereiches geändert werden.

Zur Verwaltung der Parameter der Knotenapplikation wurde daher ein eigenes Modul entwickelt, welches in die Firmware integriert werden kann. Während der Initialisierung der Firmware wird darin eine Liste aller verfügbaren Applikationsparameter aufgebaut. Jeder Listeneintrag wird durch eine Datenstruktur repräsentiert (`pgm_param_t`), die neben dem aktuellen Parameterwert, dessen eindeutige Kennung, Änderbarkeit und möglichen Wertebereich erfasst. Ein Parameter kann entweder unveränderlich sein oder nur in bestimmten Zuständen bzw. abhängig von anderen Parametern änderbar sein. Wertebereiche können auf vier verschiedene Arten angegeben werden (`limit_t`): entweder durch ihre Intervallgrenzen, durch eine Bitmaske in der nur ein Bit gesetzt werden darf oder beliebige Bits, bzw. sie sind nicht eingeschränkt. Da auf dem Mikrocontroller nur begrenzt Speicher zur Verfügung steht, wird zwar die Liste im RAM gehalten, ihre statischen Elemente (`pgm_info_t`) aber in den Flash-Speicher ausgelagert (Vgl. Listing 11.4).

```
1  typedef struct PGMEM_PARAM_T {
2     byte     paramId;       // parameter ID
3     byte     width;         // width of data, bitmask, borders
4     void*    value;         // current value
5     PGM_P*   furtherInfo;   // pgm_info_t
6  } pgm_param_t;
7
8  typedef struct PGMINFO_T {
9     byte  chgFlag; // Changeability Flag, bitwise boolean, evaluation order: b0, b1,
          ...
10                   // according to bit hierarchy: bx is ignored if (by==false)&(y<x)
11                   // b0: Changeability in general,
12                   // b1: Changeability at runtime
13                   // b2: Dependencies for changeability
14
```

[140] Die Parameter des Stack werden durch die PIB gekapselt, für die eigene Befehle zur Abfrage und Manipulation der Parameter verfügbar sind.
[141] Bspw. ob Callbacks angeschaltet sind.

```
15    limit_t  limit_type; // limitation type
16    uint8_t* limit1;  // [exclusive] bitmask or lower limit
17    uint8_t* limit2;  // upper limit or NULL
18 } pgm_info_t;
19
20 typedef enum LIMIT_T {
21    BITMASK   = TCL_PARAMETER_LIMIT_BITMASK,           // simple matching mask
22    EXCLUSIVE = TCL_PARAMETER_LIMIT_BITMASK_EXCLUSIVE, // only one bit matching in
          mask
23    BORDERS   = TCL_PARAMETER_LIMIT_BORDERS,           // matching interval
24    NA                                                 // no masking available
25 } limit_t;
```

Listing 11.4: Datenstrukturen und -typen zur Definition von Laufzeitparametern. (Vgl. `stack/Applications/App_2_parametrizable/Inc/runtime_params.h`).

Zum Abfragen und Ändern von Parametern sind drei Steuerbefehle verfügbar:

- Mittels eines *get*-Befehls kann der aktuelle Wert abgefragt werden.
- Durch einen *set*-Befehl kann versucht werden, den Parameter zu verändern.
- Eine *supported-values*-Anfrage ermittelt die Änderbarkeit eines Parameters und, falls dies zutrifft, dessen mögliche Limitierungen[142].

In der zugehörigen Paketcodierung wird der Nachrichtentyp `TCL_MSGTYPE_REQUEST` verwendet, für das Kommandoflag ist `TCL_CMD_GET`, `TCL_CMD_SET`, oder `TCL_CMD_SUPPORTEDVALUES` zu setzen und im ATTR-Feld kann ein beliebiger bekannter Laufzeitparameter, bspw. `TCL_ATTR_RUNTIME_FIRMWAREID`, eingetragen werden.

Gehen solche Steuerbefehle im TCL ein, wird anhand des `ATTR`-Flags unterschieden, ob der Parameter den TCL betrifft oder die Knotenapplikation. Im ersten Fall wird er vom TCL selbst behandelt, sonst wird die Anfrage an das Parametrierungsmodul delegiert (Vgl. Abb. 11.9).

11.5 Snifferknoten

Obwohl eine Reihe kommerzieller Sniffer erhältlich ist (Vgl. Kapitel 4.3), so sind diese nur mit ausgewählter Hardware einsetzbar, für die der Hersteller eine eigene Snifferfirmware mitliefert. PC-seitig kann zum Betrieb ebenfalls nur proprietäre Software verwendet werden, was eine Kopplung an den Testserver erschwert. Für den Einsatz im Testbed wurde daher ein eigener Sniffer entwickelt, der als Nebenprodukt der Erzeugung von Callback-Nachrichten im Knoten bei empfangenen Frames betrachtet werden kann. Als Hardwarebasis dient ein reguläres Funkmodul, auf dem eine besondere Firmware verwendet wird. Die Knotenfirmware ähnelt einer normalen Applikation, der TCL bleibt unverändert. Die

[142] Durch einen solchen Befehl kann beispielsweise sichergestellt werden, dass nur Knoten für einen Testfall verwendet werden, die die erforderlichen Merkmale unterstützen.

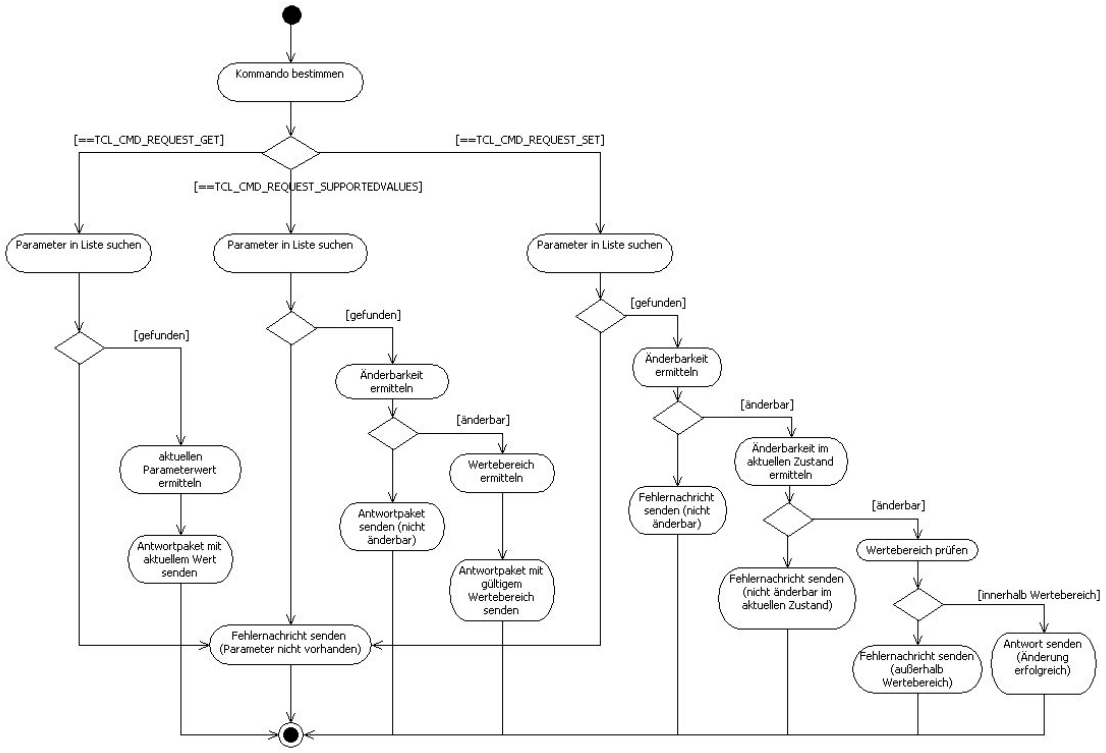


Abbildung 11.9: Vereinfachte Darstellung der Behandlung von Steuerbefehlen im Parametrierungsmodul.

Parametrierung beschränkt sich hierbei auf die Auswahl eines Kanals, auf dem zu lauschen ist. Im Gegensatz zu einer Knotenfirmware für den Normalbetrieb werden keine Dienste des Protokollstack in Anspruch genommen, so dass hier weder Callback-Funktionen vorhanden sind noch entsprechende Nachrichten erzeugt werden.

Zur Übersetzung der Knotenapplikation ist der Schalter `SNIFFER` zu setzen, wodurch ein zum regulären Betrieb des Stack abweichendes Verhalten bewirkt wird. Normalerweise empfängt ein Knotens alle Frames und verwirft sie bei unpassender Zieladressierung wieder. Andernfalls wird der Frame durch den Stack bearbeitet[143]. Durch den gesetzten Schalter wird einerseits diese Filterung ausgesetzt, andererseits wird die Bearbeitung von Frames komplett unterbunden.

Der hier besprochene Snifferknoten leitet lediglich alle empfangenen Frames über die serielle Verbindung – in Nachrichten verpackt – weiter. Die Auswertung kann nur auf der Empfängerseite vorgenommen werden. Da die hier verarbeiteten MAC-Frames komplette MPDUs[144] enthalten, in denen höherschichtige PDUs gekapselt sein können, ist bei Kenntnis des entsprechenden Protokollformats auch deren Auswertung möglich.

[143] Wenn es erforderlich ist, wird ein ACK-Frame zurückgesendet, anschließend werden Ereignisse ausgelöst, die den Ruf von Callback-Funktionen bewirken.
[144] MAC Protocol Data Units.

11.6 Elektrische Verschaltung

Um Knoten bedarfsweise neu starten zu können, wurde der Reset-Pin des Mikrocontrollers auf ein I/O-Pin rückgekoppelt. Dadurch wird es ermöglicht, dass sich der Mikrocontroller durch ein Steuerkommando selbst zurücksetzt. Der Transceiver ist davon jedoch unbetroffen, dessen Reset-Pin ist unzugänglich, so dass er vor dem Reset des Mikrocontrollers durch Funktionsaufrufe rückgesetzt werden muss:

```
1  /** ------------------------------------------------------------
2   * @brief performs a reset of the node as like as the
3   * reset button has been manually pressed
4   * ------------------------------------------------------------ */
5  void reset_node()
6  {  ...
7     wpan_mlme_reset_req(true);
8     tal_reset(true);              // invokes: init_tal_infobase()
9                                   // and trx_reset()
10    tal_rx_enable(PHY_TRX_OFF);   // switch transceiver off
11
12    // reconfigure RESETPIN_PORT:RESETPIN_PIN to be an output pin
13    RESETPIN_DDR  |=  _BV(RESETPIN_PIN);
14    // set RESETPIN_PORT:RESETPIN_PIN to low, force a node reset
15    RESETPIN_PORT &= ~ _BV(RESETPIN_PIN);
16 }
```

Listing 11.5: Funktion zum Reset des eigenen Knotens.

Die bei aufgetretenen Ereignissen von den Knoten versandten Nachrichten können Zeitstempel enthalten. Um Ereignisse knotenübergreifend korrelieren zu können, sind synchronisierte Uhren erforderlich. Es wurde daher ein „Bussystem“ zwischen den Knoten aufgebaut, durch welches alle Knoten mit einem einheitlichen miteinander verbunden werden[145].

In der oben erwähnten Stackerweiterung ist ein Zustandsautomat enthalten, welcher einen Zustand zur Uhrensynchronisation besitzt. Wird in diesen gewechselt, so schalten alle Netzknoten auf dem Bus-Pin einen Pin-Change-Interrupt frei. Ein Impuls auf dem Bus bewirkt die Auslösung dieses Interrupts, wodurch der Zustand wieder verlassen wird:

```
1  /** ------------------------------------------------------------
2   * @brief enables PINCHANGEPIN_PORT:PINCHANGEPIN_PIN to serve
3   * as a source for a pinchange interrupt
4   * ------------------------------------------------------------ */
5  void enable_pinchange_irq()
6  {  ...
7     // Pin Change Interrupt Register - enable interrupt group
8     PCICR  |= _BV(PCIE0);
9
10    // Interrupt Mask Register - enable PINCHANGEPIN_PIN to
11    // trigger an interrupt
12    PCMSK0 |= _BV(PINCHANGEPIN_PIN);
13    ...
14 }
15 ...
```

[145] Ihre Masseanschlüsse wurden ebenfalls miteinander verbunden.

```
16 /** ------------------------------------------------------------
17  * @brief Interrupt service routine for PCINT0|1|2
18  * ------------------------------------------------------------ */
19 ISR(PCINT0_vect)
20 {
21    disable_pinchange_interrupt();
22
23    if(currentState!=CLOCK_SYNC_SUCCEEDED)
24    {  start_timer();
25       currentState = CLOCK_SYNC_SUCCEEDED;
26    } else {
27       send_status_message();
28    }
29 }
```

Listing 11.6: Erkennung des Steuerimpulses zur Uhrensynchronisation.

Zur Auslösung des Impulses existiert ein dedizierter Master-Knoten, der ein weiteres Pin auf den Bus rückkoppelt. Durch einen Steuerbefehl wird dann der Impuls ausgelöst:

```
1 /** ------------------------------------------------------------
2  * @brief pulses pin PINCHG_TRIGGER_PIN (pulse duration 1us)
3  * ------------------------------------------------------------ */
4 void force_pin_change()
5 {
6    PINCHG_TRIGGER_PORT |=  _BV(PINCHG_TRIGGER_PIN);   // H
7    PAL_WAIT_1_US();
8    PINCHG_TRIGGER_PORT &= ~_BV(PINCHG_TRIGGER_PIN);   // L
9 }
```

Listing 11.7: Erzeugung des Steuerimpulses zur Uhrensynchronisation.

Die angegebenen Quellcodeauszüge wurden der Datei `TCL/src/tcl.c` entnommen und für die Darstellung nochmals vereinfacht. Zur Abstraktion verwendendete symbolische Bezeicher für Ports und Portpins werden in `TCL/inc/tcl.h` definiert.

12 Erweiterung um Steuerknoten

Die derzeit einzige Möglichkeit, auf dresden elektronik-RCBs per USB zuzugreifen und damit die Anbindung an den Testserver zu realisieren, sind Sensor-Terminal-Boards. Dies ist ineffektiv, da das STB einerseits für die bloße kommunikative Anbindung überdimensioniert ist[146] und andererseits Funktionen, die im Testbed erforderlich sind, fehlen. Als alternative Lösung können sogenannte Steuerknoten eingesetzt werden, hinter denen sich folgende Konzepte verbergen:

Hierarchisierung: Funkmodule werden nicht mehr direkt an den Testserver angeschlossen, sondern über Steuerknoten entkoppelt[147] (Vgl. Abb. 12.1). Dies erlaubt den Einsatz von mehr Funkmodulen, als der Testserver (virtuelle) serielle besitzt. Nachteilig ist, dass die Kommunikation mit den Funkmodulen im Steuerknoten gemultiplext werden muss, wodurch Knoten einerseits trägerer auf Steuerbefehle reagieren, andererseits vom Knoten versandte Nachrichten erst mit einer größeren Latenz im Server eingehen.

Zusätzliche Abstraktionsschicht: Es lassen sich Varianten von Steuerknoten realisieren, die sich gegenüber dem Server gleich verhalten, aber verschiedene Funkmodule adaptieren. Deren Anbindung ist nicht mehr auf USB festgelegt, sondern könnte auch per USART[148] erfolgen. Durch unterschiedliche mechanische Schnittstellen und durch flexible Regelung der Betriebsspannung sind Funkmodule unterschiedlicher Hersteller anschließbar.

Flexibler Anschluss der Steuerknoten an den Testserver: Bei Neukonzeption eines Steuerknotens ist dessen Verbindung mit dem Testserver frei wählbar. Statt USB kann beispielsweise Ethernet genutzt werden, wodurch einerseits größere Leitungslängen möglich werden, andererseits eine aufwändige und fehlerträchtige Verkabelung[149] entfällt. Ethernet erlaubt auf höheren Schichten sogar die Sicherung der Integrität übertragener Daten.

Funktionale Trennung: Durch Steuerknoten können Funktionen, die sich weder auf dem Testserver noch in den Funkmodulen platziert lassen, klar getrennt werden: vollständiges An- und Abschalten von Funkmodulen, Steuerung der Uhrensynchronisation, An- und Abschalten von Störquellen[150], Ansteuerung von zwischen den Funkknoten platzierten Dämpfungsgliedern.

[146] Es stellt Funktionen bereit, die nicht benötigt werden und ist teuer in der Herstellung (60 €).
[147] Etwa 4-16 Funkmodule pro Steuerknoten.
[148] Universal Asynchronous Receiver/Transmitter.
[149] Da PCs in der Regel nur wenige USB-Anschlüsse aufweisen, wären USB-Hubs erforderlich.
[150] Bswp. Wireless LAN-Sendern.

Die Umsetzung eines Steuerknotens kann im einfachsten Fall durch einen Mikrocontroller erfolgen, der über mehr als einen USART verfügt. In der Atmel AVR-Familie sind derzeit maximal 8 USARTs möglich. Eine Alternative dazu stellen die ebenfalls von Atmel angebotenen ARM-Prozessoren dar, die von dresden elektronik bereits in anderen Geräten verwendet werden. Würde ein normaler Mikrocontroller eingesetzt, könnte dessen Ethernet-Anbindung über einen LANCOM-Adapter erfolgen. Die ARM-Familie unterstützt dies bereits durch den Prozessor.

Steuerknoten sind derzeit noch nicht Bestandteil der Testplattform, sollen aber später folgen. Ihre Anbindung wurde daher im Testserver bereits in Ansätzen berücksichtigt. Die Kommunikation mit dem Steuerknoten erfordert jedoch ein geeignetes Kommunikationsprotokoll. Wird davon ausgegangen, dass die Anbindung regulärer Knoten an den Server nur noch per Steuerknoten erfolgt, könnte ein separates Protokoll zum Einsatz kommen. Sinnvoller erscheint es jedoch, beide parallel betreiben zu können, die Kommunikation gegenüber regulären Knoten in den Steuerknoten lediglich durchzutunneln und das bisherige Protokoll zu erweitern. Dann benötigen Steuerknoten eine „MAC-Adresse“, die sich von regulären Funkknoten deutlich unterscheidet. Es wird daher vorgeschlagen, entweder eine unvergebene Herstellerkennung[151] zu verwenden[152] oder mit der bisherigen[153] zu arbeiten, aber das für die Gerätekennung verwendete Intervall von der Obergrenze beginnend rückwärts zu verwenden.

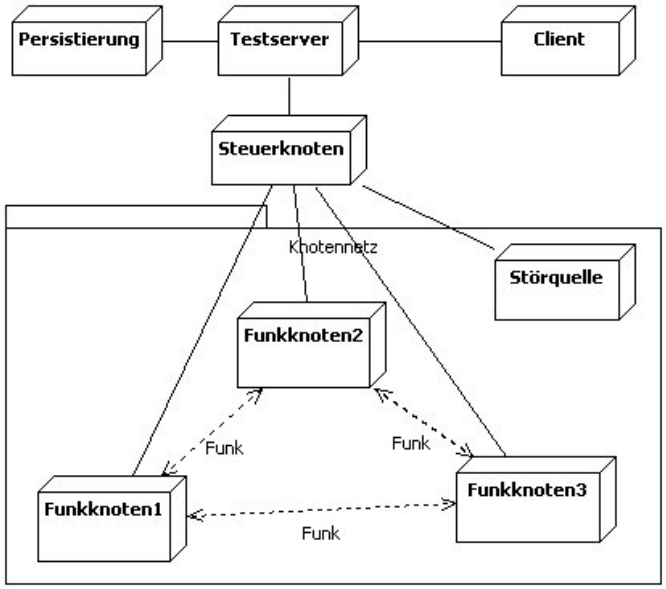


Abbildung 12.1: Erweiterung der Systemarchitektur um Steuerknoten.

[151] Die IEEE 802.15.4 MAC-Adressen teilen sich in 32 Byte Hersteller- und 32 Byte Gerätekennung.
[152] Mit dem Risiko, dass sie zukünftig doch noch vergeben wird.
[153] 00-04-25-FF.

13 Testserver

Die beiden Hauptaufgaben des Testservers sind die Verwaltung des Knotennetzes und die Behandlung von Testfällen. Wie im Kapitel 8 angesprochen, besteht der Testserver sowohl aus einer Hardware-Komponente, als auch einem zugehörigen Softwareprogramm. Die nachfolgenden Betrachtungen beziehen sich nur auf die Software-Komponente.

Im Zuge des experimentellen Prototypings wurde ein erster Server in C entwickelt, der die Kommunikation mit den Funkknoten über eine serielle Verbindung ermöglichte. Der ursprüngliche Plan – dessen evolutionäre Weiterentwicklung – stellte sich als ungeeignet heraus, da durch einen prozeduralen Ansatz viel Zeit durch die selbst zu realisierende Speicherverwaltung verloren geht. Die triviale objektorientierte Umsetzung wäre in C++ möglich gewesen unter Kapselung des bisherigen Quellcodes. Um jedoch die angestrebte Plattform- und Betriebssystemneutralität zu erreichen, fiel die Wahl schließlich auf Java. Die Pflege von Makefiles entfällt dort, der Server ist ohne Modifikation des Quellcodes auf anderen Plattformen lauffähig.

Zur Strukturierung der erbrachten Funktionalität wurde der Quellcode auf mehrere Packages aufgeteilt (Vgl. Abb. 13.1). Die links dargestellten Pakete umfassen die Behandlung von Testfällen, die auf der rechten Seite gezeigten beinhalten Funktionen bezüglich des Knotennetzes. Zentral angeordnet sind allgemein verwendete Packages. Die jeweils rechts und links unten befindlichen Packages `jaxb.*` beinhalten automatisch generierte Data Binding-Klassen für die Verarbeitung von XML-Dateien. Der Quellcode des Servers ist auf der beiliegenden CD zu finden.

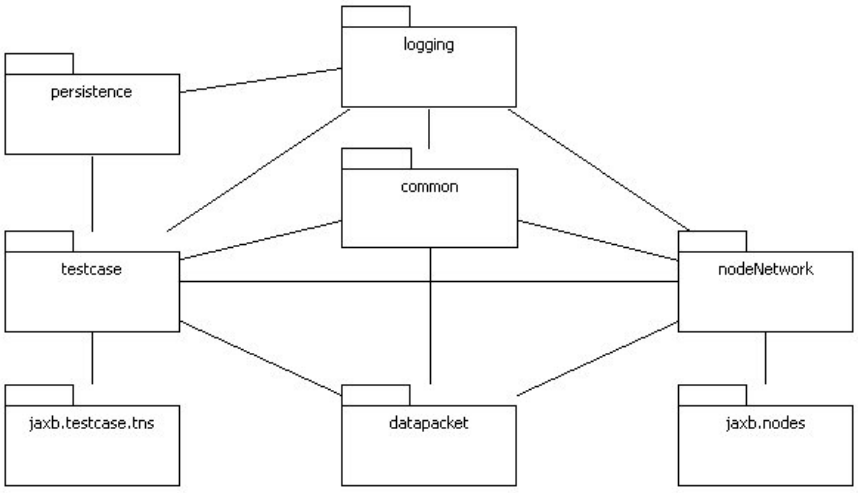


Abbildung 13.1: Package-Diagramm des Testservers.

Im Package *common* enthalten ist dabei die Klasse *Testserver.java*, welche die `main()`-Methode beinhaltet. Damit werden Knotennetz und Testfallverwaltung initialisiert sowie ein *ConsoleInputHandler* gestartet, über den sich der Server steuern lässt. Weiterhin befindet sich in diesem Package die Klasse *Transferobject.java*, eine Hilfsklasse zur Übermittlung von Fehlercodes, Fehlernachrichten und Objektreferenzen. Die Klasse *Helper.java* stellt Methoden zur Ausgabe, Formatierung und Konvertierung verschiedener Datentypen bereit. Alle weiteren Pakete werden in den nächsten Kapiteln überblicksartig vorgestellt.

13.1 Verarbeitung von XML-Dokumenten

Neben ihrer Verwendung als Testfall-Steuerdateien dienen XML-Dokumente im Server zur Definition der Knotenanbindung. Zu ihrer Verarbeitung wird ein validierender Parser eingesetzt, der Zugriff auf die darin enthaltenen Knoten erfolgt nach dem Modell der XML-Datenbindung. Die Erstellung der zugehörigen Bindungsklassen wird vorab durch den Schemacompiler `xjc` vorgenommen, der basierend auf der Schema-Definition ein eigenes Package generiert, in dem XML-Knoten durch entsprechende Klassen repräsentiert werden. Während des Parsens werden daraus Objekte erzeugt, über die einfach auf den Inhalt des Dokuments zugegriffen werden kann (Unmarshalling[154], Vgl. Abb. 13.2).

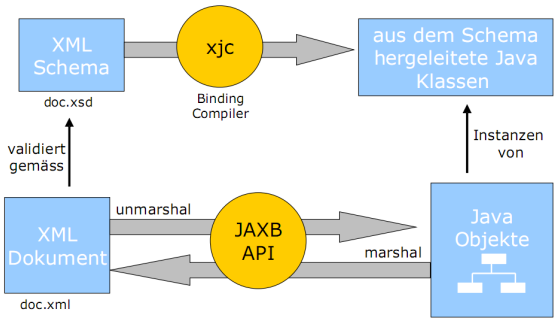


Abbildung 13.2: XML-zu-Objekt-Transformation mit dem JAXB Binding Framework [Vgl. Amr09, S.3].

Dies soll am Beispiel des in der Testfall-Steuerdatei verwendeten Knotens `<packetsToSendType>` verdeutlicht werden, der zusammen mit den verwendeten Datentypen in der Schemadatei definiert ist als:

```
1 <xsd:complexType name="packetsToSendType" >
2   <xsd:sequence>
3     <xsd:element name="timeOffset" type="tns:limitedTimeType" />
4     <xsd:element name="iterationCount" type="tns:retryCountType" minOccurs="0" />
5     <xsd:element name="iterationInterval" type="tns:limitedTimeType" minOccurs="0" />
6   </xsd:sequence>
7   <xsd:attribute name="id" type="xsd:unsignedShort" use="required" />
8 </xsd:complexType>
```

[154] Erzeugung eines Objektbaumes durch Deserialisierung aus einem XML-Dokument.

```
9
10 <xsd:simpleType name="limitedTimeType">
11   <xsd:restriction base="xsd:integer">
12     <xsd:minInclusive value="0" />
13     <xsd:maxInclusive value="86400" />
14   </xsd:restriction>
15 </xsd:simpleType>
16
17 <xsd:simpleType name="retryCountType">
18   <xsd:restriction base="xsd:byte">
19     <xsd:minInclusive value="0" />
20     <xsd:maxInclusive value="10" />
21   </xsd:restriction>
22 </xsd:simpleType>
```

Listing 13.1: Auszug aus der Testfall-Schemabeschreibung (Vgl. `testserver/config/testcase_withTns.xsd`).

Durch den Schemacompiler wird dabei folgende Klassendeklaration erzeugt:

```
1  @XmlAccessorType(XmlAccessType.FIELD)
2  @XmlType(name = "packetsToSendType", propOrder = {
3      "timeOffset",
4      "iterationCount",
5      "iterationInterval"
6  })
7  public class PacketsToSendType {
8
9      protected int timeOffset;
10     protected Byte iterationCount;
11     protected Integer iterationInterval;
12     @XmlAttribute(required = true)
13     @XmlSchemaType(name = "unsignedShort")
14     protected int id;
15
16     public int getTimeOffset() {
17         return timeOffset;
18     }
19     public void setTimeOffset(int value) {
20         this.timeOffset = value;
21     }
22     public Byte getIterationCount() {
23         return iterationCount;
24     }
25     public void setIterationCount(Byte value) {
26         this.iterationCount = value;
27     }
28     public Integer getIterationInterval() {
29         return iterationInterval;
30     }
31     public void setIterationInterval(Integer value) {
32         this.iterationInterval = value;
33     }
34     public int getId() {
35         return id;
36     }
37     public void setId(int value) {
38         this.id = value;
39     }
40 }
```

Listing 13.2: Auszug aus der durch den Schemacompiler erzeugten Klassendeklaration (Vgl. `testserver/src/jaxb/testcase/tns/PacketsToSendType.java`).

Die Datenbindung zur Laufzeit wird später anhand eines Beispiels erläutert.

13.2 Anbindung des Knotennetzes

Das Package *nodeNetwork* realisiert die Anbindung des Knotennetzes an den Testserver. Alle Zugriffe darauf werden durch die eine Singleton-Instanz der Klasse *NodeNetwork.java* gekapselt. Als Knotenarten werden „normale“ Knoten (*RegularNode.java*) und Steuerknoten (*ControllerNode.java*) unterstützt, die beide von der Basisklasse *Node.java* abgeleitet sind. Mit jedem Knoten kann eine Knotenverbindung assoziiert sein, die durch das Interface *NodeConnection.java* beschrieben wird. Deren Implementierung kann wahlweise seriell oder als Socket-Verbindung[155] erfolgen. Für den Fall der *SerialConnection* wird das RXTX Java Communication API [Vgl. Jar09] eingesetzt. Da Steuerknoten ebenfalls aggregierte Knoten besitzen können, muss aber nicht jeder Knoten über eine zugehörige Knotenverbindung verfügen. Zur Beschreibung potentiell zu verwendender Knotenanbindungen wird eine XML-Datei verwendet (Vgl. Listing 13.3).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tns:nodeConnections
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:tns="http://www.dresden-elektronik.de/testcases"
5     xsi:schemaLocation="http://www.dresden-elektronik.de/testcases nodeConnection.xsd"
          >
6
7     <comport>
8        <description>STB1</description>
9        <port>COM4</port>
10    </comport>
11
12    <socket>
13       <host>localhost</host>
14       <port>9876</port>
15    </socket>
16
17 </tns:nodeConnections>
```

Listing 13.3: Auszug aus einer Knotenverbindungs-Beschreibungsdatei (Vgl. `testserver/config/nodeConnection.xml`).

NodeNetwork.java stellt Funktionen zum Initialisieren, Reinitialisieren und Stoppen der Knoten und ihrer Verbindungen zur Verfügung. Bei der *Initialisierung des Knotennetzes* (`NodeNetwork.init()`) wird die eben beschriebene Datei geladen, validiert, geparst und in entsprechende Bindungsobjekte deserialisiert. Anschließend wird eine *NodeFactory* verwendet, um jede darin gefundene Knotenverbindung auf angeschlossene Knoten hin zu überprüfen und passende *Node*-Objekte anzulegen. Die Factory-Methode `getNodeForConnection()` versucht dabei die Verbindung zu öffnen und eine Reset-Nachricht darüber zu versenden. Mit einem Timeout wird auf eine Antwort gewartet, bei deren Eingang anhand der beinhalteten MAC-Adresse entschieden wird, ob sich hinter der Verbindung ein regulärer Knoten oder ein Steuerknoten verbirgt.

[155] Vgl. Steuerknoten.

Nachdem durch Übergabe der Knotenverbindung an den Konstruktor ein zum Knotentyp passendes Objekt angelegt wurde, beginnt dessen interne Initialisierung. Diese erfolgt durch einen Thread, der gleichzeitig einen Zustandsautomaten implementiert (Vgl. `Node.StateMachine`). Auf diese Art lassen sich mehrere Knoten parallel initialisieren. Zuerst werden die vom Knoten unterstützten Eigenschaften abgefragt und in einer Map gespeichert. Steuerknoten initialisieren danach an sie angeschlossene reguläre Knoten und aggregieren die zugehörigen Objekte ohne eine zusätzliche Knotenverbindung dafür anzulegen. Nach beendeter Initialisierung registrieren sich die Knoten selbst beim *NodeNetwork*, wo sie in einem Set abgelegt werden (`reportInitialized()`). Umgekehrt können sich Knoten selbst wieder beim *NodeNetwork* abmelden, beispielsweise bei Übertragungsfehlern (`reportFailure()`).

Sollen während des Serverbetriebs Knoten zum Netz hinzugefügt oder entfernt werden, lässt sich eine *Reinitialisierung* (Vgl. `reinit()`) durchführen. Dazu muss vorher die Verbindungsbeschreibungsdatei angepasst werden. Die Reinitialisierung kann *partiell* erfolgen (nur neue Knoten hinzufügen), was jederzeit gefahrlos möglich ist. Eine *vollständige* Reinitialisierung ist jedoch nur dann erlaubt, wenn keine Testfälle laufen. Dabei werden zuerst alle vorhandenen Knoten gestoppt und danach neue gemäß Verbindungsbeschreibungsdatei gestartet.

Über das *NodeNetwork* lassen sich ferner die Verfügbarkeit von Knoten anhand ihrer MAC-Adressen (`checkNodeAvailability()`) bzw. die Adressen aller Knoten (`getAllAddresses()`) erfragen. Zu einem durch seine MAC-Adresse gegebenen Knoten lassen sich – bspw. bevor ein Testfall ausgeführt wird – die von ihm unterstützten Eigenschaften ermitteln (`getNodeCapabilities()`) bzw. gezielt prüfen, ob der Knoten einen gegebenen Parameter mit dem gewünschten Wert unterstützen würde (`supportsCapability()`).

Soll mit Knoten kommuniziert werden, erfolgt dies wiederum über das *NodeNetwork*. Der Initiator der Verbindung muss das Interface *NodeNetworkClient* implementieren. Vor Beginn der Datenübertragung sind die beteiligten Knoten zunächst für die exklusive Nutzung zu blockieren. Dies erfolgt über die Methode `registerClient()`, wodurch der *NodeNetworkClient* gleichzeitig zur Senke für Antwortpakete des Knotens wird. Konnte wenigstens einer der erforderlichen Knoten nicht blockiert werden, schlägt die Registrierung fehl, alle bereits blockierten Knoten werden wieder freigegeben.

Pakete werden über Warteschlangen, die pro Kommunikationsrichtung einmal im *NodeNetwork* und in jedem *Node* vorhanden sind, ausgetauscht. Zur Einkettung von Paketen existieren eigene Methoden (Vgl. `NodeNetwork.enqueueForDelivery()`). Die Bearbeitung der Warteschlangen erfolgt im *NodeNetwork*

durch Threads (Vgl. `PacketDeliverer` bzw. `PacketForwarder`, in *Node*-Objekten innerhalb des Zustandsautomaten. Bevor ein Paket weitergeleitet werden kann, muss dessen Empfänger ermittelt werden. Bei der Übermittlung zum Knoten hin erfolgt dies im *NodeNetwork* anhand des zur MAC-Adresse zugeordneten Knotenobjekts (`getNodeForAddress()`), in der Gegenrichtung werden Pakete zunächst vom Knoten an das *NodeNetwork* weitergeleitet und dort der auf die MAC-Adresse registrierte *NodeNetworkClient* ermittelt, an den das Paket schließlich zugestellt wird. In Abbildung 13.3 sind Anmeldung, Versand eines Pakets, Empfang der Antwort und Weiterleitung zurück an den Client vereinfacht dargestellt. Zusammenfassend werden in Abbildung 13.4 alle im Package *nodeNetwork* vorhandenen Klassen und ihre Beziehungen nochmals dargestellt, wobei zur Übersichtlichkeit nur die relevanten Methoden angegeben wurden.

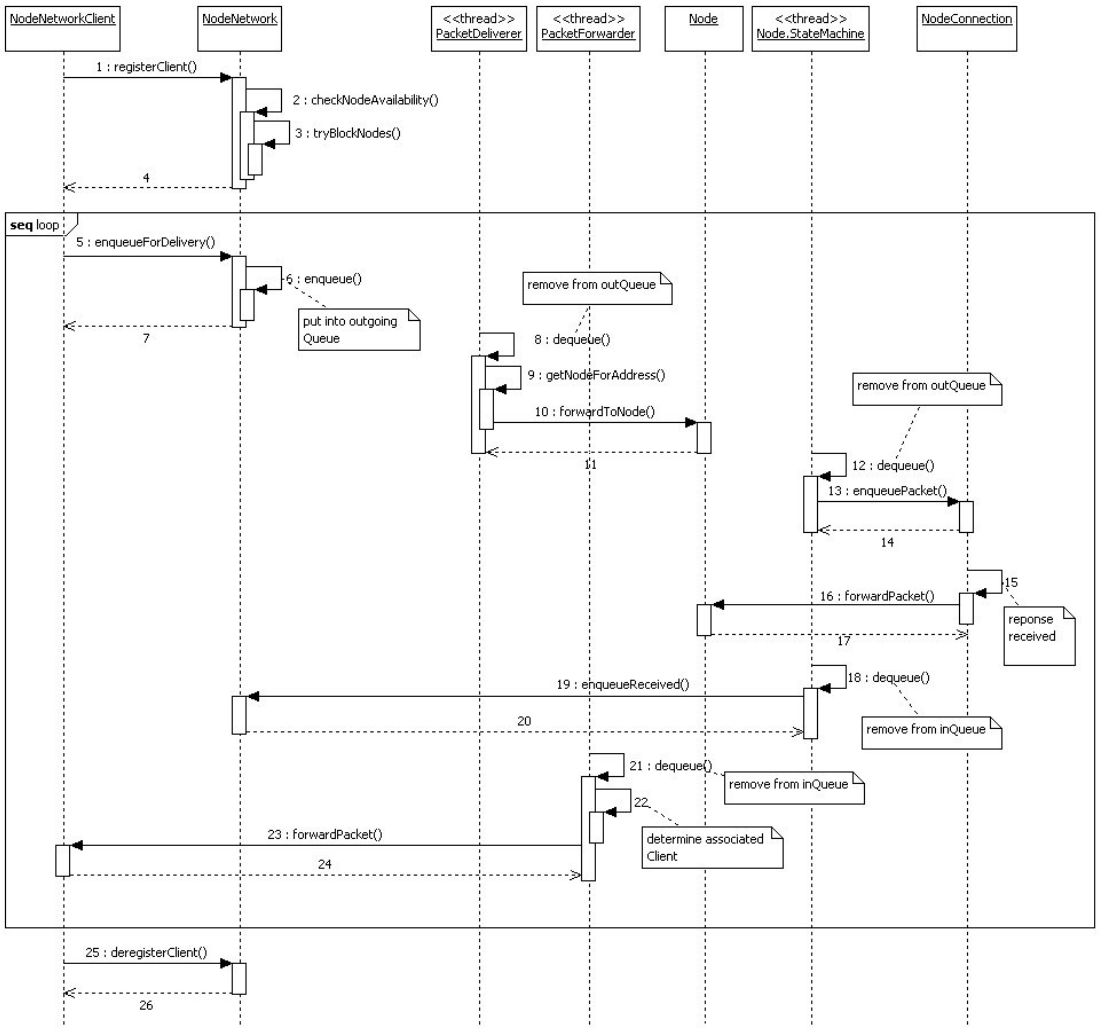


Abbildung 13.3: Interaktion zwischen NodeNetworkClient, NodeNetwork und Node.

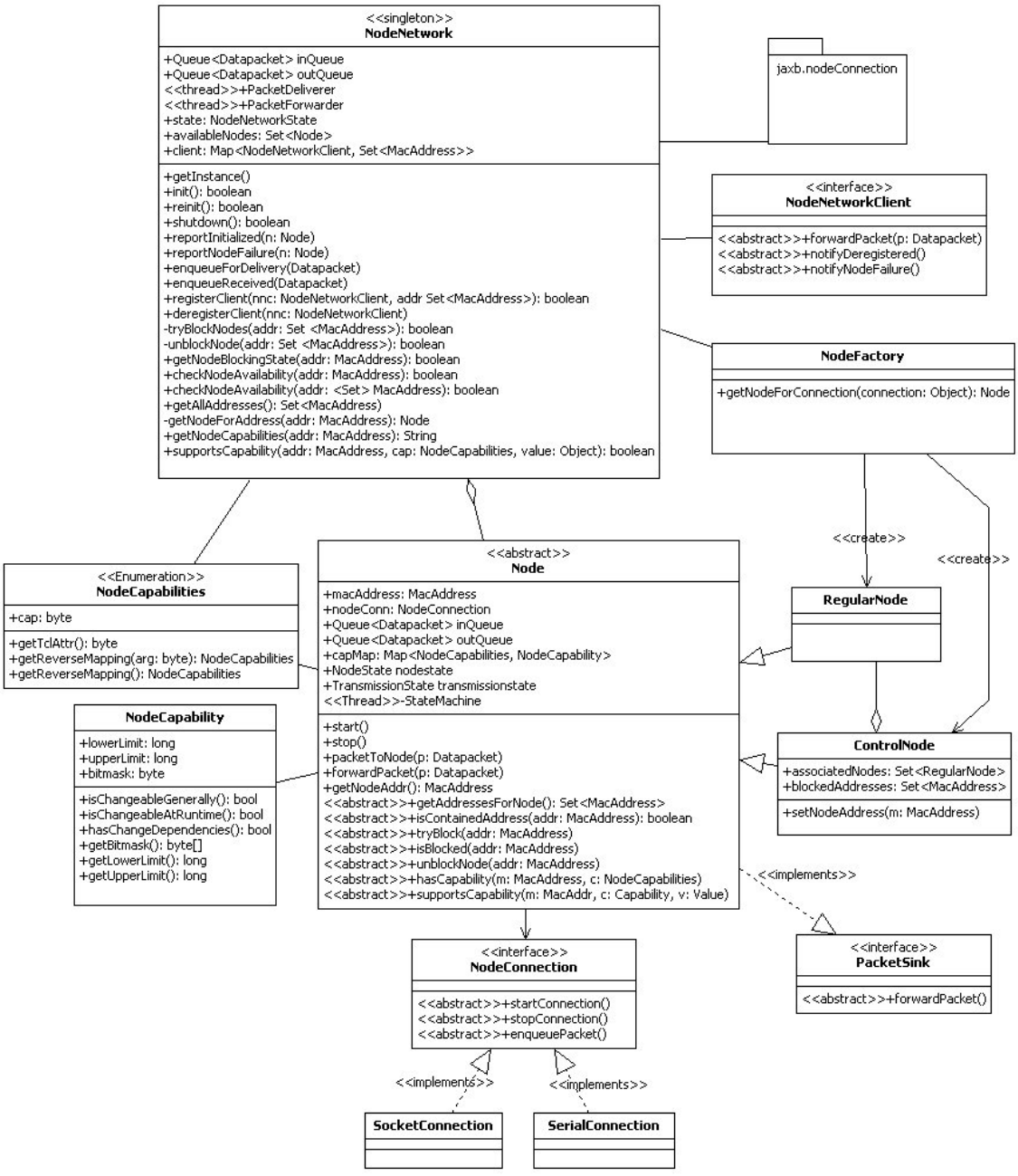


Abbildung 13.4: Reduziertes Klassendiagramm des Package *nodeNetwork*.

13.3 Nachrichtenbehandlung

Im Package *datapacket* befinden sich alle Klassen zur Nachrichtenbehandlung. Zentral ist die Klasse *Message.java*, in der das bekannte Paketformat zur Server-Knoten-Kommunikation abgebildet wird. Sie beinhaltet alle primitiven Typen, MAC-Adresse und Zeitstempel wurden in separate Klassen ausgelagert. Die Klasse *PreparedDatapacket.java* beschreibt Nachrichten, die in Verbindung mit einem Testfall zeitverzögert zu versenden sind. *Datapacket.java* kapselt zu übermittelnde Nachrichten nochmals, indem die Nachrichtenrichtung und ein lokal generierter Zeitstempel hinzugefügt werden, da die in den vom Knoten versendeten Paketen enthaltenen Zeitstempel ggf. ungenau sein können. Die Klasse *Commands.java* ist die Entsprechung zur `command.h` des Protokollstack[156]. *MyStackConst.java* kapselt verschiedene Konstanten, die für die semantische Interpretation von Nachrichten erforderlich sind. Diese wird durch die Klassen *PaketInterpreter.java* und *PacketFilter.java* realisiert, worauf später eingegangen wird.

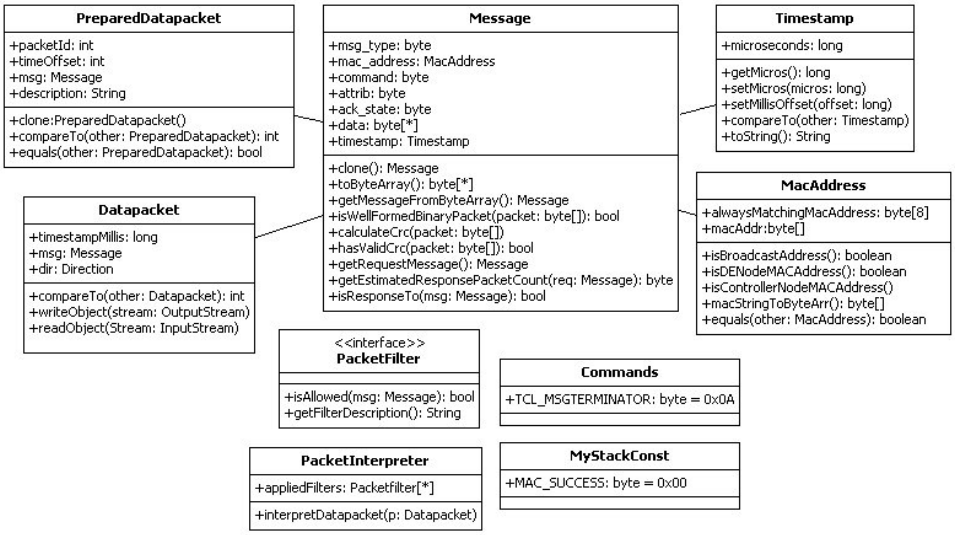


Abbildung 13.5: Reduziertes Klassendiagramm des Package *datapacket*.

Bereits in der *NodeConnection* werden Byteströme in Nachrichten zerlegt und in *Message*-Objekte konvertiert (`Message.getMessageFromByteArray()`). Zu versendende Nachrichten werden umgekehrt erst unmittelbar vor ihrem Versand serialisiert (`Message.toByteArray()`). In der Klasse *Message.java* existieren weiterhin Methoden zur Validierung und Generierung von Prüfsummen sowie mehrere statische Hilfsmethoden zur Erzeugung vorgefertigter Nachrichten, beispielsweise zur Abfrage eines Knotenparameteters (`getRequestMessage()`). Um feststellen zu können, ob eine eingehende Nachricht eine Antwort auf die zuvor gesendete Anfrage darstellt, gibt es Methoden wie `isResponseTo()`.

[156] Darin sind alle gültigen Belegungen der Nachrichtenfelder definiert.

13.4 Verwaltung von Testfällen

Sämtliche Testfälle betreffende Funktionalität wird im Package *testcases* zusammengefasst. Die Abbildung 13.6 beinhaltet eine vereinfachte Darstellung der darin enthaltenen Klassen und ihren Assoziationen.

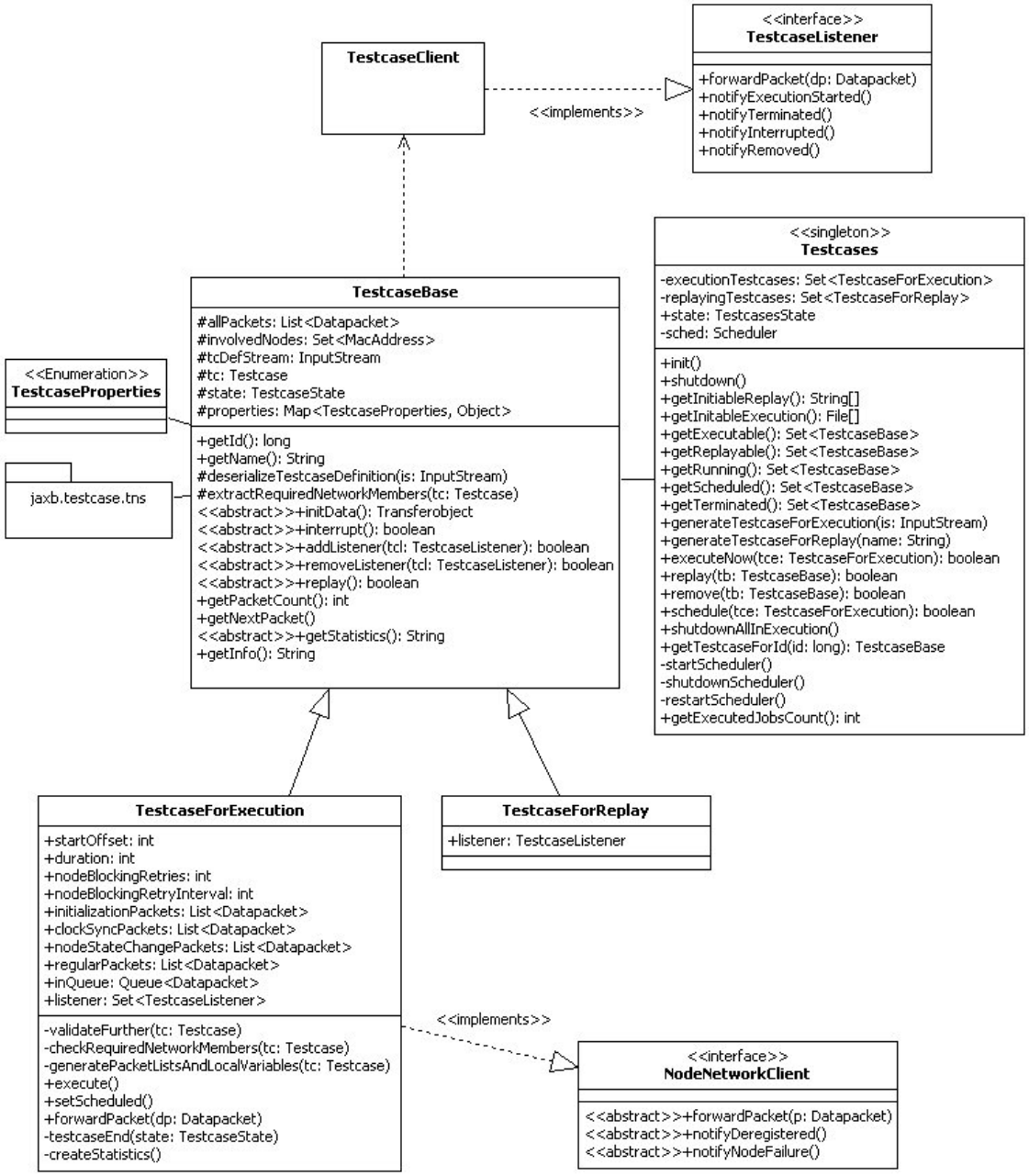


Abbildung 13.6: Klassendiagramm des Package *testcases*.

Analog zum Knotennetz existiert hier die eine als Singleton implementierte Verwaltungsklasse: *Testcases.java* kapselt ausführbare Testfälle (*TestcaseForExecution.java*) bzw. nochmals abzuspielende Testfälle (*TestcaseForReplay.java*). Dazu stellt sie zwei Factory-Methoden bereit, mit der sich Testfälle anlegen lassen: `generateTestcaseForExecution()` erstellt ein Testfallobjekt aus einer Testfall-Beschreibung, `generateTestcaseForReplay()` lädt einen Testfall aus

der Persistenzschicht. So erzeugte Testfallobjekte werden – getrennt nach Typ – in jeweils einem Set gespeichert. Einmal angelegt, können auszuführende Testfälle mit `executeNow()` gestartet werden. Wiederzugebende Testfälle und auszuführende Testfälle, die beendet wurden, lassen sich mit `replay()` nochmals abspielen, `remove()` entfernt sie wieder. Vom Zustand eines Testfallobjektes ist es abhängig, welche Aktionen damit möglich sind. Mittels der Methoden `getReplayable()`, `getExecutable()`, `getRunning()` oder `getTerminated()` lassen sich vorhandene Testfälle nach Zuständen kategorisiert erfragen.

In Verbindung mit dem Anlegen eines Testfalles wird er initialisiert (Vgl. `initData()`), wobei dies abhängig von dessen Typ unterschiedlich erfolgt. Bei einem *auszuführenden Testfall* wird die Testfallbeschreibung über einen InputStream an den Konstruktor übergeben, welcher anschließend in Bindungsobjekte deserialisiert wird (Vgl. Listing 13.4). Sind bislang keine Fehler aufgetreten, werden nun die Adressen der für den Testfall erforderlichen Knoten extrahiert (`extractRequiredNetworkMembers()`) und basierend auf den angegebenen Paketen und Ein- bzw. Ausschaltzeitpunkten mit `generatePacketListsAndLocalVariables()` temporäre Paketlisten erzeugt.

```
1  protected static Transferobject deserializeTestcaseDefinition(InputStream is)
2  {
3     Testcase tc = null;
4     File xsdFile = new File("config/testcase_withTns.xsd");
5
6     try {
7        JAXBContext jc = JAXBContext.newInstance(Testcase.class);
8        Unmarshaller u = jc.createUnmarshaller();
9        SchemaFactory schemaFactory =
10          SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
11       Schema schema = schemaFactory.newSchema(xsdFile);
12       u.setSchema(schema);
13       tc=(Testcase)u.unmarshal(is);
14    }
15
16    catch (Exception e) {
17       return new Transferobject(true, "JAXB failed " + e.getMessage(), null );
18    }
19    // all fine
20    return new Transferobject(false, null, tc);
21 }
```

Listing 13.4: Deserialisierung der Testfallbeschreibung (Vgl. `testserver/testcase/TestcaseBase.java`).

Im Zuge der Initialisierung wird ebenfalls eine Referenz auf die Persistierungsschicht angefordert und die Testfallbeschreibung darin abgelegt. Weiterhin wird eine Map erzeugt (`properties`), in der später sukzessive ausgewählte Eigenschaften des Testfalles gespeichert werden[157].

Die Abarbeitung eines Testfalles erfolgt durch die Methode `execute()`, welche durch die *Testcases*-Kapsel aufgerufen wird. Bei der Ausführung wird zunächst

[157] Zeitstempel für Start und Beendigung der Testfallausführung, Ende-Status sowie Paketstatistiken.

erfragt, ob der Testfall bereits gestartet wurde. War dies nicht der Fall, erfolgt die weitere Verarbeitung innerhalb eines Zustandsautomaten. Darin wird zuerst die Starzeit erfasst, danach überprüft, ob im Knotennetz die erforderlichen Knoten vorhanden sind und ob sie die vom Testfall gewünschten Eigenschaften bereitstellen (Vgl. `checkRequiredNetworkMembers()`). Anschließend wird versucht, den Testfall als *NodeNetworkClient* zu registrieren. Sollte dies nicht auf Anhieb funktionieren, weil bspw. gerade ein anderer Testfall einen benötigten Knoten blockiert, erfolgt nach einer Wartezeit ein erneuter Versuch. Dies geschieht solange, bis entweder alle Knoten blockiert werden konnten oder die maximale Anzahl erlaubter Wiederholungen abgelaufen ist.

Unter der Annahme, dass die Registrierung erfolgreich war, werden nun alle Knoten parametriert, im Anschluss erfolgt die Synchronisation der Knotenuhren[158]. Ist auch dies geschehen, beginnt die wirkliche Durchführung des Testfalles wozu zwei Timer gestartet werden. Der erste versendet während des Testfalles Nachrichten an die Knoten, der andere schaltet Knoten entsprechend ihres Einschaltzeitpunktes in den Zustand, in dem die Ausführung der Knotenapplikation beginnt. Beide Timer verwenden dazu die vorab erstellten, aus *PreparedDatapackets* bestehenden Paketlisten, die bereits nach Versandzeit sortiert vorliegen, und versenden ein Paket genau dann, wenn dessen Zeitstempel mit der aktuellen Relativzeit übereinstimmt. Als Bezugspunkt dafür wird die Systemzeit zum Start des Testfalles bzw. nach erfolgter Uhrensynchronisation verwendet. Analog werden empfangene Pakete mit relativen Zeitstempeln versehen.

Nachdem der Testfall die gewünschte Zeit gelaufen ist, wechselt er in einen Ende-Zustand, woraus `testcaseEnd()` gerufen wird. Die gleiche Methode wird aufgerufen, wenn der Testfall zwischenzeitlich unterbrochen werden soll (`interrupt()`) oder innerhalb der Ausführung des Zustandsautomaten Fehler auftraten, bspw. weil Knoten nicht verfügbar waren. Darin erfolgt eine Beendigung des Testfalles, die je nach Automatenzustand, aus dem die Methode aufgerufen wird, unterschiedlich verläuft. Wurde der Testfall zumindest gestartet, werden hierin alle Knoten zurückgesetzt und eine Deregistrierung gegenüber dem *NodeNetwork* vorgenommen. Abschließend wird der Testfall persistiert, wobei die Liste übertragener Pakete sowie die `properties`-Map gesichert werden. Er bleibt aber in der Testcases-Kapsel bestehen und lässt sich anschließend nochmals wiedergeben. Die verschiedenen Zustände und mögliche Übergänge werden nochmals in der Abbildung 13.7 dargestellt.

Bei der *Wiederherstellung eines bereits persistierten Testfalles*, die ebenfalls durch `initData()` erfolgt, wird die Testfallbeschreibung aus der Persistenzschicht ge-

[158] Vgl. Zustandsautomat der Knoten.

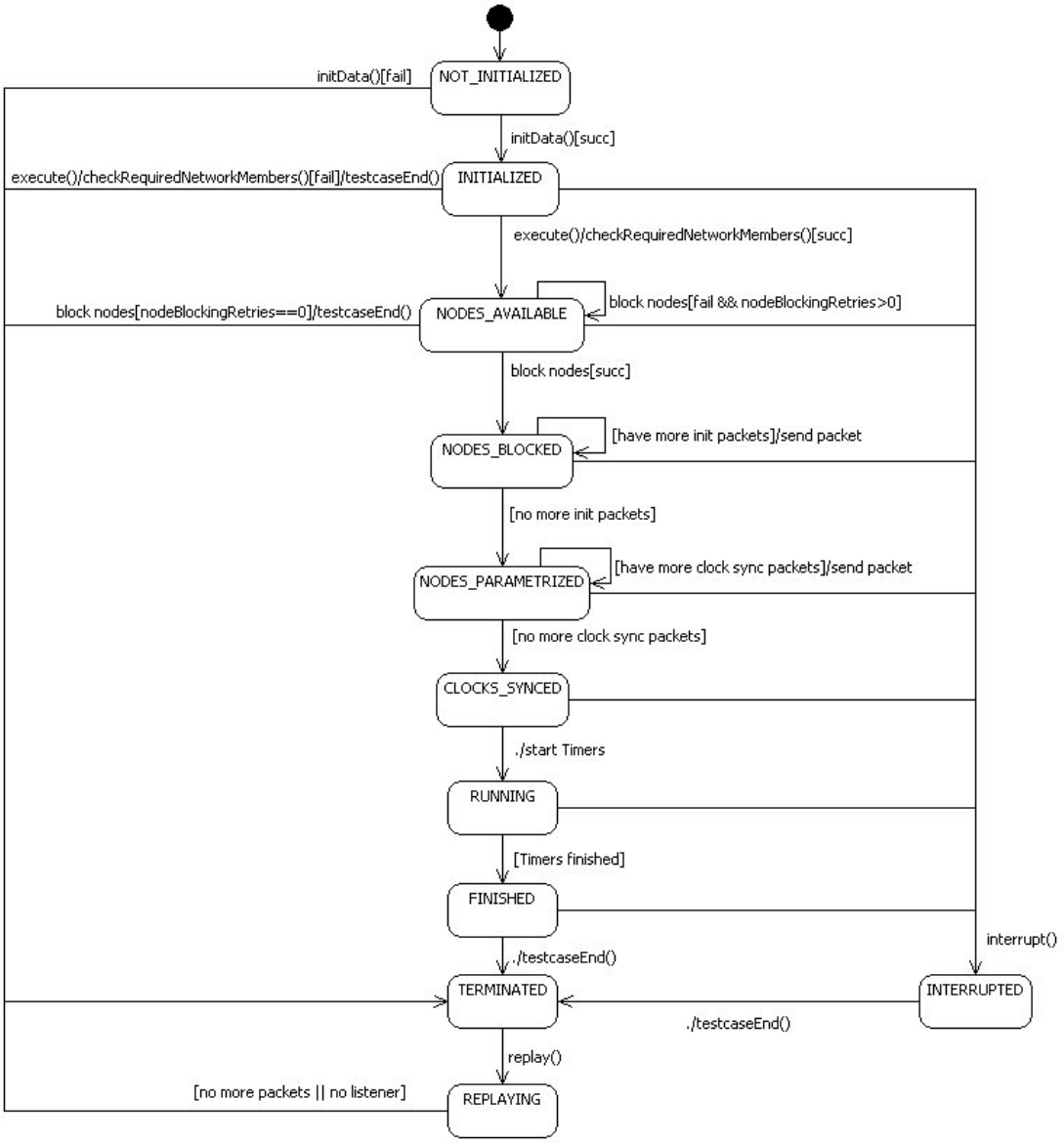


Abbildung 13.7: Mögliche Zustände und Zustandsübergänge eines ausführbaren Testfalles.

laden. Die Deserialisierung und die Ermittlung der beteiligten Netzknoten erfolgt analog zum *TestcaseForExecution*. Listen für zu sendende Pakete werden jedoch nicht aufgebaut, stattdessen sämtliche während der ursprünglichen Ausführung übermittelten Pakete wiederhergestellt. Start- und Endezeit, letzter Zustand des Zustandsautomaten sowie Statistiken werden durch die Map `properties` abgebildet und ebenfalls geladen.

Normalerweise wird ein Testfall ohne jegliche Ausgaben abgearbeitet. Vor seiner Ausführung oder Wiedergabe können jedoch *TestcaseListener* registriert werden, die dann sämtlichen Paketverkehr als Kopie zugestellt bekommen und über Zustandswechsel des Testfalles informiert werden. Endet beispielsweise ein Testfall, wird der Listener mit `notifyTerminated()` darüber in Kenntnis gesetzt. Ein registrierter *TestcaseListener* kann selbst entscheiden, wie mit den an ihn

weitergeleiteten Paketen zu verfahren ist, bspw. kann er deren Inhalt interpretieren. Bei Replay eines Testfalles werden die Pakete einzeln angefordert (Vgl. `getNextPacket()`). Damit dies möglich ist, kann bei einem Replay nur ein TestcaseListener an einen Testfall gebunden werden. Bevor er durch Registrierung eines weiteren Listeners ersetzt wird, erfolgt eine Meldung `notifyRemoved()`.

Nur ansatzweise implementiert wurde ein Schedulingmechanismus, um Testfälle zeitverzögert ausführen zu können. Dazu wurde der Quartz-Scheduler [Vgl. Ter09] verwendet, der als JAR[159] in den Klassenpfad des Servers einzubinden ist. Testfälle werden dabei auf auszuführende Tasks abgebildet, die sich gruppieren lassen. Damit Testfälle auch einen Neustart des Testservers überstehen, unterstützt der Quartz-Scheduler die Zwischenspeicherung von Tasks in einer Datenbank. Der Scheduler wird dabei ebenfalls durch *Testcases* verwaltet.

13.5 Zentrales Logging

Zur zentralen Behandlung von zur Laufzeit anfallenden Meldungen wird die Klasse *MyLogger.java* aus dem Package *logging* verwendet. Durch deren Factory-Methode `getInstance()` wird das Entwurfsmuster Singleton realisiert. Mittels der Methode `reportMsg()` lassen sich Meldungen unterschiedlicher Art (Information, Warnung, Fehler, etc.) übergeben, wobei jede Meldungsstufe individuell behandelt wird. So kann die Ausgabe auf der Konsole erfolgen und/oder in Dateien umgelenkt werden.

13.6 Persistierung von Testfällen

Das Package *persistence* dient der Persistierung von Testfällen. Es umfasst derzeit drei Klassen (Vgl. Abb. 13.8). Durch das Interface *TestcasePersistence* wird eine Schnittstelle vorgegeben, die Persistierungsmechanismen implementieren müssen. Derzeit wird nur Dateisystem-basiert gearbeitet, durch andere implementierende Klassen, kann aber später einfach eine Datenbank-Anbindung realisiert werden. Die Klasse *PersistenceFactory* hat zwei getrennte Aufgaben. Einerseits erzeugt sie ein zur aktuellen Implementierung passendes Persistierungsobjekt, mit dem Testfälle gespeichert werden können (Vgl. *LogFilePersistence*), andererseits ermittelt sie existierende Testfälle und delegiert Methoden zu ihrer Wiederherstellung an die verwendete Persistierungsart. Sowohl zum Speichern als auch zur Wiederherstellung von Testfällen werden drei unterschiedliche Methoden verwendet, mit denen Testfallbeschreibung, Paketlisten und Testfalleigenschaften getrennt behandelt werden. In der derzeitigen Implementierung werden

[159] Java Archive.

dazu drei Dateien mit einem gemeinsamen Präfix verwendet, bei einer späteren Datenbank-basierten Implementierung widerspiegelt dies die Trennung in unterschiedliche Tabellen.

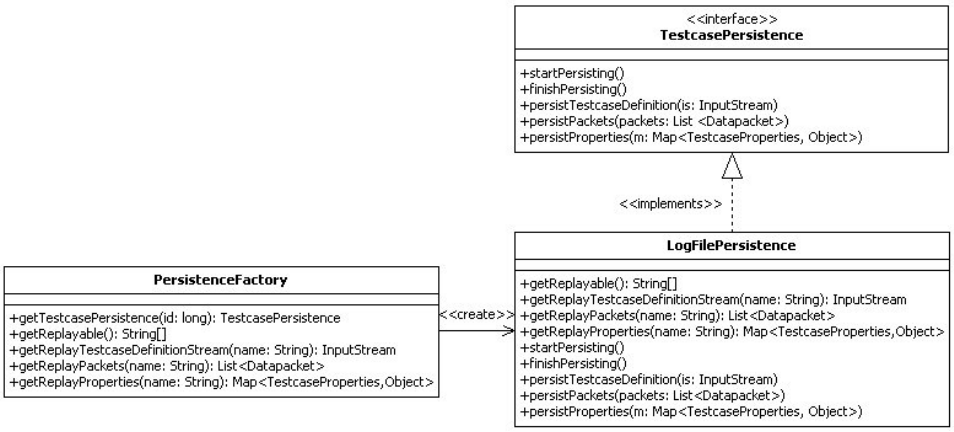


Abbildung 13.8: Klassendiagramm des Package *persistence*.

13.7 Interpretation von Nachrichten

Während der Testfallausführung werden sämtliche übertragene Pakete aufgezeichnet, wozu Objekte der Klasse *Datapacket* verwendet werden. Um ihre Semantik zu ermitteln, wurde ein Interpreter entwickelt, der das verwendete Binärformat in eine textuelle Form überführt. Die Paketinterpretation kann dabei wahlweise zur Laufzeit oder während der späteren Wiedergabe eines Testfalles erfolgen. Sie wird in *PacketInterpreter* gekapselt - der mit Abstand größten Klasse des Testservers. Deren Methode `interpretDatapacket()` hat neben der semantischen Interpretation die Aufgabe, Pakete zu validieren. Dabei wird einerseits geprüft, ob der Nachrichtentyp zur -richtung passt, andererseits, ob die Nachrichtenfelder untereinander eine gültige Kombination bilden, weshalb der Interpreter auch zur Vorvalidierung von Datenpaketen vor ihrem Versand genutzt werden kann. Auf den ersten Blick gehören Validierung und Interpretation nicht direkt zusammen, da aber die Interpretation valide Pakete voraussetzt – und um redundanten Code zu vermeiden – , wurden beide Funktionen in einer Methode zusammengefasst. Die Methode `interpretDatapacket()` liefert ein `InterpretResult`-Objekt zurück, das neben der textuellen Repräsentation des Interpretationsergebnisses einen Fehlercode beinhaltet (Vgl. Listing 13.5).

```
1  /** returncode of packet validator/interpreter */
2  public enum INTERPRET_STATUS {
3    ALL_FINE,           // everything went fine
4    NO_CONTENT,         // datapacket has no content
5    ILLEGAL_DIRECTION,  // flag combination doesn't match packet direction
6    FLAG_ERR,           // flags (Type, Cmd, Attr) dont go together
7    LEN_ERR,            // data fields length doesn't match command
8    UNKNOWN_CMD,        // flag combination unknown/handler not yet implemented
9    FILTERED_OUT        // interpretation has been surpressed by a message filter
10 };
11
12
13 /** returncode and result of datapacket interpretation */
14 public class InterpretResult
15 {
16   INTERPRET_STATUS retCode;
17   String interpretation;
18
19   public InterpretResult(INTERPRET_STATUS is, String s)
20   {
21     retCode = is;
22     interpretation = s;
23   }
24
25   public String getInterpretation()
26   {   return interpretation;
27   }
28
29   public INTERPRET_STATUS getErrorCode()
30   {   return retCode;
31   }
32 }
```

Listing 13.5: Interpretationsergebnis und mögliche Fehlercodes (Vgl. `testserver/datapacket/PacketInterpreter.java`).

Die Interpretation eines Datenpakets erfolgt feldweise, wobei das jeweilige Interpretationsergebnis an einen `StringBuffer` angefügt wird. Sie endet bei Auftreten eines Fehlers oder Erreichen des Paketendes. Zu Beginn werden Nachrichtenrichtung und Zeitstempel in eine menschenlesbare Form gebracht. Es folgen alle Felder des Nachrichtenformats gemäß der in Tabelle 9.1 angegegebenen Reihenfolge mit Ausnahme, dass MAC-Adresse und Nachrichtentyp vertauscht wurden. Bis einschließlich des ACK-Feldes beschränkt sich der Interpretationsvorgang auf die Ausgabe des Feldnamens bzw. die Validierung des aktuellen Feldes gegenüber den vorigen.

Besonders interessant ist erst das Datenfeld, da Knoten bei der Erzeugung von Nachrichten in Callback-Funktionen verschiedene Variablen und Datenstrukturen hintereinander darin ablegen. Um diese zu interpretieren werden – aus Gründen der Übersichtlichkeit und der Mehrfachverwendung – Hilfsmethoden verwendet. Beispielsweise ist im Datenfeld aller Nachrichten, die bei Confirmations erzeugt werden, ein Statusfeld enthalten, welches mit der Methode `getIEEE-Status()` bearbeitet wird (Vgl. 13.6). Sämtliche zur Interpretation von Datenstrukturen erforderlichen Konstantendefinitionen wurden aus mehreren Header-Dateien des Stack entnommen und in der Klasse *MyStackConst.java* zusammengestellt.

```
1  private static String getIEEEStatus(byte status)
2  {
3    // @see: ieee_const.h
4    switch(status)
5    {
6      case MyStackConst.MAC_SUCCESS:              return "MAC_SUCCESS                ";
7      case MyStackConst.MAC_COUNTER_ERROR:        return "MAC_COUNTER_ERROR          ";
8      case MyStackConst.MAC_BEACON_LOSS:          return "MAC_BEACON_LOSS            ";
9      case MyStackConst.MAC_DENIED:               return "MAC_DENIED                 ";
10
11     ...
12
13     default:                                    return String.format("%02x", status);
14   }
15 }
```

Listing 13.6: Interpretation des IEEE-Statuscodes (Vgl. `testserver/datapacket/PacketInterpreter.java`).

Aber auch komplexe Datenstrukturen können im Datenfeld enthalten sein. Oft verwendet wird der `wpan_addr_spec_t`, der durch die Methode `interpretWpanAddrSpecT()` decodiert wird. Dabei werden rekursiv weitere interpretierende Methoden aufgerufen oder Hilfsmethoden der Klasse *common/Helper.java* zur formatierten Feldausgabe genutzt (Vgl. Listing 13.8). Zu beachten ist hierbei, dass die Elemente der Datenstrukturen Little-Endian-codiert sind und daher Byte-weise invertiert ausgegeben werden müssen.

```
1  private static String interpretWpanAddrSpecT(Message msg, int offsetInDataField)
2  {
3    StringBuilder sb = new StringBuilder();
4
5    //  @see: mac_api.h
6    //  typedef struct
7    //  wpan_addr_spec_tag {
8    //
9    //  // Address mode
10   //  (@ref WPAN_ADDRMODE_NONE, @ref WPAN_ADDRMODE_SHORT, or @ref WPAN_ADDRMODE_LONG)
11   //  uint8_t AddrMode;
12   //
13   //  uint16_t PANId;
14   //
15   //  // Device address. If AddrMode is @ref WPAN_ADDRMODE_SHORT,
16   //  // it is interpreted as 16 bit address.
17   //  uint64_t Addr;
18   //  } wpan_addr_spec_t;
19
20   if(msg.getData()==null)
21     return "";
22
23   byte addrMode = msg.getData()[0+offsetInDataField];
24   sb.append("addrMode: ");
25   sb.append(getAddrMode(addrMode));
26   sb.append(", PanID: ");
27   sb.append(getDatafieldFormatted(msg, 1+offsetInDataField, 2,true));
28   sb.append(", Addr: ");
29   sb.append(interpretAddrField(msg, 3+offsetInDataField, addrMode));
30
31   return sb.toString();
32 }
```

Listing 13.7: Interpretation der Datenstruktur wpan_addr_spec_t (Vgl. `testserver/datapacket/PacketInterpreter.java`).

Auf diese Weise wird aus der Binärdarstellung der Nachricht

```
03 00 04 25 FF FF 17 45 E5 42 07 7C 00 00 00 00 00 10 D0 FE 0E 16 00 02 FE CA 80 3C 0
    A 00 00 00 00 00
```

Listing 13.8: Binärdarstellung einer MAC Callback-Nachricht.

anhand Interpretation

```
>> [37] 03 00 04 25 FF FF 17 45 E5 42 07 7C 00 00 00 00 00 10 D0 FE 0E 16 00 02 FE CA
     00 00 00 00 00 00 00 00 80 3C 0A
 TS(DP):  +            8:266:000 [s:ms:us]
 TS(M):   +            8:128:322 [s:ms:us]
 diff:    +              138      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  REQ      ASSOC.                          DOCA
  Channel: 16 , ChannelPage: 00
  Coordinator: addrMode: shrt , PanID: CA FE , Addr: 00 00
  Capabilities: PAN Coord:0 FFD:0 mainsPwrd:0 RxOnWhenIdle:0 reqShrtAddr:1
```

Listing 13.9: Interpretation einer MAC Callback-Nachricht.

deutlich, dass der Knoten mit der MAC-Adresse `00:04:25:FF:FF:17:45:E5` sich an dem auf Kanal 22 vom Coordinator mit der Short-Adresse `0x0000` und der PAN-ID `0xCAFE` bereitgestellten PAN anmelden möchte, dabei eine Short-Adresse anfordert und das Paket 138 ms benötigte, um vom Knoten zum Server zu gelangen.

Die Interpretation von Datenfeldern, die MAC-Frames beinhalten, erfolgt durch die Methode `interpretFrameContent()`. Sie arbeitet den Frame von links beginnend ab. Wie aus Abbildung 13.9 ersichtlich ist, werden hier Felder variabler Breite verwendet. Zur Platzeinsparung wird beispielsweise eine Adresskompression eingesetzt, bei der durch ein Flag im FCF[160] des Frame (Vgl. Feld *PAN-ID Compression* in Abb. 13.10) bestimmt wird, dass PAN-ID von Absender und Empfänger identisch sind. Bei der Interpretation werden daher Hilfsobjekte angelegt, anhand deren sich die Decodierung der Folgefelder bestimmt.

Octets: 2	**1**	**0/2**	**0/2/8**	**0/2**	**0/2/8**	**0/5/6/10/ 14**	**variable**	**2**
Frame Control	Sequence Number	Destination PAN Identifier	Destination Address	Source PAN Identifier	Source Address	Auxiliary Security Header	Frame Payload	FCS
		Addressing fields						
MHR							MAC Payload	MFR

Abbildung 13.9: Format eines IEEE 802.15.4 MAC-Frame [Vgl. IEE06, S. 138].

[160] Frame Control Field.

Bits: 0–2	3	4	5	6	7–9	10–11	12–13	14–15
Frame Type	Security Enabled	Frame Pending	Ack. Request	PAN ID Compression	Reserved	Dest. Addressing Mode	Frame Version	Source Addressing Mode

Abbildung 13.10: Aufbau des Frame Control Feld eines IEEE 802.15.4 MAC-Frame [Vgl. IEE06, S. 138].

Aus der in Nachricht nach Listing 13.10 geht durch Aufbereitung (Vgl. Listing 13.11) schließlich hervor, dass ein *Beacon-Request*-Frame vom Knoten mit der Adresse `00:04:25:FF:FF:17:33:6B` versendet wurde.

```
1 05 00 04 25 FF FF 17 33 6B 6A 12 00 00 00 00 00 00 22 63 FE 0A 03 08 55 FF FF FF FF
      07 00 00
```

Listing 13.10: Binärdarstellung einer Frame Callback-Nachricht.

```
1  >> [33] 05 00 04 25 FF FF 17 33 6B 6A 12 00 00 00 00 00 00 22 63 FE 0A 03 08 55 FF FF
       FF FF 07 00 00 FF 0A
2   TS(DP):  +             0:109:000 [s:ms:us]
3   TS(M):   +             0:004:714 [s:ms:us]
4   diff:    +               105      [ms]
5   ## Src: 00:04:25:FF:FF:17:33:6B ##
6  FRAME    TX       MAC COMMAND FRAME             DOCA
7  03 08 55 FF FF FF FF 07 00 00
8  MHR:
9   FCF: type:C |sec:- |pend:- |ack:- |panIdCompr:-
10  addrModes:  dest:S src:N |version:2003 |seqNo:85
11  addressing:
12   dest  PanID:FF FF  addr:FF FF
13 MAC Payload:
14  Beacon Req
15 MFR: 0000 (unreliable)
```

Listing 13.11: Interpretation einer Frame-Callback-Nachricht.

Um in der Vielzahl von Nachrichten, die bei einem Testfall anfallen können, nicht den Überblick zu verlieren, können Pakete gefiltert werden. Innerhalb der Methode `interpretDatapacket()` wird daher zu Beginn mit `isFilteredOut()` eine Liste von Paketfiltern durchlaufen und die Interpretation mit Statuscode `FILTERED_OUT` sofort wieder verlassen, sobald der erste Filter ein Paket nicht akzeptiert. Paketfilter werden durch das Interface *PacketFilter.java* beschrieben und sind frei konfigurierbar. Beispielsweise können anhand des Nachrichtentyps Frames enthaltene Nachrichten sofort ausgeblendet werden, da diese den größten Anteil am Paketaufkommen haben, aber nicht immer benötigt werden. In der Paketfilterklasse sind dazu einige Beispielfilter definiert. Mit der Methode `addFilter()` lassen sie sich zu einer Paketinterpreter-Instanz hinzufügen, mittels `removeFilter()` wieder entfernen. Die Filterung ist jedoch nicht kontextsensitiv möglich, d.h. sie betrachtet jeweils nur die aktuelle Nachricht und nicht deren „Vorgeschichte“.

Neben der Interpretation einzelner Pakete lassen sich anhand von Paketsequenzen Ereignisse detektieren. Im IEEE 802.15.4-Standard wird dazu anhand von Sequenzdiagrammen erläutert, auf welche Anfragen (Request) welche Antworten (Confirmation) zu erwarten sind. Ein Beispiel dazu wurde bereits in Abb. 11.5 angegeben. Problematisch sind in diesem Zusammenhang folgende Punkte:

- Diese Paketsequenzen müssen nicht in der gegebenen Reihenfolge eingehen. Es können zwischendurch andere Nachrichten eingeschoben worden sein.
- Es können sich mehrere Paketsequenzen, welche zur Detektion eines Ereignisses benötigt werden, gegenseitig überlappen.
- Durch falsche Zeitstempel lassen sich Pakete nicht Knoten-übergreifend korrelieren. Aufgrund unterschiedlicher Übertragungslatenzen kann aber auch keine verlässliche Zeitstempelung im Server vorgenommen werden.
- Die Adressierung der Pakete erfolgt nicht einheitlich – es werden teilweise MAC-Adressen genutzt, teilweise aber auch Short-Adressen. Zur gegenseitigen Auflösung ist eine Mapping-Tabelle erforderlich, die während der Auswertung erst aufzubauen ist.
- Ein ggf. gesetzter Paketfilter kann Nachrichten unterdrücken, die zur Erkennung eines Ereignisses erforderlich wären.
- Die Erkennung ist abhängig vom verwendeten Protokoll bzw. der höchsten verwendeten Schicht nach Schichtenmodell. Eine Erkennung auf IEEE 802.15.4 MAC-Ebene ist bei Verwendung von ZigBee nur begrenzt aussagekräftig. Bei Einsatz proprietärer Protokolle trifft dies ebenfalls zu.

14 Clients

Da Clients für die Ausführung von Testfällen nicht unmittelbar erforderlich sind, wurde lediglich eine Kommandozeilen-basierte Benutzerschnittstelle für den Testserver entwickelt (Vgl. Abb. 14.1). Sie ermöglicht einen wesentlichen Teil der Clientfunktionen. Durch Eingabe der in eckigen Klammern angegebenen Zeichen bzw. Zeichenkombinationen wird in ein entsprechendes Untermenü verzweigt und interaktiv das weitere Vorgehen erfragt. Aus Platzgründen kann an dieser Stelle nicht weiter darauf eingegangen werden.

```
+--- main menu ---------------------------------+
| [t]estcase control                            |
|  + [l]ist, [p]repare, e[x]ecute, [r]eplay     |
|    [i]nterrupt, re[m]ove, [s]tatistics        |
| [n]ode network control                        |
|  + [i]nit,[r]einit+[p|f],[s]hutdown,          |
|    [a]ccessible Nodes,[b]lock,[c]apabilities  |
|    [l]owlevel node access                     |
| [l]oglevel change                             |
|  + [I|E|W|P|F|S] + [0|1]                      |
| [m]isc commands                               |
| [q]uit testserver                             |
+-----------------------------------------------+
```

Abbildung 14.1: Oberfläche des kommandozeilenbasierten Client.

15 Persistierung

Um die in Zusammenhang mit Testfällen anfallenden Daten zu speichern, ist die Verwendung einer Datenbank wünschenswert. Hier gilt sinngemäß das gleiche wie für Clients – zu Demonstrationszwecken wurde lediglich eine Objekt-(De-)Serialisierung implementiert. Zur besseren Übersichtlichkeit könnten die Testfälle hierarchisch einer Kategorie „Tests“ und diese wiederum „Versuchen“ zugeordnet werden. Ebenfalls ließen sich „Versuchsverantwortliche“ oder „Projekte“ zur Einordnung eines Testfalles definieren. Im Datenbankschema widerspiegeln könnte sich die bereits bei Testfällen vorgenommene Unterteilung: In einer separaten Tabelle für Testfälle bilden die Testfalleigenschaften deren Einträge, die Testfallbeschreibung wird lediglich als BLOB[161] referenziert. Zur Ablage der Knotenfirmware – ebenfalls als BLOB – wird die Verwendung einer separaten Tabelle vorgeschlagen. Da Paketlisten sehr umfangreich ausfallen können, sollten diese pro Testfall in jeweils einer eigenen Tabelle gespeichert werden.

[161] Binary Large Object.

16 Durchführung und Auswertung eines Testfalles

An dieser Stelle soll anhand eines konkreten Beispiels die Interaktion des Testservers mit den Funkknoten während der Durchführung eines Testfalles beschrieben werden. Auf sämtlichen Knoten des Knotennetzes ist dazu die im Kapitel 11.3 beschriebene Firmware aufgespielt. Durch die Testfallbeschreibung, die sich nochmals vollständig im Anhang A befindet, werden daraus 2 Knoten selektiert:

```
15      <requiredNetworkMember>
16         <description>PAN-Coordinator</description>
17         <addr>00-04-25-FF-FF-17-33-6B</addr>
18         <type>
19            <regularNode>
20               <role>coordinator</role>
21            </regularNode>
22         </type>
23      </requiredNetworkMember>
24
25      <requiredNetworkMember>
26         <description>Sensornode</description>
27         <addr>00-04-25-FF-FF-17-45-E5</addr>
28         <type>
29            <regularNode>
30               <role>end-device</role>
31            </regularNode>
32         </type>
33      </requiredNetworkMember>
```

Listing 16.1: Auszug aus der Testfall-Steuerdatei für den Beispieldurchlauf (1).

Simuliert werden soll hier das Verhalten des End-Device bei Störungen des Coordinators, der dazu während des Testfalles erst verzögert zu- und vorzeitig wieder abgeschaltet wird:

```
58      <nodeStateChange>
59         <nodeAddress>00-04-25-FF-FF-17-33-6B</nodeAddress>
60         <timeOffset>20</timeOffset>
61         <newState>on</newState>
62      </nodeStateChange>
63      <nodeStateChange>
64         <nodeAddress>00-04-25-FF-FF-17-33-6B</nodeAddress>
65         <timeOffset>40</timeOffset>
66         <newState>off</newState>
67      </nodeStateChange>
```

Listing 16.2: Auszug aus der Testfall-Steuerdatei für den Beispieldurchlauf (2).

Nachdem der Testfall gemäß den Erläuterungen in Kapitel 13.4 im Testserver angelegt wurde, kann die Ausführung beginnen. Der Testfall soll hier erst vollständig abgearbeitet und anschließend dessen Auswertung vorgenommen werden, die anhand von Auszügen im Folgenden dargestellt ist.

Zu Beginn werden beide Knoten zurückgesetzt (Vgl. Listing 16.3, Pakete 1, 2, dargestellt ist nur ein Knoten), danach startet die Parametrierung mit dem Setzen des Kanals (Pakete 3,5). Gut erkennbar ist das stop&wait-Verhalten – auf jede Anfrage erfolgt unmittelbar eine Antwort. Das zwischendurch eingehende Paket 4 ist

eine Statusmeldung, die nach dem Reset automatisch versendet wurde. Schließlich wird die Ausführung der Knotenapplikation gestartet (Pakete 23, 24).

```
packet 1/156
<< [23] 00 00 04 25 FF FF 17 33 6B 00 00 00 00 00 00 00 04 AA FE 00 3E 0A
 TS(DP):  -             1:906:000 [s:ms:us]
 TS(M):   +             0:000:000 [s:ms:us]
 diff:    +              1906      [ms]
 ## Dst: 00:04:25:FF:FF:17:33:6B ##
REQ      STATECHG Resetted                      DOCA
---------------------------------
packet 2/156
>> [23] 01 00 04 25 FF FF 17 33 6B 00 00 00 00 00 00 00 04 AA 01 00 C0 0A
 TS(DP):  -             1:875:000 [s:ms:us]
 TS(M):   +             0:000:000 [s:ms:us]
 diff:    +              1875      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
RSP      STATECHG Resetted                      ACK
---------------------------------
packet 3/156
<< [24] 00 00 04 25 FF FF 17 33 6B 00 00 00 00 00 00 00 02 18 FE 01 16 9D 0A
 TS(DP):  -             1:859:000 [s:ms:us]
 TS(M):   +             0:000:000 [s:ms:us]
 diff:    +              1859      [ms]
 ## Dst: 00:04:25:FF:FF:17:33:6B ##
REQ      SET      runtime Channel               DOCA
  new value: [16 ]
---------------------------------
packet 4/156
>> [23] 04 00 04 25 FF FF 17 33 6B 00 00 00 00 00 00 00 06 FF FE 00 6D 0A
 TS(DP):  -             1:016:000 [s:ms:us]
 TS(M):   +             0:000:000 [s:ms:us]
 diff:    +              1016      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
STATUS   UNKNOWN                                DOCA
---------------------------------
packet 5/156
>> [23] 01 00 04 25 FF FF 17 33 6B 00 00 00 00 00 00 00 02 18 01 00 74 0A
 TS(DP):  -             1:000:000 [s:ms:us]
 TS(M):   +             0:000:000 [s:ms:us]
 diff:    +              1000      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
RSP      SET      runtime Channel               ACK
---------------------------------
...
---------------------------------
packet 23/156
<< [23] 00 00 04 25 FF FF 17 45 E5 30 6F 01 00 00 00 00 00 04 AD FE 00 9F 0A
 TS(DP):  +             0:094:000 [s:ms:us]
 TS(M):   +             0:094:000 [s:ms:us]
 diff:    +                 0      [ms]
 ## Dst: 00:04:25:FF:FF:17:45:E5 ##
REQ      STATECHG TcStart                       DOCA
---------------------------------
packet 24/156
>> [23] 01 00 04 25 FF FF 17 45 E5 F2 B5 00 00 00 00 00 00 04 AD 01 00 78 0A
 TS(DP):  +             0:125:000 [s:ms:us]
 TS(M):   +             0:046:578 [s:ms:us]
 diff:    +               -79      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
RSP      STATECHG TcStart                       ACK
```

Listing 16.3: Auszug aus der Paketfolge des Testfalldurchlaufs (1).

Das End-Device beginnt darauf, das Netz nach einem Coordinator abzusuchen (Vgl. Listing 16.4, Paket 27). Hier wird die Verbindung zwischen Inanspruchnahme von MAC-Diensten und übertragenen MAC- Frames deutlich - jeder *Scan-Request* resultiert im Versand von *MAC Command Frames*, die einen Beacon anfor-

dern (Vgl. Paket 28). Dies bleibt jedoch erfolglos – Statuscode **MAC_NO_BEACON** (Paket 44). Das End-Device startet den Scan-Vorgang erneut (Paket 45).

```
packet 27/226
>> [30] 03 00 04 25 FF FF 17 45 E5 20 A6 00 00 00 00 00 00 10 D8 FE 07 01 00 F8 FF 07
     05 00 26 0A
 TS(DP):  +             0:109:000 [s:ms:us]
 TS(M):   +             0:042:528 [s:ms:us]
 diff:    +               -67      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  REQ      SCAN                          DOCA
  scantype: MLME_SCAN_TYPE_ACTIVE      , scan duration: 05
  channelMask: 07 FF F8 00 , channelPage: 00
---------------------------------
packet 28/226
>> [33] 05 00 04 25 FF FF 17 45 E5 06 AF 00 00 00 00 00 00 22 63 FE 0A 03 08 96 FF FF
     FF FF 07 00 00 15 0A
 TS(DP):  +             0:125:000 [s:ms:us]
 TS(M):   +             0:044:806 [s:ms:us]
 diff:    +               -81      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
FRAME    TX       MAC COMMAND FRAME             DOCA
03 08 96 FF FF FF FF 07 00 00
MHR:
 FCF: type:C |sec:- |pend:- |ack:- |panIdCompr:-
 addrModes:  dest:S src:N |version:2003 |seqNo:150
 addressing:
  dest  PanID:FF FF  addr:FF FF
MAC Payload:
 Beacon Req
MFR: 0000 (unreliable)
---------------------------------
...
---------------------------------
packet 44/156
>> [31] 03 00 04 25 FF FF 17 45 E5 45 17 7C 00 00 00 00 00 12 D8 FE 08 EA 01 00 00 00
     00 00 00 6C 0A
 TS(DP):  +             8:250:000 [s:ms:us]
 TS(M):   +             8:132:421 [s:ms:us]
 diff:    +              -118      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  CNF      SCAN                          DOCA
  scantype: MLME_SCAN_TYPE_ACTIVE      , page: 00 , unscanned ch: 00 00 00 00
  result List entries: 0
  status: MAC_NO_BEACON
---------------------------------
packet 45/156
>> [30] 03 00 04 25 FF FF 17 45 E5 68 1B 7C 00 00 00 00 00 10 D8 FE 07 01 00 F8 FF 07
     06 00 AC 0A
 TS(DP):  +             8:266:000 [s:ms:us]
 TS(M):   +             8:133:480 [s:ms:us]
 diff:    +              -133      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  REQ      SCAN                          DOCA
  scantype: MLME_SCAN_TYPE_ACTIVE      , scan duration: 06
  channelMask: 07 FF F8 00 , channelPage: 00
```

Listing 16.4: Auszug aus der Paketfolge des Testfalldurchlaufs (2).

Zwischenzeitlich wird der Coordinator angeschaltet und setzt in der PIB ein Feld, wodurch die Assoziation anderer Geräte erlaubt wird (Vgl. Listing 16.5, Pakete 64, 65). Er scannt ebenfalls das Netz ab, um sicherzustellen, dass kein anderer Coordinator durch ihn gestört wird (Paket 66).

```
packet 64/156
>> [25] 03 00 04 25 FF FF 17 33 6B 84 33 2D 01 00 00 00 00 10 DA FE 02 41 01 80 0A
 TS(DP):  +            20:125:000 [s:ms:us]
 TS(M):   +            19:739:524 [s:ms:us]
 diff:    +              -386      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
WPAN CB  REQ      SET                            DOCA
  attr: macAssociationPermit          , value: 01
---------------------------------
packet 65/156
>> [25] 03 00 04 25 FF FF 17 33 6B 47 36 2D 01 00 00 00 00 12 DA FE 02 00 41 45 0A
 TS(DP):  +            20:125:000 [s:ms:us]
 TS(M):   +            19:740:231 [s:ms:us]
 diff:    +              -385      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
WPAN CB  CNF      SET                            DOCA
  attr: macAssociationPermit
  status: MAC_SUCCESS
---------------------------------
packet 66/156
>> [30] 03 00 04 25 FF FF 17 33 6B 48 3A 2D 01 00 00 00 00 10 D8 FE 07 01 00 F8 FF 07
     01 00 02 0A
 TS(DP):  +            20:125:000 [s:ms:us]
 TS(M):   +            19:741:256 [s:ms:us]
 diff:    +              -384      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
WPAN CB  REQ      SCAN                           DOCA
  scantype: MLME_SCAN_TYPE_ACTIVE      , scan duration: 01
  channelMask: 07 FF F8 00 , channelPage: 00
```

Listing 16.5: Auszug aus der Paketfolge des Testfalldurchlaufs (3).

Da das Ergebnis auch hier negativ ist (Vgl. Listing 16.6, Paket 85), startet der Coordinator daraufhin ein eigenes PAN (Paket 86) durch einen *Start Request*, was durch Paket 87 erfolgreich quittiert wird (*Start Confirmation*).

```
packet 85/156
>> [31] 03 00 04 25 FF FF 17 33 6B 26 73 39 01 00 00 00 00 12 D8 FE 08 EA 01 00 00 00
     00 00 00 D7 0A
 TS(DP):  +            20:891:000 [s:ms:us]
 TS(M):   +            20:542:246 [s:ms:us]
 diff:    +              -349      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
WPAN CB  CNF      SCAN                           DOCA
  scantype: MLME_SCAN_TYPE_ACTIVE      , page: 00 , unscanned ch: 00 00 00 00
  result List entries: 0
  status: MAC_NO_BEACON
---------------------------------
packet 86/156
>> [32] 03 00 04 25 FF FF 17 33 6B 99 77 39 01 00 00 00 00 10 DB FE 09 FE CA 16 00 0F
     0F 01 00 00 A4 0A
 TS(DP):  +            20:906:000 [s:ms:us]
 TS(M):   +            20:543:385 [s:ms:us]
 diff:    +              -363      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
WPAN CB  REQ      START                          DOCA
  PanId: CA FE , Channel: 16 , Page: 00 , BO: 0F , SO: 0F , Coord: true
  BattLifeExt: false, CoordRealignment: false
---------------------------------
packet 87/156
>> [24] 03 00 04 25 FF FF 17 33 6B 72 8B 39 01 00 00 00 00 12 DB FE 01 00 9A 0A
 TS(DP):  +            20:906:000 [s:ms:us]
 TS(M):   +            20:548:466 [s:ms:us]
 diff:    +              -358      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
WPAN CB  CNF      START                          DOCA
  status: MAC_SUCCESS
```

Listing 16.6: Auszug aus der Paketfolge des Testfalldurchlaufs (4).

Das End-Device scannt hingegen immer noch. Da aber nun ein Coordinator existiert, wird die Anfrage dort empfangen (Vgl. Listing 16.7, Paket 104) und durch einen *Beacon-Frame* beantwortet (Paket 105), der schließlich im End-Device eingeht (Paket 107). Anhand des Felds **`panCoord`** wird angezeigt, dass der Beacon-Frame von einem PAN-Coordinator kam, **`assocPermit`** signalisiert, dass eine Anmeldung an diesem Netz möglich ist.

```
packet 104/156
>> [39] 05 00 04 25 FF FF 17 33 6B AD BD 14 02 00 00 00 00 21 63 01 10 2A BA 15 02 00
     2D 03 08 4D FF FF FF FF 07 36 1E FD 0A
 TS(DP):  +            35:281:000 [s:ms:us]
 TS(M):   +            34:913:709 [s:ms:us]
 diff:    +              -368      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
FRAME    RX EXT   MAC COMMAND FRAME             ACK
2A BA 15 02 00 2D 03 08 4D FF FF FF FF 07 36 1E
TS: 02 15 BA 2A , ED:00, LQI:2D
MHR:
 FCF: type:C |sec:- |pend:- |ack:- |panIdCompr:-
 addrModes:  dest:S src:N |version:2003 |seqNo:77
 addressing:
  dest  PanID:FF FF  addr:FF FF
MAC Payload:
 Beacon Req
MFR: 361E (unreliable)
--------------------------------
packet 105/156
>> [36] 05 00 04 25 FF FF 17 33 6B 64 CE 14 02 00 00 00 00 22 60 FE 0D 00 80 35 FE CA
     00 00 FF CF 00 00 00 00 D7 0A
 TS(DP):  +            35:281:000 [s:ms:us]
 TS(M):   +            34:917:988 [s:ms:us]
 diff:    +              -364      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
FRAME    TX       BEACON FRAME                  DOCA
00 80 35 FE CA 00 00 FF CF 00 00 00 00
MHR:
 FCF: type:B |sec:- |pend:- |ack:- |panIdCompr:-
 addrModes:  dest:N src:S |version:2003 |seqNo:53
 addressing:
  src   PanID:CA FE  addr:00 00
MAC Payload:
 BO:15, SO:15, capSlot:15, battLifeExt:-, panCoord:x, assocPermit:x
 GTS: descrCount:0, permit: -
MFR: 0000 (unreliable)
--------------------------------
...
--------------------------------
packet 107/156
>> [42] 05 00 04 25 FF FF 17 45 E5 0C 6C 18 02 00 00 00 00 21 60 01 13 27 68 19 02 37
     FF 00 80 35 FE CA 00 00 FF CF 00 00 16 86 07 0A
 TS(DP):  +            35:281:000 [s:ms:us]
 TS(M):   +            35:154:956 [s:ms:us]
 diff:    +              -127      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
FRAME    RX EXT   BEACON FRAME                  ACK
27 68 19 02 37 FF 00 80 35 FE CA 00 00 FF CF 00 00 16 86
TS: 02 19 68 27 , ED:37, LQI:FF
MHR:
 FCF: type:B |sec:- |pend:- |ack:- |panIdCompr:-
 addrModes:  dest:N src:S |version:2003 |seqNo:53
 addressing:
  src   PanID:CA FE  addr:00 00
MAC Payload:
 BO:15, SO:15, capSlot:15, battLifeExt:-, panCoord:x, assocPermit:x
 GTS: descrCount:0, permit: -
MFR: 0016 (unreliable)
```

Listing 16.7: Auszug aus der Paketfolge des Testfalldurchlaufs (5).

Da das End-Device noch auf anderen Kanälen nach Coordinatoren scannt, wird die Antwort zunächst zurückgestellt und resultiert erst nach Abschluss des Scanvorganges in einer *Scan Confirmation* (Vgl. Listing 16.8, Paket 114). So wurde nicht bemerkt, dass der Coordinator inzwischen wieder durch einen simulierten Reset ausgefallen ist. Ein nun folgender Versuch, sich am gefundenen Netz anzumelden (Pakete 115, 116), scheitert (Paket 117 – Statuscode **`MAC_NO_ACK`**).

```
packet 114/156
>> [52] 03 00 04 25 FF FF 17 45 E5 2F AC 64 02 00 00 00 00 12 D8 FE 1D 00 01 00 00 00
     00 00 01 02 FE CA 00 00 00 00 00 00 00 00 16 00 FF CF 37 FF 27 68 19 02 D5 0A
 TS(DP):  +            40:266:000 [s:ms:us]
 TS(M):   +            40:152:111 [s:ms:us]
 diff:    +              -114      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  CNF      SCAN                          DOCA
  status: MAC_SUCCESS
  scantype: MLME_SCAN_TYPE_ACTIVE      , page: 00 , unscanned ch: 00 00 00 00
  result List entries: 1
  list entry: 0 coordAddrSpec: addrMode: shrt , PanID: CA FE , Addr: 00 00 , Channel:
       16 , ChPage: 00
  SuperframeSpec: CF FF , GTS permit: 37 , LQI: FF , TS: 02 19 68 27
  status: MAC_SUCCESS
--------------------------------
packet 115/156
>> [37] 03 00 04 25 FF FF 17 45 E5 1C B3 64 02 00 00 00 00 10 D0 FE 0E 16 00 02 FE CA
     00 00 00 00 00 00 00 00 80 CC 0A
 TS(DP):  +            40:313:000 [s:ms:us]
 TS(M):   +            40:153:884 [s:ms:us]
 diff:    +              -160      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  REQ      ASSOC.                        DOCA
  Channel: 16 , ChannelPage: 00
  Coordinator: addrMode: shrt , PanID: CA FE , Addr: 00 00
  Capabilities: PAN Coord:0 FFD:0 mainsPwrd:0 RxOnWhenIdle:0 reqShrtAddr:1
--------------------------------
packet 116/156
>> [44] 05 00 04 25 FF FF 17 45 E5 1A E7 64 02 00 00 00 00 22 63 00 15 23 C8 52 FE CA
     00 00 FF FF E5 45 17 FF FF 25 04 00 01 80 00 00 C6 0A
 TS(DP):  +            40:313:000 [s:ms:us]
 TS(M):   +            40:167:194 [s:ms:us]
 diff:    +              -146      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
FRAME    TX       MAC COMMAND FRAME             NACK
23 C8 52 FE CA 00 00 FF FF E5 45 17 FF FF 25 04 00 01 80 00 00
MHR:
 FCF: type:C |sec:- |pend:- |ack:x |panIdCompr:-
 addrModes:  dest:S src:L |version:2003 |seqNo:82
 addressing:
  dest  PanID:CA FE  addr:00 00
  src   PanID:FF FF  addr:00 04 25 FF FF 17 45 E5
MAC Payload:
 Association Req,
 capabilities: PAN Coord:0 FFD:0 mainsPwrd:0 RxOnWhenIdle:0 reqShrtAddr:1
MFR: 0000 (unreliable)
--------------------------------
packet 117/156
>> [26] 03 00 04 25 FF FF 17 45 E5 66 ED 64 02 00 00 00 00 12 D0 FE 03 FF FF E9 AE 0A
 TS(DP):  +            40:313:000 [s:ms:us]
 TS(M):   +            40:168:806 [s:ms:us]
 diff:    +              -145      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  CNF      ASSOC.                        DOCA
  assocShortAddr: FF FF
  status: MAC_NO_ACK
```

Listing 16.8: Auszug aus der Paketfolge des Testfalldurchlaufs (6).

Wird in einem *zweiten Versuch* der Coordinator zur gleichen Zeit ein-, aber erst später wieder ausgeschaltet, gelingt unterdessen die Anmeldung mit der Zuweisung einer Short-Adresse (Vgl. Listing 16.9, Paket 127) und das End-Device beginnt mit einer regelmäßigen Datenübertragung. Die Pakete 128-133 geben dabei einen Übertragungszyklus wieder.

```
packet 127/226
>> [26] 03 00 04 25 FF FF 17 45 E5 AD 54 6C 02 00 00 00 00 12 D0 FE 03 01 00 00 3C 0A
 TS(DP):  +            40:797:000 [s:ms:us]
 TS(M):   +            40:653:997 [s:ms:us]
 diff:    +              -144      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  CNF      ASSOC.                          DOCA
  assocShortAddr: 00 01
  status: MAC_SUCCESS
---------------------------------
packet 128/226
>> [39] 03 00 04 25 FF FF 17 45 E5 8F 7E 92 02 00 00 00 00 10 C0 FE 10 02 02 FE CA 00
     00 00 00 00 00 00 01 01 01 00 FF 0A
 TS(DP):  +            43:312:000 [s:ms:us]
 TS(M):   +            43:155:087 [s:ms:us]
 diff:    +              -157      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  REQ      DATA                            DOCA
  SrcAddrMode: shrt
  DstAddrMode: shrt
  DstPanId: CA FE DstAddr: 00 00
  TxOptions: ACK: 1 , trx (only if beacon used): CAP, direct
  msduHandle: 01 , msduLength: 01
  msduData: 00
---------------------------------
packet 129/226
>> [35] 05 00 04 25 FF FF 17 45 E5 70 90 92 02 00 00 00 00 22 61 01 0C 61 88 C8 FE CA
     00 00 01 00 00 81 FF C7 0A
 TS(DP):  +            43:312:000 [s:ms:us]
 TS(M):   +            43:159:664 [s:ms:us]
 diff:    +              -153      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
FRAME    TX       DATA FRAME                      ACK
61 88 C8 FE CA 00 00 01 00 00 81 FF
MHR:
 FCF: type:D |sec:- |pend:- |ack:x |panIdCompr:x
 addrModes:  dest:S src:S |version:2003 |seqNo:200
 addressing:
  dest  PanID:CA FE  addr:00 00
  src   PanID:CA FE  addr:00 01
MAC Payload: 00
MFR: 81FF (unreliable)
---------------------------------
packet 130/226
>> [28] 05 00 04 25 FF FF 17 33 6B 0A 41 8F 02 00 00 00 00 22 62 01 05 02 00 C8 00 00
     23 0A
 TS(DP):  +            43:312:000 [s:ms:us]
 TS(M):   +            42:942:730 [s:ms:us]
 diff:    +              -370      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
FRAME    TX       ACK FRAME                       ACK
02 00 C8 00 00
MHR:
 FCF: type:A |sec:- |pend:- |ack:- |panIdCompr:-
 addrModes:  dest:N src:N |version:2003 |seqNo:200
 addressing:
MAC Payload: none
MFR: 0000 (unreliable)
---------------------------------
packet 131/226
>> [41] 05 00 04 25 FF FF 17 33 6B E2 44 8F 02 00 00 00 00 21 61 01 12 43 3C 90 02 00
     2D 61 88 C8 FE CA 00 00 01 00 00 8A 8C C1 0A
 TS(DP):  +            43:312:000 [s:ms:us]
 TS(M):   +            42:943:714 [s:ms:us]
 diff:    +              -369      [ms]
```

```
 ## Src: 00:04:25:FF:FF:17:33:6B ##
FRAME    RX EXT   DATA FRAME                    ACK
43 3C 90 02 00 2D 61 88 C8 FE CA 00 00 01 00 00 8A 8C
TS: 02 90 3C 43 , ED:00, LQI:2D
MHR:
 FCF: type:D |sec:- |pend:- |ack:x |panIdCompr:x
 addrModes:  dest:S src:S |version:2003 |seqNo:200
 addressing:
  dest  PanID:CA FE  addr:00 00
  src   PanID:CA FE  addr:00 01
MAC Payload: 00
MFR: 8A8C (unreliable)
--------------------------------
packet 132/226
>> [53] 03 00 04 25 FF FF 17 33 6B 14 4B 8F 02 00 00 00 00 11 C0 FE 1E 02 FE CA 01 00
     00 00 00 00 00 00 02 FE CA 00 00 00 00 00 00 00 00 01 00 2D C8 43 3C 90 02 86 0A
 TS(DP):  +            43:328:000 [s:ms:us]
 TS(M):   +            42:945:300 [s:ms:us]
 diff:    +              -383      [ms]
 ## Src: 00:04:25:FF:FF:17:33:6B ##
WPAN CB  IND      DATA                          DOCA
  SrcAddr: addrMode: shrt , PanID: CA FE , Addr: 00 01
  DstAddr: addrMode: shrt , PanID: CA FE , Addr: 00 00
  msduLength: 01 , msdu: 00
  mpduLinkQual: 2D , DSN: C8 , TS: 02 90 3C 43
--------------------------------
packet 133/226
>> [29] 03 00 04 25 FF FF 17 45 E5 80 95 92 02 00 00 00 00 12 C0 FE 06 01 00 ED BE 3A
     01 53 0A
 TS(DP):  +            43:328:000 [s:ms:us]
 TS(M):   +            43:160:960 [s:ms:us]
 diff:    +              -168      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  CNF      DATA                          DOCA
  msduHandle: 01 , TS: 01 3A BE ED
  status: MAC_SUCCESS
```

Listing 16.9: Auszug aus der Paketfolge des Testfalldurchlaufs (7).

Wird schließlich auch hier der Coordinator zurückgesetzt, wird dies anhand ausbleibender Übertragungsbestätigungen (Vgl. Listing 16.10, Paket 175 – Statuscode **MAC_NO_ACK**) deutlich. Zur Übertragung wird ein MAC-Frame genutzt, nach dessen Versand die in Paket 174 gezeigte Nachricht erzeugt wurde. Bereits hier signalisiert das Statusfeld (**NACK**), das der Frame zwar verschickt, das erforderliche ACK-Frame aber nicht empfangen wurde.

```
packet 173/226
>> [39] 03 00 04 25 FF FF 17 45 E5 C2 8C 9D 03 00 00 00 00 10 C0 FE 10 02 02 FE CA 00
     00 00 00 00 00 00 00 08 01 01 00 47 0A
 TS(DP):  +            60:937:000 [s:ms:us]
 TS(M):   +            60:656:834 [s:ms:us]
 diff:    +              -281      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  REQ      DATA                          DOCA
  SrcAddrMode: shrt
  DstAddrMode: shrt
  DstPanId: CA FE DstAddr: 00 00
  TxOptions: ACK: 1 , trx (only if beacon used): CAP, direct
  msduHandle: 08 , msduLength: 01
  msduData: 00
--------------------------------
packet 174/226
>> [35] 05 00 04 25 FF FF 17 45 E5 B4 C8 9D 03 00 00 00 00 22 61 00 0C 61 88 CF FE CA
     00 00 01 00 00 D2 FF 00 0A
 TS(DP):  +            60:968:000 [s:ms:us]
 TS(M):   +            60:672:180 [s:ms:us]
 diff:    +              -296      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
```

```
FRAME    TX       DATA FRAME                    NACK
61 88 CF FE CA 00 00 01 00 00 D2 FF
MHR:
 FCF: type:D |sec:- |pend:- |ack:x |panIdCompr:x
 addrModes:  dest:S src:S |version:2003 |seqNo:207
 addressing:
  dest  PanID:CA FE  addr:00 00
  src   PanID:CA FE  addr:00 01
MAC Payload: 00
MFR: D2FF (unreliable)
---------------------------------
packet 175/226
>> [29] 03 00 04 25 FF FF 17 45 E5 E1 CD 9D 03 00 00 00 00 12 C0 FE 06 08 E9 93 3A 19
     02 5E 0A
 TS(DP):  +            60:968:000 [s:ms:us]
 TS(M):   +            60:673:505 [s:ms:us]
 diff:    +              -295      [ms]
 ## Src: 00:04:25:FF:FF:17:45:E5 ##
WPAN CB  CNF      DATA                          DOCA
  msduHandle: 08 , TS: 02 19 3A 93
  status: MAC_NO_ACK
```

Listing 16.10: Auszug aus der Paketfolge des Testfalldurchlaufs (8).

Anhand der in den Paketen enthaltenen Zeitstempel könnte an dieser Stelle beispielsweise ermittelt werden, wieviel Zeit zwischen dem Bereitstehen eines Coordinators und der Anmeldung des End-Device daran verstreicht. Leider wird aber auch deutlich, dass die Uhren in den Knoten nicht synchron laufen. Dies ist durch die im Kapitel 11.2 angesprochenen Interruptsperren begründet, durch die Timer-Interrupt-Signale verloren gehen und der Knoten-lokale Zeitstempel zeitweise nicht inkrementiert wird. Der Versatz der Zeitstempel nimmt über die Versuchsdauer hin stetig zu und wird ebenfalls durch die Aktivität des Knotens beeinflusst – überträgt oder empfängt ein Knoten nur wenige Pakete, bleibt die Abweichung gering.

17 Projektstand und Ausblick

Im derzeitigen Projektstand umgesetzt wurde eine parametrierbare Knotenfirmware, mit der Knoten zur Laufzeit gesteuert und beobachtet werden können. Der entwickelte Testserver erlaubt den Betrieb mehrerer damit ausgestatteter Knoten, wobei im XML-Format definierte Testfälle parallel ausführbar sind und aufgezeichnet werden. Ein plattformneutraler Einsatz wird durch die Implementierung in Java ermöglicht. Zur Server-Knoten-Kommunikation wird ein Binärprotokoll verwendet, dessen Paketformat für spätere Erweiterungen offen ist. Anhand eines Paketinterpreters lassen sich zur Laufzeit oder bei der nochmaligen Wiedergabe von Testfällen die Vorgänge im Netz nachvollziehen.

Während der Entwicklung wurden nur Atmel-basierte Funkmodule verwendet, auf welche die Knotenfirmware derzeit noch manuell zu übertragen ist. Die Erweiterung des Protokollstack beeinflusst durch die Nachrichtenerzeugung das Laufzeitverhalten der Applikation. Aus der asynchronen Kommunikation zwischen Mikrocontroller und Transceiver können fehlerhaft erzeugte Nachrichten resultieren, die Verwendung von Interrupt-Sperren stört einen Knoten-lokalen Zeitgeber. Erschwerend bei der Auswertung von Testfällen ist, dass Nachrichten aufgrund ihrer unbestimmten Laufzeit nicht in der Reihenfolge ihrer Entstehung im Server eingehen. Einer zukünftigen Erweiterung vorbehalten ist die automatische paketübergreifende Erkennung von Ereignissen, wobei die zu ihrer Detektion erforderlichen Paketsequenzen abhängig vom Protokoll und der eingesetzten Applikation sind. In Steuerdateien lassen sich lediglich statische Parameter definieren, so dass ein reaktives Verhalten während der Testfallausführung manuell erfolgen muss. Wünschenswert wäre eine Benutzeroberfläche, mit der sich Testfälle passend zur aktuellen Konfiguration des Knotennetzes zusammenstellen lassen.

Der aktuelle Projektstand entspricht daher keinem fertig einsetzbaren Produkt, sondern stellt eine Zwischenversion innerhalb des Software-Lebenzyklus' dar. Er ist als schwerpunktorientiert implementiertes „Proof-of-concept" zu verstehen, wobei die dabei entdeckten Stolpersteine bei der Weiterentwicklung nützlich sein können. Die entworfene Systemarchitektur bildet ein Grundgerüst, das bereits für primitive Tests von Funkknoten und daraus gebildeten Netzen eingesetzt werden kann. Erst mit der Bereitstellung realer Knotenapplikationen können jedoch konkrete, darauf zugeschnittene Testszenarien entworfen, durchgeführt und sinnvoll ausgewertet werden.

Parallel zur Entwicklung des Panelsystems und dessen Integration das Testbed muss die Systemarchitektur um die Verwendung von Steuerknoten erweitert und verfeinert werden, entsprechende Vorkehrungen wurden dafür bereits getroffen.

Sollte sich dabei herausstellen, dass das System in seiner bisherigen Form so nicht verwendbar ist, darf vor einem Redesign nicht zurückgeschreckt werden. Gute Aussichten auf Wiederverwendung haben die Modifikationen des Atmel IEEE 802.15.4 Protokollstack und der daraus hervorgegangene Sniffer.

Dass ZigBee-basierte Funknetze derzeit noch nicht die gewünschte Marktdurchdringung erreicht haben, liegt einerseits daran, dass es wirtschaftlich nicht sinnvoll ist, bereits bestehende und zuverlässig funktionierende Anlagen auszutauschen – Verbreitungspotenzial besteht daher vorrangig bei der geplanten Auf- oder kompletten Neuausrüstung. Andererseits wird der neuartigen Technologie mit einer gewissen Skepsis begegnet, da bis auf Konzepte und eine Reihe zertifizierter Funkmodule nur wenige produktiv einsetzbare Lösungen bereitstehen und es widersprüchliche Aussagen bezüglich ihrer Störanfälligkeit gibt. Erst wenn sich Funknetze durch eine von kabelgebundenen Lösungen gewohnte Einfachheit und Robustheit auszeichnen, wird ihre Verbreitung zunehmen.

Alternativ stehen proprietäre Lösungen bereit, die an ZigBee hinsichtlich Leistungsumfang und Flexibilität zwar nicht heranreichen, aber gerade deshalb einfacher zu validieren und zu zertifizieren sind. Auch bei inzwischen selbstverständlichen Funktechnologien, wie beispielsweise GSM, Bluetooth oder Wireless LAN, betrug die Zeitspanne zwischen ihrer Konzeption, der Veröffentlichung von Standards, einer Bereitstellung in Geräten und dem breiten Markteinsatz jeweils mehrere Jahre. ZigBee steht mit der Veröffentlichung der letzten Spezifikation im Jahr 2007 und mittlerweile ersten verfügbaren Lösungen für den Produktiveinsatz hier noch am Anfang. Der ernstzunehmende Einsatz wird durch die Komplexität des Protokolls sowie einer verpflichtenden Zertifizierung eigener Anwendungen erschwert, weshalb viele Anbieter einfachere Protokolle einsetzen oder anwendungsspezifisch proprietär entwickeln.

Funktechnologien, die auf autonom arbeitenden, energiesparsamen, selbstorganisierenden Knoten basieren, sind zweifelsohne zukunftsträchtig. Ganz egal, welche Technologie sich letztendlich durchsetzen wird, Testsysteme dazu werden stets benötigt, um entwickelte Anwendungen vor der Auslieferung an den Kunden ausgiebig evaluieren zu können. Das in dieser Arbeit beschriebene Konzept einer Testplattform bietet diesbezüglich genügend Erweiterungspotenzial mit Entwicklungszeiten im Bereich von Mann-Monaten. Bei Fortsetzung des Projekts sollten bereits vorhandene Testplattformen im Auge behalten und ihre Veränderungen in die eigenen Überlegungen mit einbezogen werden.

Literatur- und Quellenverzeichnis

[Amr09] AMRHEIN, BEATRICE: JAXB – Java Architecture for XML Binding / Berner Fachhochschule Technik und Informatik. Version: Jun. 2009. http://www.sws.bfh.ch/~amrhein/Skripten/XML/JAXBSkript.pdf, Abruf: 5. Dez. 2009. 2009. – Forschungsbericht

[Atm09a] ATMEL CORPORATION (Hrsg.): *6LoWPAN*. http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4578. Version: 2009, Abruf: 16. Nov. 2009

[Atm09b] ATMEL CORPORATION (Hrsg.): *AVR2025: IEEE 802.15.4 MAC Software Package for AVR Z-Link, revision 2.3.1*. http://www.atmel.com/forms/software_download.asp?fn=dl_AVR2025_MAC_v_2%_3_1.zip. Version: 2009, Abruf: 16. Nov. 2009

[Atm09c] ATMEL CORPORATION (Hrsg.): *MCU Wireless - Devices*. http://www.atmel.com/dyn/products/devices.asp?family_id=676. Version: 2009, Abruf: 16. Nov. 2009

[Atm09d] ATMEL CORPORATION (Hrsg.): *MCU Wireless - Tools & Software*. http://www.atmel.com/dyn/products/tools.asp?family_id=676. Version: 2009, Abruf: 16. Nov. 2009

[Atm09e] ATMEL CORPORATION (Hrsg.): *MCU Wireless - ZigBit Wireless Modules*. http://www.atmel.com/products/zigbee/zigbit.asp?family_id=676. Version: 2009, Abruf: 16. Nov. 2009

[Atm09f] ATMEL CORPORATION (Hrsg.): *Produktpalette*. http://www.atmel.com/products/. Version: 2009, Abruf: 16. Nov. 2009

[Atm09g] ATMEL CORPORATION (Hrsg.): *Webpräsenz der Firma*. http://www.atmel.com. Version: 2009, Abruf: 16. Nov. 2009

[Aue09] AUER, WOLFGANG AND RITSCHEL, PATRICK: Nah und fern - Zigbee in der Praxis. In: *iX - Magazin für professionelle Informationstechnik* (2009), Jan., S. 120–124

[Bou09] BOUTET, ANTOINE: *Senslab - Very Large scale open wireless sensor network testbed*. http://www.senslab.info/index.php/Testbed. Version: Nov. 2009, Abruf: 22. Nov. 2009

[Cha06] CHAE, M.J. AND YOO, H.S. AND KIM, J.R. AND CHO, M.Y.: Bridge condition monitoring system using Wireless network (CDMA and Zigbee) / Korea Institute of Construction Technology. Version: 2006. http://www.iaarc.org/external/isarc2006-cd/www/ISARC2006/00066_200606122103.pdf, Abruf: 10. Aug. 2009. 2006. – Forschungsbericht

[Cit08] *CitySense – An Open, Urban-Scale Sensor Network Testbed*. http://www.citysense.net/. Version: 2008, Abruf: 22. Nov. 2009

[DE09a] DRESDEN ELEKTRONIK INGENIEURTECHNIK GMBH (Hrsg.): *Datenfunklösungen 802.15.4*. http://www.dresden-elektronik.de/shop/cat4.html. Version: 2009, Abruf: 12. Nov. 2009

[DE09b] DRESDEN ELEKTRONIK INGENIEURTECHNIK GMBH (Hrsg.): *dresden elektronik - Elektronikentwicklung - Elektronikfertigung*. http://www.dresden-elektronik.de. Version: 2009, Abruf: 12. Nov. 2009

[DN09] DAINTREE NETWORKS, INC. (Hrsg.): *Daintree Sensor Network Analyzer*. http://www.daintree.net/products/sna.php. Version: 2009, Abruf: 21. Nov. 2009

[Dun09] DUNKELS, ADAM : *Contiki - The Operating System for Embedded Smart Objects - the Internet of Things*. http://www.sics.se/contiki/. Version: 2009, Abruf: 16. Nov. 2009

[Ead07] EADY, FRED: *Hands-On ZigBee - Implementing 802.15.4 with Microcontrollers*. 1. Aufl. Oxford : Newnes, 2007

[Exs] EXSCALE: *ExScale: Extreme Scale Wireless Sensor Networking*. http://ceti.cse.ohio-state.edu/exscal/, Abruf: 12. Nov. 2009

[FS09] FREESCALE SEMICONDUCTOR, INC. (Hrsg.): *MC13213: 2.4 GHz RF Transceiver and 8-bit MCU with 60K of Flash for ZigBee Applications*. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MC13213%. Version: 2009, Abruf: 17. Nov. 2009

[FTD08] FUTURE TECHNOLOGY DEVICES INTERNATIONAL LTD. (Hrsg.): *Technical Note TN 103, FTDI USB Data Transfer Efficiency*. http://www.ftdichip.com/Documents/TechnicalNotes/TN_103_FTDI_USB_Data_Transfer_Efficiency(FT_000097).pdf. Version: 2008, Abruf: 22. Nov. 2009

[FTD09] FUTURE TECHNOLOGY DEVICES INTERNATIONAL LTD. (Hrsg.): *FT245R - USB FIFO IC*. http://www.ftdichip.com/Products/FT245R.htm. Version: 2009, Abruf: 22. Nov. 2009

[Gif] GIFFORD, Ian ; IEEE 802.15 WORKING GROUP FOR WPAN (Hrsg.): *IEEE 802.15 Working Group for Wireless Personal Area Networks (WPANs)*. http://www.ieee802.org/15/, Abruf: 12. Nov. 2009

[Gis08] GISLASON, DREW: *Zigbee Wireless Networking*. 1. Aufl. Oxford, Woburn : Butterworth Heinemann, 2008

[Haa08] HAAS, HEIDI: Intelligenz auf dem Silbertablett. In: *Bauguide Hausautomation* 1 (2008), 12-15. http://www.ceesar.ch/fileadmin/Dateien/PDF/Intelligenz.pdf, Abruf: 10. Aug. 2009

[Hae08] HAENSELMANN, THOMAS: *Sensor Networks*. Version: Dec. 2008. http://pi4.informatik.uni-mannheim.de/~haensel/sn_book.pdf, Abruf: 10. Aug. 2009

[Heg08] HEGGLIN, RAPHAEL: Wenn das Haus mitdenkt. In: *ecoLife* 1 (2008), 14-17. http://www.ceesar.ch/fileadmin/Dateien/PDF/Publikationen/0801_ecoLife_Artikel_WennDasHausMitdenkt.pdf, Abruf: 10. Aug. 2009

[Hom03] HOMERF™TECHNICAL COMMITTEE (Hrsg.): *HomeRF 2.01 Specification*. http://www.palowireless.com/homerf/homerfspec.asp. Version: 2003, Abruf: 12. Nov. 2009

[IEE03] IEEE COMPUTER SOCIETY (Hrsg.): *IEEE Std 802.15.4-2003*
IEEE Standard for Information technology –
Telecommunications and information exchange between systems –
Local and metropolitan area networks –
Specific requirements
Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs). Version: 2003. http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf, Abruf: 11. Jun. 2009

[IEE06] IEEE COMPUTER SOCIETY (Hrsg.): *IEEE Std 802.15.4-2006*
IEEE Standard for Information technology –
Telecommunications and information exchange between systems –
Local and metropolitan area networks –
Specific requirements
Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)
(Revision of IEEE Std 802.15.4-2003). Version: 2006. http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf, Abruf: 10. Okt. 2009

[IEE07] IEEE COMPUTER SOCIETY (Hrsg.): *IEEE 802.15.4a-2007*
IEEE Standard for Information technology –
Telecommunications and information exchange between systems –
Local and metropolitan area networks –
Specific requirements
Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)
Amendment 1: Add Alternate PHYs. Version: 2007. http://standards.ieee.org/getieee802/download/802.15.4a-2007.pdf, Abruf: 10. Okt. 2009

[IEE09a] IEEE COMPUTER SOCIETY (Hrsg.): *IEEE 802.15.4c-2009*
IEEE Standard for Information technology –
Telecommunications and information exchange between systems –
Local and metropolitan area networks –
Specific requirements
Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)
Amendment 2: Alternative Physical Layer Extension to support one or more of the Chinese 314-316 MHz, 430-434 MHz, and 779-787 MHz bands. Version: 2009. http://standards.ieee.org/getieee802/download/802.15.4c-2009.pdf, Abruf: 10. Okt. 2009

[IEE09b] IEEE COMPUTER SOCIETY (Hrsg.): *IEEE 802.15.4d-2009 IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 3: Alternative Physical Layer Extension to support the Japanese 950 MHz bands*. Version: 2009. http://standards.ieee.org/getieee802/download/802.15.4d-2009.pdf, Abruf: 10. Okt. 2009

[IES08] INDUSTRIAL EMBEDDED SYSTEMS (Hrsg.): *Arch Rock Brings IP-Based Wireless Sensor Networking to Atmel Embedded Hardware*. http://www.industrial-embedded.com/news/Technology+Partnerships/11353. Version: Apr. 2008, Abruf: 16. Nov. 2009

[Jar09] JARVI, KEANE: *RXTX: The Prescription for Transmission*. http://www.rxtx.org. Version: 04 2009, Abruf: 7. Dez. 2009

[Kar06] KARTES, CHRISTOPH: Interferenz im 2,4-GHz-Band. In: *Funkschau* (2006), Nov., 12-15. http://www.funkschau.de/fileadmin/media/heftarchiv/articles/Jahrgang_2006/11_2006/fs_0611_s12.pdf

[Mat] MATTERN, FRIEDEMANN: *Ubiquitous Computing: Schlaue Alltagsgegenstände – Die Vision von der Informatisierung des Alltags* http://www.vs.inf.ethz.ch/publ/papers/mattern2004_sev.pdf

[MN09a] MESHNETICS (Hrsg.): *MeshBean Development Board*. http://www.meshnetics.com/dev-tools/meshbean/. Version: 2009, Abruf: 17. Nov. 2009

[MN09b] MESHNETICS (Hrsg.): *ZigBee FAQ*. http://www.meshnetics.com/zigbee-faq/#9. Version: 2009, Abruf: 17. Nov. 2009

[MN09c] MESHNETICS (Hrsg.): *ZigBee Modules*. http://www.meshnetics.com/zigbee-modules/. Version: 2009, Abruf: 17. Nov. 2009

[MN09d] MESHNETICS (Hrsg.): *ZigBee Networking Topologies*. http://www.meshnetics.com/netcat_files/11_148.png. Version: 2009, Abruf: 16. Nov. 2009

[Mot09] HARVARD UNIVERSITY (Hrsg.): *Harvard Sensor Network Testbed*. http://motelab.eecs.harvard.edu/. Version: Feb. 2009, Abruf: 22. Nov. 2009

[Pal09] PALANISWAMI, M. ; UNIVERSITY OF MELBOURNE (Hrsg.): *BigNet - Sensor Network Testbed*. http://www.issnip.unimelb.edu.au/research_program/sensor_networks/infrastructure/bignet_testbed. Version: Sept. 2009, Abruf: 22. Nov. 2009

[Pet09] PETROVA, ET AL.: Performance Study of IEEE 802.15.4 Using Measurements and Simulations / RWTH Aachen University, Department of Wireless Networks. Version: März 2009. http://www.mobnets.rwth-aachen.de/pub/ZigBee.pdf, Abruf: 10. Okt. 2009. 2009. – Forschungsbericht

[Pis01] PISTER, KRISTOFER S.J. AND KAHN, JOE AND BOSER,BERNHARD: *SMART DUST - Autonomous sensing and communication in a cubic millimeter*. http://robotics.eecs.berkeley.edu/~pister/SmartDust/. Version: 2001, Abruf: 10. Aug. 2009

[Sca08] FREIE UNIVERSITÄT BERLIN, AG COMPUTER SYSTEMS & TELEMATICS (Hrsg.): *Welcome at ScatterWeb*. http://cst.mi.fu-berlin.de/projects/ScatterWeb/. Version: 2008, Abruf: 22. Nov. 2009

[SeN09] UNIVERSITY OF ROSTOCK, FACULTY OF INFORMATICS AND ELECTRICAL ENGINEERING (Hrsg.): *About Senets*. http://senets.e-technik.uni-rostock.de/. Version: 10 2009, Abruf: 22. Nov. 2009

[Sik04] SIKORA, AXEL: Short-Range Wireless Networking mit IEEE802.15.4 und ZigBee: Möglichkeiten und Herausforderungen / Berufsakademie Lörrach, Fachrichtung Informationstechnik, Steinbeis-Transferzentrum für Embedded Design und Networking. Version: 6 2004. www.stzedn.de/tl_files/files/stz_zigbee_de_entwicklerforum_040706.pdf, Abruf: 10. Okt. 2009. 2004. – Forschungsbericht

[SSS08] SSS ONLINE, INC. (Hrsg.): *ZigBee Stuff*. http://sss-mag.com/zigbee.html. Version: Aug. 2008, Abruf: 17. Nov. 2009

[Ten07] TENBUSCH, TIM: Anwendungsszenarien für Sensornetze. Version: Jul. 2007. http://cst.mi.fu-berlin.de/teaching/SS07/19554-S-TI/reports/tenbusch07anwendungsszenarien.pdf, Abruf: 14. Aug. 2009. 2007. – Forschungsbericht

[Ter09] TERRACOTTA, INC.: *Quartz Enterprise Job Scheduler*. http://www.opensymphony.com/quartz/. Version: 12 2009, Abruf: 7. Dez. 2009

[Twi08] TU BERLIN (Hrsg.): *TKN Wireless Indoor Sensor network Testbed (TWIST)*. http://www.twist.tu-berlin.de/. Version: Jun. 2008, Abruf: 22. Nov. 2009

[Wac09] WACHTLER, AXEL: *µracoli - The µController Radio Communication Library*. http://www.nongnu.org/uracoli/. Version: 2009, Abruf: 21. Nov. 2009. – Bibliothek für Z-Link basierte Atmel-Transceiver

[Wei91] WEISER, MARK: The Computer for the 21st Century. Version: 09 1991. http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html, Abruf: 10. Aug. 2009. 1991

[WI] WIRESHARK FOUNDATION (Hrsg.): *Wireshark - the worlds foremost network protocol analyzer*. http://www.wireshark.org/, Abruf: 21. Nov. 2009. – ein freier Netzwerk-Protokoll-Analyzer

[Wik09] WIKIPEDIA: *ZigBee — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=ZigBee&oldid=324255850#Origi%ns_of_ZigBee_name. Version: 2009, Abruf: 17. Nov. 2009

[Win09] WINAVR (Hrsg.): *WinAVR : AVR-GCC for Windows*. http://winavr.sourceforge.net/. Version: 2009, Abruf: 1. Dez. 2009

[Wob09] WOBST, RICHARD: Sicher funken - Kryptografie in ZigBee-Netzwerken. In: *iX - Magazin für professionelle Informationstechnik* (2009), Apr., S. 145–148

[ZA09a] ZIGBEE ALLIANCE (Hrsg.): *ZigBee Alliance, FAQ*. http://www.zigbee.org/About/FAQ/tabid/192/Default.aspx#21. Version: 2009, Abruf: 16. Nov. 2009

[ZA09b] ZIGBEE ALLIANCE (Hrsg.): *ZigBee Alliance Mitgliederliste*. http://www.zigbee.org/About/OurMembers/tabid/191/Default.aspx. Version: 2009, Abruf: 12. Nov. 2009

[ZA09c] ZIGBEE ALLIANCE (Hrsg.): *ZigBee Alliance, ZigBee Spezifikation*. http://www.zigbee.org/ZigBeeSpecificationDownloadRequest/tabid/311/Def%ault.aspx. Version: 2009, Abruf: 16. Nov. 2009

[ZEN09] MICROCHIP TECHNOLOGY INC. (Hrsg.): *ZENA Network Analyzer*. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId%=1406&dDocName=en520682. Version: 2009, Abruf: 21. Nov. 2009

Anhang

A Testfall-Steuerdatei für Beispieldurchlauf

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:testcase
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tns="http://www.dresden-elektronik.de/testcases"
  xsi:schemaLocation="http://www.dresden-elektronik.de/testcases testcase.xsd">

  <networkDefinition>
    <networkParameters>
      <superframeOrder>15</superframeOrder>
      <beaconOrder>15</beaconOrder>
      <channel>22</channel>
      <panId>CAFE</panId>
    </networkParameters>

    <requiredNetworkMember>
      <description>PAN-Coordinator</description>
      <addr>00-04-25-FF-FF-17-33-6B</addr>
      <type>
        <regularNode>
          <role>coordinator</role>
        </regularNode>
      </type>
    </requiredNetworkMember>

    <requiredNetworkMember>
      <description>Sensornode</description>
      <addr>00-04-25-FF-FF-17-45-E5</addr>
      <type>
        <regularNode>
          <role>end-device</role>
        </regularNode>
      </type>
    </requiredNetworkMember>
  </networkDefinition>

  <testcaseDefinition>
    <testcaseParameters>
      <description>switch coordinator delayed on and earlier off</description>
      <startOffset>5</startOffset>
      <duration>60</duration>
      <loglevel>full</loglevel>
      <executionLaterAllowed>false</executionLaterAllowed>
      <rescheduleOnServerShutdown>false</rescheduleOnServerShutdown>
      <nodeAvailabilityCheck>
        <retryCounter>3</retryCounter>
        <waitDuration>2</waitDuration>
      </nodeAvailabilityCheck>
    </testcaseParameters>

    <!--  switch end-device on at the beginning of the testcase -->
    <nodeStateChange>
      <nodeAddress>00-04-25-FF-FF-17-45-E5</nodeAddress>
      <timeOffset>0</timeOffset>
      <newState>on</newState>
    </nodeStateChange>

    <!--  switch coordinator on after 20 time units, off after 40 -->
    <nodeStateChange>
      <nodeAddress>00-04-25-FF-FF-17-33-6B</nodeAddress>
      <timeOffset>20</timeOffset>
      <newState>on</newState>
    </nodeStateChange>
    <nodeStateChange>
      <nodeAddress>00-04-25-FF-FF-17-33-6B</nodeAddress>
      <timeOffset>40</timeOffset>
      <newState>off</newState>
    </nodeStateChange>
  </testcaseDefinition>
</tns:testcase>
```

Listing A.1: Im Beispieldurchlauf genutzte Testfall-Steuerdatei

B Schemadefinition der Testfall-Steuerdatei

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.dresden-elektronik.de/testcases"
  xmlns:tns="http://www.dresden-elektronik.de/testcases"  >

  <!--  ##################  common type definitions  ################### -->

  <xsd:simpleType name="retryCountType">
    <xsd:restriction base="xsd:byte">
      <xsd:minInclusive value="0" />
      <xsd:maxInclusive value="10" />
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="macAddress">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="([0-9,a-f,A-F]{2}-){7}[0-9,a-f,A-F]{2}"></xsd:pattern>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="oneHexByte">
    <xsd:restriction base="xsd:hexBinary">
      <xsd:length value="1"></xsd:length>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="limitedTimeType">
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="0" />
      <xsd:maxInclusive value="86400" />
    </xsd:restriction>
  </xsd:simpleType>


  <!--  ###################  network definition  #################### -->
  <xsd:complexType name="networkParametersType">
    <xsd:all>
      <xsd:element name="beaconOrder" minOccurs="0" >
        <xsd:simpleType>
          <xsd:restriction base="xsd:byte">
            <xsd:minInclusive value="0" />
                    <xsd:maxInclusive value="15" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="superframeOrder" minOccurs="0" >
        <xsd:simpleType>
          <xsd:restriction base="xsd:byte">
            <xsd:minInclusive value="0" />
                    <xsd:maxInclusive value="15" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="channel">
        <xsd:simpleType>
          <xsd:restriction base="xsd:byte">
            <xsd:minInclusive value="0" />
            <xsd:maxInclusive value="26" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="panId">
        <xsd:simpleType>
          <xsd:restriction base="xsd:hexBinary">
            <xsd:length value="2"></xsd:length>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="security" minOccurs="0">
        <xsd:complexType>
```

```
        <xsd:sequence>
          <xsd:element name="securityEnabled" type="xsd:boolean" />
          <xsd:element name="networkKey" type="xsd:anySimpleType" minOccurs="0" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:all>
</xsd:complexType>

<xsd:complexType name="requiredNetworkMemberType">
  <xsd:all>
    <xsd:element name="description" type="xsd:string" minOccurs="0" />
    <xsd:element name="addr" type="tns:macAddress" />
    <xsd:element name="type">
      <xsd:complexType>
        <xsd:choice>
          <xsd:element name="regularNode">
            <xsd:complexType>
              <xsd:all>
                <xsd:element name="role">
                  <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                      <xsd:enumeration value="coordinator" />
                      <xsd:enumeration value="end-device" />
                    </xsd:restriction>
                  </xsd:simpleType>
                </xsd:element>
                <xsd:element name="ucType" type="xsd:string" minOccurs="0" />
                <xsd:element name="trxType" type="xsd:string" minOccurs="0" />
                <xsd:element name="stack" minOccurs="0">
                  <xsd:complexType>
                    <xsd:sequence>
                      <xsd:element name="version" type="xsd:string" />
                      <xsd:element name="parameters" type="xsd:string" />
                    </xsd:sequence>
                  </xsd:complexType>
                </xsd:element>
                <xsd:element name="hfAntenna" minOccurs="0">
                  <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                      <xsd:enumeration value="internal" />
                      <xsd:enumeration value="external" />
                    </xsd:restriction>
                  </xsd:simpleType>
                </xsd:element>
              </xsd:all>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="controllerNode">
            <xsd:complexType>
              <xsd:all>
                <xsd:element name="description" type="xsd:string" minOccurs="0" />
                <!--  ... -->
              </xsd:all>
            </xsd:complexType>
          </xsd:element>

        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:all>
</xsd:complexType>


<!--  ###############  common testcase parameters  ################# -->

<xsd:complexType name="testcaseParametersType">
  <xsd:all>
    <xsd:element name="description" type="xsd:string" minOccurs="0" />
    <xsd:element name="duration" type="tns:limitedTimeType" />
    <xsd:element name="startOffset" type="tns:limitedTimeType" />
    <xsd:element name="loglevel" minOccurs="0">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="full" />
```

```
            <xsd:enumeration value="limited" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>

      <xsd:element name="executionLaterAllowed" type="xsd:boolean" />
      <xsd:element name="rescheduleOnServerShutdown" type="xsd:boolean" />

      <xsd:element name="nodeAvailabilityCheck">
        <xsd:complexType>
          <xsd:all>
            <xsd:element name="retryCounter" type="tns:retryCountType" />
            <xsd:element name="waitDuration" type="tns:limitedTimeType" />
          </xsd:all>
        </xsd:complexType>
      </xsd:element>
    </xsd:all>
  </xsd:complexType>


  <xsd:complexType name="nodeStateChangeType" >
    <xsd:all>
      <xsd:element name="nodeAddress" type="tns:macAddress" />
      <xsd:element name="timeOffset" type="tns:limitedTimeType" />
      <xsd:element name="newState">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="on" />
            <xsd:enumeration value="off" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:all>
  </xsd:complexType>


  <xsd:complexType name="dataPacketsType" >
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0" />
      <xsd:element name="message">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="msg_t">
              <xsd:simpleType>
                <xsd:restriction base="xsd:hexBinary">
                  <xsd:length value="1"></xsd:length>
                  <xsd:pattern value="00|02"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="addr" type="tns:macAddress" />

            <xsd:element name="cmd">
              <xsd:simpleType>
                <xsd:restriction base="tns:oneHexByte">
                  <xsd:pattern value="0[1-6]|1[0-3]"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>

            <xsd:element name="attr">
              <xsd:simpleType>
                <xsd:restriction base="tns:oneHexByte">
                  <xsd:pattern value="0[0-7]|1[0-F]|2[0-8]|4[0-F]|5[0-F]|A[A-E]|B
                      [0-7]|D3"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>

            <xsd:element name="ack">
              <xsd:simpleType>
                <xsd:restriction base="tns:oneHexByte">
                  <xsd:pattern value="FE"/>
                </xsd:restriction>
              </xsd:simpleType>
```

```
            </xsd:element>

            <xsd:element name="data" type="xsd:hexBinary" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:unsignedShort" use="required" />
  </xsd:complexType>

  <xsd:complexType name="packetsToSendType" >
    <xsd:sequence>
      <xsd:element name="timeOffset" type="tns:limitedTimeType" />
      <xsd:element name="iterationCount" type="tns:retryCountType" minOccurs="0" />
      <xsd:element name="iterationInterval" type="tns:limitedTimeType" minOccurs="0"
          />
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:unsignedShort" use="required" />
  </xsd:complexType>



  <!--  ##################  testcase definition  ################### -->

  <xsd:element name="testcase">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="networkDefinition">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="networkParameters" type="tns:networkParametersType"
                  />
              <xsd:element name="requiredNetworkMember" type="
                  tns:requiredNetworkMemberType" maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="testcaseDefinition">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="testcaseParameters" type="tns:testcaseParametersType
                  " />
              <xsd:element name="nodeStateChange" type="tns:nodeStateChangeType"
                  minOccurs="0" maxOccurs="unbounded" />
              <xsd:element name="dataPacket" type="tns:dataPacketsType" minOccurs="0"
                   maxOccurs="unbounded" />
              <xsd:element name="packetToSend" type="tns:packetsToSendType" minOccurs
                  ="0" maxOccurs="unbounded" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>

    <!-- packets to be sent must be defined prior  -->
    <xsd:key name="packetId">
      <xsd:selector xpath="./testcaseDefinition/dataPacket" />
      <xsd:field xpath="@id" />
    </xsd:key>
    <xsd:keyref name="keyIdref" refer="tns:packetId">
      <xsd:selector xpath="./testcaseDefinition/packetToSend" />
      <xsd:field xpath="@id" />
    </xsd:keyref>

    <!-- defined packets must refer to existing nodes  -->
    <xsd:key name="macRef">
      <xsd:selector xpath="./networkDefinition/requiredNetworkMember" />
      <xsd:field xpath="addr" />
    </xsd:key>
    <xsd:keyref name="keyIdref2" refer="tns:macRef">
      <xsd:selector xpath="./testcaseDefinition/dataPacket/message" />
      <xsd:field xpath="addr" />
    </xsd:keyref>

    <!-- node state change packets must refer existing nodes -->
```

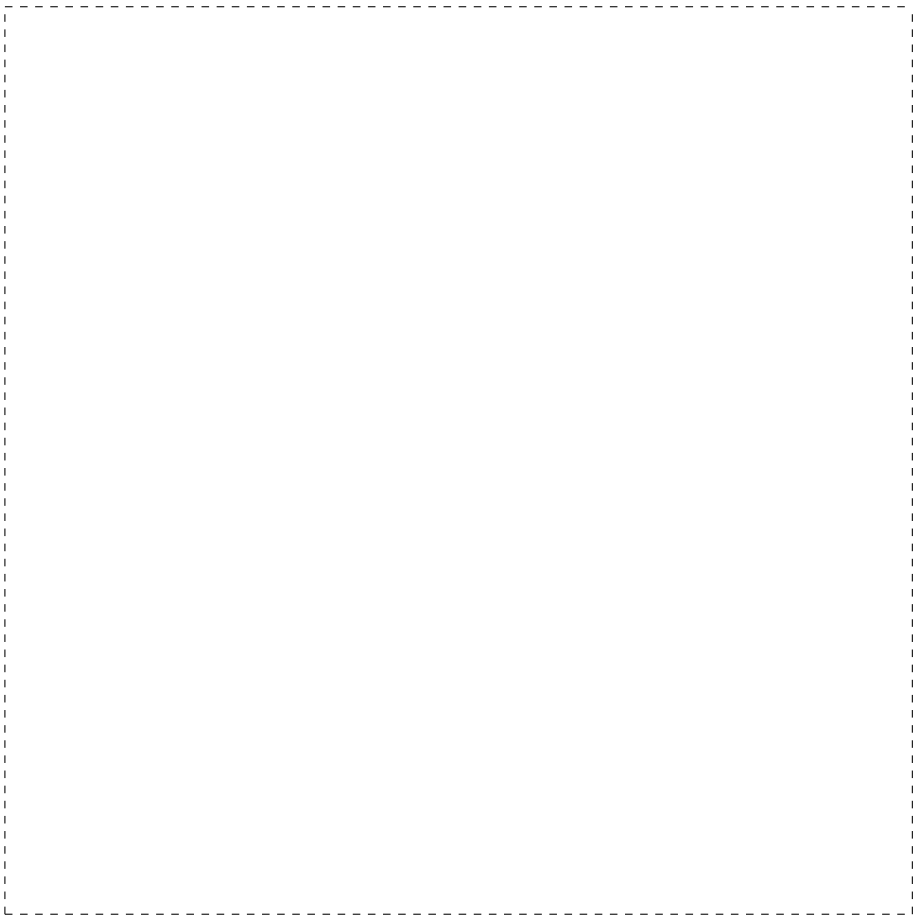
```
289     <xsd:key name="nscRef">
290       <xsd:selector xpath="./networkDefinition/requiredNetworkMember" />
291       <xsd:field xpath="addr" />
292     </xsd:key>
293     <xsd:keyref name="keyIdref3" refer="tns:nscRef">
294       <xsd:selector xpath="./testcaseDefinition/nodeStateChange" />
295       <xsd:field xpath="nodeAddress" />
296     </xsd:keyref>
297
298     <!--  each node has to be switched on and off only once -->
299     <xsd:unique name="nstUnique">
300       <xsd:selector xpath="./testcaseDefinition/nodeStateChange"/>
301       <xsd:field xpath="nodeAddress"/>
302       <xsd:field xpath="newState"/>
303     </xsd:unique>
304
305
306   </xsd:element>
307
308 </xsd:schema>
```

Listing B.1: Vollständige Schemadatei zur Beschreibung von Testfällen

C Beiliegendes Medium

Verzeichnisstruktur

```
|
+-- testserver/       - enthält den Quellcode des Testservers
|
+-- stack/            - enthält den modifizierte Quellcode
|                       des Atmel IEEE 802.15.4-Protokollstack
|
+-- diplomarbeit.pdf  - die vorliegende Arbeit in elektronischer Form.
```



Selbständigkeitserklärung

Ich versichere hiermit, dass ich meine Diplomarbeit mit dem Thema

> *Entwurf und prototypische Implementierung einer Test- und Visualisierungsplattform für dynamische IEEE-802.15.4- und Zigbee- Netzwerke*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Dresden, den 15. Dezember 2009

Daniel Müller