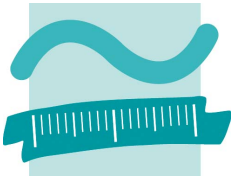


Beuth Hochschule für Technik Berlin

Fachbereich VII

Bachelorarbeit

Entwicklung und Test einer SNMP-basierten Automatisierungslösung für Sensorknoten in einem 6LoWPAN Funknetz

Development and evaluation of a SNMP-based automation solution for 6LoWPAN sensor nodes in a wireless network

eingereicht von:	Sven Zehl [S760383]
Betreuer:	Prof. Dipl.-Inform. Thomas Scheffler
Gutachter:	Prof. Dr.-Ing. Peter Gober
Abgabedatum:	14. August 2012

Kurzreferat

Im Verlauf dieser Arbeit wurde eine Automatisierungslösung zur Steuerung und Überwachung eines 230V Verbrauchers entwickelt. Die unterste Übertragungsschicht bildet ein Funknetz im *IEEE 802.15.4* Standard. Da für die Vermittlungsschicht *IPv6* eingesetzt wird, kommt zur Adaptierung beider Modelle das *6LoWPAN* Protokoll zum Einsatz. Die Steuerung des *Smart Object* wird durch einen 8 Bit Mikrocontroller der Firma Atmel und das darauf eingesetzte Betriebssystem *Contiki OS 2.6* übernommen. Auf der Anwendungsschicht, wird eine *SNMPv3* Implementierung ausgeführt, welche durch das integrierte *User Based Security Model*, Sicherheit durch Verschlüsselung und Authentifizierung bietet.
Im Laufe dieser Arbeit werden die einzelnen eingesetzten Technologien im ersten Schritt untersucht, um dann im weiteren Verlauf das daraus entwickelte Gesamtkonzept eingehend zu prüfen und auszuwerten.

Abstract

During this thesis, an automation solution for controlling and monitoring of a 230V consumer has been developed. The lowest transmission layer is represented by a radio network based on the *IEEE 802.15.4* standard. Because *IPv6* is seated in the network layer, the *6LoWPAN* protocol is used for adapting both models. The control of the *Smart Object* is handled by a 8 bit microcontroller of the company Atmel and the above running operating system *Contiki OS 2.6*. On the application layer, the *SNMPv3* with the *User Based Security Model* is implemented. This ensures security by offering encryption and authentication mechanisms.
In the course of this work the single used technologies are examined in the first step, then in the next step, the total concept developed from it, is thoroughly checked and evaluated.

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Berlin, den 13. August 2012

. . . . . .
(Unterschrift des Kandidaten)

Prof. Dipl.Inform. Thomas Scheffler

Beuth Hochschule für Technik Berlin
Luxemburger Str. 10
13353 Berlin

Telefon 0 30 / 4504 - 2648
E-Mail scheffler@beuth-hochschule.de

Berlin, den 12. April 2012

Aufgabenstellung der Abschlussarbeit von Herr Sven Zehl
"Entwicklung und Test einer SNMP-basierten Automatisierungslösung für Sensorknoten in einem 6LoWPAN Funknetz"

Aufgabenstellung:

Das Simple Network Management Protokoll (SNMP) wird aufgrund seines einfachen Aufbaus und der weiten Verbreitung von Implementierungen standardmäßig für die Verwaltung von Netzwerkressourcen eingesetzt.

Im Rahmen dieser Arbeit soll untersucht werden, wie gut sich SNMP für die Steuerung von *Smart Objects* in einem 6LoWPAN-Netzwerk nutzen lässt.

Mit Hilfe von existierenden Implementierungen für Managementstationen (z.B. Mibbrowser) soll es möglich sein, die Konfigurationsdaten des auf dem *Smart Object* installierten SNMP Agenten abzufragen und ggf. zu setzen. Die Kommunikation soll idealerweise sowohl über SNMPv1 als auch über SNMPv3 möglich sein. Die SNMPv3 Implementierung soll dabei die Verschlüsselung mit Pre-Shared Keys unterstützen.

Die Testergebnisse sind geeignet auszuwerten. Dazu ist ein geeigneter Testplan zu erstellen und ggf. ein Testtool zu entwerfen.

Folgende Teilaspekte sind in der Arbeit zu betrachten:

- Installation und ggf. Anpassung einer SNMP-Implementierung für Contiki 2.5.
- Test der SNMP Implementierung mit existierenden Managementstationen und wechselnden Netzwerkbedingungen.
- Bewertung der Komplexität und sonstiger Anforderungen der Lösung (Speicher-, Energieverbrauch, Rechenzeit, Übertragungsszeit, ...).
- Dokumentation der Entwicklung sowie möglicher Erweiterungen in einem Wiki.

Für die Bearbeitung der Aufgabenstellung werden gute Kenntnisse der Grundlagen der Netzwerktechnik vorausgesetzt. Der Bearbeiter muss sich in das Thema IPv6, SNMP und die Netzwerkkommunikation des Contiki-Betriebssystems einarbeiten.

LINKS:

SNMP `http://tools.ietf.org/html/rfc3411`

Mibbrowser: `http://ireasoning.com/mibbrowser.shtml`

Contiki-Doku: `http://www.sics.se/~adam/contiki/docs/`

Contiki-SNMP: `http://code.google.com/p/contiki-snmp/`

Thomas Scheffler

Inhaltsverzeichnis

1. Einleitung

Seit dem das Internet im Jahre 1969 aus dem *ARPA Net*[1] hervorging, wird es mittlerweile von etwa zwei Milliarden Menschen genutzt, was in etwa 30 Prozent der gesamten Weltbevölkerung entspricht, wobei die Wachstumsrate pro Jahr zwischen 2000 und 2011 etwa 500 Prozent betrug [MMG, 2012]. Doch all diese Zahlen sind nichts im Vergleich zu der Revolution welche dem Internet in den nächsten Jahren bevorsteht. Diese Revolution wird verursacht durch eine neu hinzugekommene Nutzergruppe. Kleine unscheinbare Geräte, die aufgrund der in den letzten Jahren stark fortgeschrittenen Entwicklung in den Bereichen der Mikroelektronik, Prozessortechnik und Kommunikationstechnik, immer kleiner, leistungsstärker und preisgünstiger wurden. Diese Geräte bekommen zunehmend die Möglichkeit das Internet zu nutzen. Sie sind praktisch überall integrierbar und ermöglichen es somit theoretisch jedem Gerät global mit anderen Geräten, Internetseiten oder Menschen zu kommunizieren. Diese Nutzung des Internets wird als das *Internet of Things* bezeichnet und die Nutzer dieses *Internet of Things* als *Smart Objects*. Experten gehen davon aus, dass die Anzahl dieser *Smart Objects* in den nächsten Jahren problemlos die Billionengrenze erreichen wird [Shelby and Bormann, 2009]. Da das bisherige *Internet Protokoll* der Version 4 prinzipiell keine freien Adressen mehr besitzt [IANA, 2012], kommt als Internet Protokoll des *Internet Of Things* nur das Internet Protokoll Version 6 in Frage[2]. Dieses bietet aufgrund des größeren Adressraums von prinzipiell 2^{128} Adressen[3] auch in Zukunft genügend Adressraum. Da solche Geräte auch beweglich sein können, ist es sinnvoll für die Netzanbindung eine Funktechnologie zu verwenden. Aus diesem Grund entwickelte die IEEE[4] den Funkstandard *802.15.4*[5], welcher es den *Smart Objekts* ermöglicht über kleine Datenpakete, sogenannte Frames über Funk miteinander zu kommunizieren. Da das globale Internet als überliegende Schicht das zuvor erwähnte Internetprotokoll zur Kommunikation nutzt, wurde mit der Einführung von *6LoWPAN*[6] eine Adaptionsschicht zwischen diesen beiden Protokollen geschaffen.
Durch diese Technologien und durch 8 Bit Betriebssysteme mit vollständigem Support dieser Technologien, wie zum Beispiel das Betriebssystem *Contiki OS* [Dunkels et al., 2004], ist es möglich kleine Mikroprozessoren mit beispielsweise nur 128 kByte ROM und lediglich 8 kByte RAM, internetfähig zu machen. Diese Geräte wie zum Beispiel das *AVR Zigbit Modul*[7], welches sogar über eine integrierte Antenne verfügt, liegt momentan in der Preisklasse von knapp 20 Euro. Durch den zukünftig ansteigenden Bedarf dieser Geräte und dem somit bevorstehenden Wandel vom derzeitigen Marktmodell des Oligopols zum Polypol, wird sich dieser Preis sehr wahrscheinlich so sehr verringern, dass in unmittelbarer Zukunft einem standardmäßigen Einbau in nahezu allen Geräten nichts entgehen stehen wird.

[1] Advanced Research Projects Agency Network
[2] [Deering and Hinden, 1998]
[3] $2^{128} \approx$ 340 Sextillionen Adressen
[4] Institute of Electrical and Electronics Engineers
[5] [IEEE, 2003, 2006, 2007]
[6] IPv6 over Low power Wireless Personal Area Network [Kushalnagar et al., 2007]
[7] siehe Anhang A.12.3

Zur Steuerung der *Smart Objekts* auf der obersten Schicht, sind viele Möglichkeiten denkbar, jedoch ist es sinnvoll bereits bestehende Entwicklungen in das *Internet of Things* zu migrieren. Hier kommt das *Simple Network Management Protokoll* (SNMP) zum Einsatz, es ist aufgrund seines einfachen Aufbaus weit verbreitet und ist de facto der Standard bei der Verwaltung von Netzwerkressourcen. In der neuen Version 3 des Simple Network Management Protokolls, wurde in puncto Sicherheit viel getan, wodurch es nach heutigen Maßstäben als sicheres Protokoll bezeichnet werden kann.
Diese Arbeit beschäftigt sich mit dem Zusammenspiel all dieser genannten Technologien und Entwicklungen. Und so ist das festgelegte Ziel die Entwicklung und Evaluierung eines *Smart Objekts* mit *SNMPv3* Unterstützung auf Basis des *802.15.4* Funknetzes, *6LoWPAN* als Adaptionsschicht und *Contiki OS* als Betriebssystem.

Das nun folgende Kapitel 2 behandelt kurz den Funkstandard *802.15.4* und die Adaptionsschicht *6LoWPAN*. Der anschließende Hauptteil des Kapitels untersucht detailliert das *Simple Network Management Protokoll* wobei der Schwerpunkt auf die Version 3 des Protokolls gelegt wird. Kapitel 3 geht auf das Betriebssystem *Contiki OS* ein und beschreibt die speziellen Funktionsweisen, die es ermöglichen mit 8 Bit Prozessoren einen vollständigen TCP/IP Protokollstapel zu nutzen. Im weiteren Verlauf dieses Kapitels wird das Programm *Contiki SNMP* der Jacobs Universität vorgestellt, welches einen ersten Ansatz einer vollständig umgesetzten SNMP Implementierung für *Smart Objects* darstellt. Nachdem diese Implementierung und deren Aufbau anhand des Quellcodes erläutert wurde, wird in Kapitel 4 das Ergebnis des praktischen Teils dieser Arbeit vorgestellt. Das *SNMPv3 6LoWPAN Socket*, ein *Smart Object*, dass es über *SNMPv3* auf Basis *Pre-Shared Keys* unter anderem ermöglicht einen 230V Verbraucher zu steuern. Die Entwicklung wird dann in Abschnitt 4.3 einigen Leistungstests sowie einem Funktionstestplan zur Garantie der Sicherheit unterzogen. Im letzten Kapitel 5 erfolgt schließlich eine Zusammenfassung der gesamten Arbeit.

2. Grundlagen

Auf den folgenden Seiten dieses Kapitels werden die Grundlagen, welche für das weitere Verständnis dieser Arbeit notwendig sind, besprochen. Zuerst wird der Standard IEEE[1] 802.15.4 kurz zusammengefasst. Es folgt die Adaptionsschicht zwischen 802.15.4 und IPv6[2], das 6LoWPAN[3] Protokoll. Dieses wird in Abschnitt 2.2 kurz beschrieben. Der Schwerpunkt dieses Kapitels richtet sich jedoch auf Abschnitt 2.3, hier wird das Simple Network Management Protokoll detailliert untersucht, wobei das Hauptaugenmerk auf die dritte Version gelegt wird.

2.1. IEEE 802.15.4

Die 4. Arbeitsgruppe der IEEE 802.15 definierte erstmals im Jahre 2003 den Standard 802.15.4, worin die Spezifikationen für den Physical Layer (PHY) und den Medium Acces Control Layer (MAC) innerhalb von Low-Power Wireless Personal Area Networks festgelegt wurden [IEEE, 2003]. Dieser Standard wurde im Jahr 2006 und 2007 nochmals überarbeitet und verbessert [IEEE, 2006, 2007]. Es wird in 802.15.4 lediglich der PHY Layer sowie der MAC Layer definiert, die überliegenden Schichten sind nicht festgelegt. Spezifikationen für die überliegende Schicht sind beispielsweise Zigbee[4] oder das in Kapitel 2.2 vorgestellte 6LoWPAN. IEEE 802.15.4 bietet Datenraten bis zu 250 kb/s und verwendet als Zugriffsverfahren CSMA/CA[5]. Die Schlüsselfunktionen dieses Standards sind niedrige Komplexität, niedrige Kosten und niedriger Energieverbrauch. IEEE 802.15.4 ist momentan die Standard Technologie im Bereich der Sensornetzwerke. Dieses Kapitel bietet einen kurzen Überblick.

2.1.1. Geräte Typen

Innerhalb der IEEE 802.15.4 Standards sind zwei Gerätetypen spezifiziert, einmal das *Full Function Device (FFD)* und das *Reduced Function Device (RFD)*. Während das FFD die volle MAC Funktionalität besitzt und als PAN[6] Coordinator eingesetzt werden kann, bietet das RFD nur eine reduzierte MAC Funktionalität und kann nur als einfaches Netzwerkgerät operieren. Ein FFD kann grundsätzlich mit einem anderen FFD sowie einem RFD kommunizieren, während ein RFD nur mit einem FFD Daten austauschen kann.

2.1.2. Netzwerk Topologien

Innerhalb [IEEE, 2003] sind zwei Netzwerk Topologien definiert, die *Stern Topologie* und die *Peer to Peer Topologie*. Bei der Stern Topologie in Abbildung 2.1 bildet ein FFD den

[1] Institute of Electrical and Electronics Engineers
[2] Internet Protokoll Version 6
[3] IPv6 over Low power Wireless Personal Area Network
[4] `http://www.zigbee.org`
[5] Carrier Sense Multiple Access/Collision Avoidance
[6] Personal Area Network

sogenannten *PAN Coordinator*. Dieser fungiert wie eine Art Gateway, also als zentraler Knotenpunkt. Jede Verbindung muss über den PAN Coordinator aufgebaut werden, egal ob es sich beim verbindenden Gerät um ein FFD oder ein RFD handelt. Die Stern Topologie wird für kleine Netzwerke empfohlen, bei denen die Latenzzeiten sehr klein sein müssen.

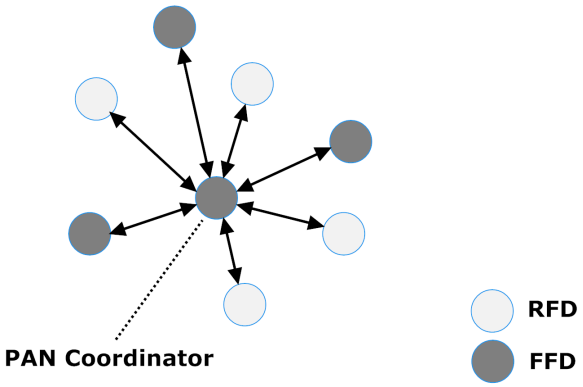


Abbildung 2.1.: Stern Topologie

Die zweite mögliche Topologie, die Peer to Peer Topologie, enthält ebenso einen PAN Coordinator. Jedoch ist bei dieser Topologieform eine Kommunikation der Geräte untereinander ebenso möglich. Es ergibt sich somit die Möglichkeit sehr viel komplexere, vermaschte Netzwerke, auch auf Basis von Multi-Hop Verbindungen, aufzubauen die auch selbstheilend und selbstorganisierend operieren können, wobei das Routing den überliegenden Schichten überlassen wird, siehe dazu Kapitel 2.2). In Abbildung 2.2 ist der Aufbau einer solchen Topologie zu sehen. Die Peer to Peer Topologie ist bei großen Netzwerken, bei denen die Latenzzeiten keine große Rolle spielen, zu wählen.

Jeder Teilnehmer besitzt eine globale 64-Bit Adresse, welche der 64-Bit Adresse entspricht, die auch Geräte in IEEE 802.3[7] und IEEE 802.11[8] Netzwerken nutzen. Um jedoch den Paketoverhead so gering wie möglich zu halten, tauschen bei IEEE 802.15.4 die Teilnehmer beim Betreten eines Personal Area Networks diese 64 Bit Adresse gegen eine verkürzte 16-Bit lokale Adresse. Die Adressvergabe übernimmt hierbei der PAN Coordinator. Außerdem wird zur Abtrennung des Netzwerks von anderen Netzwerken innerhalb des Funkempfangsbereichs der sogenannte *PAN Identifier* verwendet, welcher vom PAN Coordinator ausgewählt wird.

2.1.3. MAC Layer

Zur Übertragung von Daten werden Frames genutzt, wobei dazu vier Rahmentypen zur Verfügung stehen, *Beacon Frames, Command Frames, Data Frames* und *Acknowledgement Frames*. Der Acknowledgement Frame kann als Bestätigung für versendete Frames genutzt werden. Ein Beacon Frame kann nur vom PAN Coordinator versendet werden und besitzt des weiteren eine spezielle Funktion: die Bildung eines *Super Frames*. Hierzu markiert jeweils ein Beacon Frame den Beginn und ein weiterer das Ende des Super Frames. Super Frames (Abbildung 2.3) werden eingesetzt, wenn der *beacon-enabled* Modus genutzt wird. Sie ermöglichen

[7] Ethernet
[8] WLAN

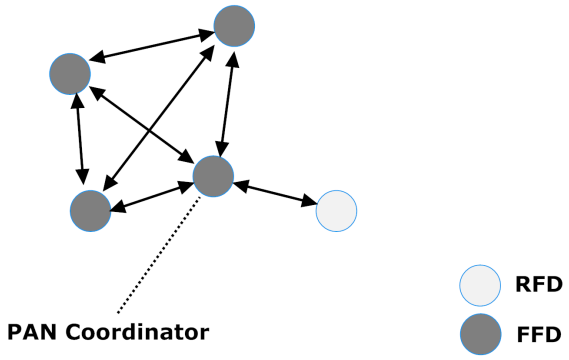


Abbildung 2.2.: Peer to Peer Topologie

es den Geräten eine gewisse Zeit in eine Art Ruhemodus zu schalten, um Energie zu sparen. Außerdem enthalten Super Frames eine *Contention Access Period* (CAP) wo der Zugriff auf das Medium über einen *CSMA/CA* Algorithmus mit Backoff Zeit geregelt wird, sowie eine *Contention Free Period*. Innerhalb der *Contention Free Period* hat der PAN Coordinator die Möglichkeit bestimmten Geräten einen Zeitschlitz zu reservieren.

Außer dem *beacon-enabled* Modus gibt es noch einen zweiten Modus, den *non beacon-enabled* Modus, auch *unslotted Mode* gennant. Im *non beacon-enabled* Modus wird ausschließlich das *CSMA/CA* Verfahren für den Kanalzugriff verwendet.

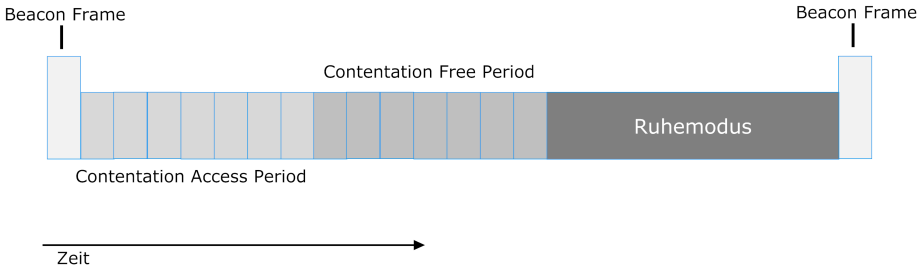


Abbildung 2.3.: Superframe Aufbau

Noch nicht besprochen, wurden der MAC Frame, welcher zu Kontroll- und Fehlerbehebungszwecken genutzt wird und der Data Frame. Der Data Frame enthält die Nutzdaten, wobei hier beachtet werden muss, dass die allgemeine Rahmengröße auf 127 Byte beschränkt ist. Abzüglich dem Frame Header bleiben für die höheren Schichten maximal 102 Byte an Nutzdaten. Und das bei deaktivierter Sicherheit.

2.1.4. Sicherheit

Der Standard 802.15.4 definiert Funktionen für *Access Control*, *Data Encryption*, *Frame Integrity* und *Sequentiell Freshness*. Für die *Access Control* Funktion ist eine Zugriffsliste vorgesehen worin vertrauenswürdige und nicht vertrauenswürdige Geräte verwaltet werden. Zur Regelung der *Data Encryption* Funktion ist ein symmetrisches Verschlüsselungsverfahren mit Pre-Shared Key festgelegt. Wobei der Schlüsselaustausch in die Hände der höheren Schichten

gelegt wird. Zur Umsetzung der *Frame Integrity* ist ein *CBC-MAC* Verfahren definiert. Zu guter Letzt wird mit der *Sequentiell Freshness* Funktion mithilfe einer *freshness* Variable die Aktualität der Nachricht überprüft.

→ Weiterführende Information zu 804.15.4 sind in den IEEE Standards [IEEE, 2003, 2006, 2007] zu finden, eine gute Übersicht bietet [Gutierrez et al., 2011].

2.2. 6LoWPAN

Die kleine Paketgröße von maximal 102 Byte[9] und minimal 81 Byte[10] Nutzdaten in 802.15.4 Netzen ermöglicht es nicht die Standard IPv6 Anforderungen aus RFC2460 [Deering and Hinden, 1998], worin die minimale MTU Größe 1280 Bytes beträgt, zu erfüllen. Da IPv6 mit dem großen Adressraum, dem Autokonfigurationsvorgang sowie seiner freien Verfügbarkeit und der weiten Verbreitung wie geschaffen ist, um die Anforderungen an ein *Internet of Things* zu ermöglichen, musste eine Adaptionsschicht definiert werden, welche Fragmentierung sowie Komprimierung ermöglicht. Die Anforderungen an diese Adaptionsschicht wurden in RFC 4919 [Kushalnagar et al., 2007] festgelegt. In RFC 4944 [Montenegro et al., 2007] wurden diese Anforderungen umgesetzt. Außerdem wurde in RFC 6282 [Hui and Thubert, 2011] ein neues Header Kompressionsformat festgelegt, welches das in Kompressionsformat in [Montenegro et al., 2007] ersetzen soll.
Dieses Kapitel bietet einen kurzen Überblick über die 6LoWPAN Umsetzung in [Montenegro et al., 2007, Hui and Thubert, 2011]

2.2.1. Adressierung

IEEE 802.15.4, siehe dazu Kapitel 2.1, definiert zwei Adresstypen, einmal die erweiterte 64 Bit Adresse, sowie eine 16-Bit gekürzte Adresse in Verbindung mit der PAN-ID. In RFC 4944 werden beide Adresstypen unterstützt um den IPv6 Stateless Address Autokonfigurationsvorgang durchzuführen. Wird die 64 Bit IEEE EUI-64 genutzt, so wird der Interface Identifier, definiert in [Hinden and Deering, 2006], wie in RFC 2464 [Crawford, 1998] für Ethernet beschrieben, auch für 802.15.4 Netze auf diese Art gebildet. Bei Verwendung der kurzen 16 Bit 802.15.4 Adresse wird mithilfe der PAN-ID zuerst eine 48 Bit Adresse geformt, welche dann wieder nach [Crawford, 1998] zu einem Interface Identifier umgewandelt wird. Der generierte Interface Identifier wird anschließend mit dem Präfix `FE80::/64` versehen und bildet so die Link lokale IPv6 Adresse (siehe Abbildung 2.4).

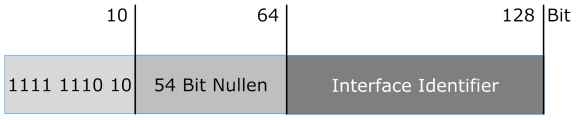


Abbildung 2.4.: Bildung der Link lokalen IPv6 Adresse

[9] bei deaktivierter Sicherheit
[10] mit aktivierter Sicherheit (AES-CCM-64)

2.2.2. Adaption Layer

Um die Anforderungen in RFC 4919 [Kushalnagar et al., 2007] umzusetzen, wurden in RFC 4919 vier Encapsulation Header sowie diverse Komprimierungs-Header[11] festgelegt. Der *Dispatch Header* ist immer vorhanden und identifiziert den nachfolgenden Datenteil, dieser kann entweder mit einem Komprimierungs-Header oder mit dem IPv6 Paket beginnen.
Der zweite festgelegte Header ist der *Mesh Addressing Header*. Er ermöglicht es innerhalb des 802.15.4 Netzes vermaschtes Routing zu verwenden[12]. Zur weiteren Umsetzung übernehmen FFDs (siehe Kapitel 2.1) die Weiterleitungsfunktion. Der Mesh Addressing Header wird immer dann eingesetzt, wenn keine direkte Erreichbarkeit mit dem Empfänger besteht.
Ein sehr wichtiger Header ist der *Fragmentation Header*, er findet immer dann Verwendung, wenn ein IPv6 Paket nicht in einen einzigen 802.15.4 Frame passt. Jedes 802.15.4 Paket enthält dann einen Fragmentation Header. Im ersten Fragmentation Header muss die Gesamt-IP-Paketlänge enthalten sein. In den folgenden Fragementation Headern wird dies empfohlen. Jedes 802.15.4 Paket das ein IP Fragment enthält, hat außerdem eine bestimmte Nummer im Header Feld *datagram_tag* gespeichert, wodurch die Fragmente einander zugeordnet werden können. Des Weiteren wird immer der jeweilige Offset des IP Fragments im Fragmentation Header mitgeführt.
Außerdem wurde ein *Broadcast Header* definiert, er ist eine Art Zusatzfunktion des Mesh Addressing Headers und ermöglicht es Broadcast Nachrichten zu versenden. Durch das Übermitteln einer Sequenznummer können Nachrichtenduplikate ermittelt werden.
Um die Paketdatenlänge weiter zu verringern, wurden in RFC 4919 zwei Headerkomprimierungsverfahren festgelegt. Das HC1 Verfahren zur Komprimierung des IPv6 Headers und das HC2 Verfahren zur Komprimierung des UDP/TCP und ICMP Headers. *HC1* eignet sich jedoch nur für IPv6 Header mit lokalen Adressen und *HC2* bietet keine Komprimierung der UDP Checksumme. Deshalb wurden in RFC 6282 [Hui and Thubert, 2011] zwei neue Kompressionsmechanismen *IPHC* und *NHC* festgelegt. *IPHC* ermöglicht die Komprimierung von lokalen und globalen IPv6 Adressen, sowie von Multicast IPv6 Adressen. Durch den weiteren Einsatz von *NHC* statt *HC2* können außerdem nicht nur UDP/TCP/ICMP Header, sondern auch IPv6 Extension-Header, komprimiert werden. Außerdem bietet *NHC* die Möglichkeit das UDP Checksummen Feld zu komprimieren, wodurch zwei weitere Byte eingespart werden können.

2.3. Simple Network Management Protokoll

Der wohl bekannteste Kommunikationsmechanismus aus dem Bereich des Netzwerkmanagements ist das *Simple Network Management Protcol (SNMP)*, es erlaubt sowohl das Überwachen sowie auch das Konfigurieren von Netzwerkkomponenten. Bis zum Jahre 1988 gab es viele unterschiedliche Standards an Netzwerkprotokollen zum Management der Netzwerkgeräte, neben SNMP z.B. auch HEMS[13]. Jedoch wurde im Februar 1988, auf einem Meeting des IAB[14], die Empfehlung verfasst, SNMP weiterzuentwickeln [Cerf, 1988, Bless et al., 2005]. Es folgte im August 1988 der Standard SNMPv1, definiert in [Case et al., 1988]. Diese Version

[11] Erweitert durch RFC 6282 [Hui and Thubert, 2011]
[12] Der 802.15.4 Standards enthält keine Definition für Routing-Algorithmen, es wird dort auf überliegende Schichten verwiesen.
[13] High-Level Entity Management Systems
[14] Internet Architecture Board

ist heutzutage eigentlich schon historisch, wird jedoch immer noch von einigen Geräteproduzenten verwendet. Die größte Schwachstelle von SNMPv1 ist das Sicherheitskonzept. Es gibt zwar eine passwortähnliche Funktion, den sogenannten Community String, jedoch wird dieser im Klartext übertragen und stellt somit keine wirkliche Sicherheit dar [siehe Case et al., 1990]. Um dieses Problem zu beheben, wurden 1992 Sicherheitsmechanismen in [Case et al., 1988] festgelegt. Diese wurden jedoch als optional gekennzeichnet und deshalb kaum implementiert. Mit der Einführung von SNMPv2, 1993 kamen einige neue Protokolloperationen hinzu (siehe Abschnitt 2.3.7), jedoch leider keine weiteren Sicherheitsmechanismen. Erst mit der Einführung von SNMPv3 im Jahre 2002 und damit den RFCs3410-3418[15] wurde der derzeitig gültige Standard eingeführt. SNMPv3 enthält das User-Based Security Model (USM), welches Schutz gegen Modifikation der Daten, des Absenders sowie der Nachrichtenreihenfolge und Maßnahmen gegen Abhören bietet [Blumenthal and Wijnen, 2002].

In den folgenden Abschnitten wird SNMP detailliert erklärt. Das Hauptaugenmerk wird jedoch auf SNMPv3 gelegt, da dies die oben genannten Sicherheitsmerkmale bietet.

2.3.1. Transportmechanismen

Die grundlegende Kommunikationsform von SNMP basiert auf zwei verschiedenen Kommunikationswegen:

bidirektional: Hier wird das Frage-Antwort Prinzip bzw. Befehl-Bestätigung Prinzip angewandt, die Managementstation stellt dem SNMP Agenten eine Frage bzw. Aufgabe und dieser antwortet. Dies ermöglicht sowohl das Überwachen, wie auch das Verwalten von Komponenten.(sogenannte Set und Get Befehle, siehe dazu Abschnitt 2.3.7)[Schwenkler, 2005].

unidirektional: Die zweite Kommunikationsform kann ausschließlich von der überwachten Komponente, also dem SNMP Agent, zur Management Station durchgeführt werden. Dieser Kommunikationsweg wurde eingerichtet, damit die überwachte Komponente die Möglichkeit hat die Management Station über besondere Ereignisse zu informieren (sogenannte Trap Befehle, siehe dazu Abschnitt 2.3.7). [Schwenkler, 2005]

Als Transportprotokoll verwendet SNMP meistens das UDP Protokoll, außerdem wird für die bidirektionale Kommunikation der Port 161 und für die unidirektionale Kommunikation der Port 162 genutzt. SNMP ist jedoch nicht an die Transportschicht gebunden, so ist als Transportprotokoll z.B. auch Apple Talk oder IPX[16] möglich [Presuhn, 2002b].

2.3.2. Structure of Management Information und Management Information Base

Das SNMP Protokoll an sich, legt nicht fest welche Informationen ein SNMP Agent zur Verfügung stellt. Dies kann individuell vom Hersteller festgelegt werden. Damit nun dem SNMP Manager bekannt ist welche Daten auf dem SNMP Agent zur Verfügung stehen, wird eine

[15] [Case et al., 2002b, Harrington et al., 2002, Case et al., 2002a, Levi et al., 2002, Blumenthal and Wijnen, 2002, Wijnen et al., 2002, Presuhn, 2002a,b,c]

[16] Internetwork Packet Exchange

Management Information Base (MIB) festgelegt.
In dieser MIB befinden sich die sogenannten Managed Objects, also die Objekte, die abgerufen bzw. abgeändert werden können. Der Aufbau einer solchen MIB wird durch die sogenannte Structure of Management Information (SMI) festgelegt [McCloghrie et al., 1999a, Rose and McCloghrie, 1990]. Grundsätzlich werden Managed Objects durch drei Eigenschaften definiert:

Name: Der Objekt Name wird bei SNMP Object Identifier (OID) genannt. Dieser kann entweder numerisch oder mit der Wort-Punkt Darstellung angegeben werden.

Typ und Syntax: Die Datentypen, die ein Managed Object annehmen kann, sind in der Abstract Syntax Notation One (ASN.1) festgelegt. Da ein Agent, welcher mit Java geschrieben wurde, auch mit einem Manager, der in C programmiert wurde, kommunizieren muss, musste eine Norm festgelegt werden, welche die Probleme mit unterschiedlich definierten Datentypen löst.

Encoding: Selbst durch die Festlegung der Datentypen mit ASN.1 gibt es immer noch das Problem mit der Übertragung. Auch hier muss eine Norm festgelegt werden, mit der die zu übertragenden Daten codiert werden. ASN.1 beinhaltet hierfür die sogenannten Basic Encoding Rules (BER). Diese codieren die Daten mit festgelegten Regeln zu einem Oktett String, welcher dann über das Medium übertragen werden kann.

Der Ursprung der OID kommt aus dem baumartigen Aufbau der MIB. Die MIB wird von dem MIB Manager wie eine Art "Telefonbuch" genutzt. Um das gewünschte Objekt erreichen zu können, muss eine bestimmte Nummernkombination gewählt werden. Diese Nummernkombination wird OID genannt. Die MIB enthält weiterhin auch Informationen wie den Namen, den Datentyp sowie Informationen zu Lese/Schreibzugriff Berechtigungen. In Abbildung 2.5 ist der Auszug einer MIB abgebildet. Firmen und Institutionen verwenden immer den Unterzweig *iso.org.internet.private.enterprises*. In diesem Unterzweig kann kostenlos bei der IANA ein firmenspezifischer Unterzweig reserviert werden. Zum Beispiel hat die Beuth Hochschule Berlin den Unterzweig 22109[17]. Wird im nebenstehenden MIB Beispiel nun die OID für den Schaltzustand der 6LoWPAN Steckdose betrachtet, so kann man deutlich ablesen, dass diese *1.3.6.1.4.1.22109.100.600.1.0* lautet. Die Null am Ende der OID nennt sich Instance Value, sie sind der Index innerhalb des Managed Object. Es gibt zwei Objekttypen in MIBs, skalare und tabulare. Skalare Managed Objects beinhalten nur einen Wert, während tabulare Managed Objects mehrere Werte beinhalten können. Die Instance Value bei skalaren Managed Objects ist immer 0, bei tabularen Managed Objects entspricht sie dem Tabellenindex in dem der gesuchte Wert liegt.
Tabellen werden innerhalb der MIB auch durch die Baumstruktur in Abbildung 2.5 abgebildet. Jedoch erhält nicht nur jedes Tabellenfeld eine OID, sondern es werden für deren Darstellung noch weitere Sub-OIDs benötigt. Zuerst wird die Wurzel der Tabelle durch eine OID gekennzeichnet. Innerhalb dieser Wurzel OID befindet sich eine weitere Sub-OID, worin die Definition der darunterliegenden Spalten gekennzeichnet wird. Eine Stufe weiter sitzt die eigentliche Spalten Sub-OID. Und in der vierten Hierarchiestufe ist dann das eigentliche Objekt zu finden.
Zur Verdeutlichung dieser Darstellung, wird die Interface Tabelle, welche innerhalb der MIB2

[17] Alle bisher reservierten Unterzweige sind unter `http://www.iana.org/assignments/enterprise-numbers` einzusehen

definiert ist, erklärt. Dazu muss nun Tabelle 2.1 betrachtet werden. Sie zeigt den Auschnitt einer im Unterzweig MIB2 vorhanden Interface Tabelle. Dieser Tabellenausschnitt ist ebenfalls in Abbildung 2.5 zu sehen. Die Wurzel OID der Tabelle bildet hier *ifTable* mit der Sub OID 2, somit ist die Wurzel OID der Tabelle *ifTable*: 1.3.6.1.2.1.2.**2**. Die Spaltendefinition bildet *ifEntry* mit der Sub OID 1. Soll also in der Tabelle auf eines der Objekte des Typs *ifEntry* zugegriffen werden, so muss die OID 1.3.6.1.2.1.2.**2.1** gewählt werden. Da aber mit dieser Nummernkombination noch nicht bekannt ist in welcher Spalte und Zeile das ausgewählte Objekt liegt, muss zuerst die Spalte und dann die Zeile des gesuchten Objekts angegeben werden. Soll also auf die Beschreibung (*IfDescr*) des 3. Interfaces zugegriffen werden, so muss die OID mit der diese aufgerufen werden kann 1.3.6.1.2.1.2.**2.1.2.3** lauten.

→OIDs können aus bis zu 128 Sub-OIDs bestehen.[Presuhn, 2002a]

Tabelle 2.1.: Ausschnitt der Interfaces Tabelle aus der MIB2

ifTable (2)	
ifEntry (1)	
ifIndex (1)	*IfDescr (2)*
Zelle (1)	Zelle (1)
Zelle (2)	Zelle (2)
Zelle (3)	Zelle (3)

2.3.3. ASN.1

ASN.1 (Abstract Syntax Notation One) bietet zwei Kategorien von Datentypen, primitive und komplexe. Zu den primitive Datentypen zählen Integer, Octet, String, Null, Boolean und Object Identifier, während komplexe Datentypen eine Zusammenfassung von primitiven Datentypen darstellen. Der wichtigste komplexe Datentyp für SNMP ist Sequence. Sequence ist eine einfache Liste von Datentypen, wobei jedes Feld einen anderen Datentyp haben kann. In den RFCs 1155 [Rose and McCloghrie, 1990] und 2578 [McCloghrie et al., 1999a] wurden spezielle SNMP Datentypen festgelegt. SMIv1 und SMIv2 werden beide noch aktuell von einigen Herstellern verwendet, unterscheiden sich jedoch in den vorhandenen Datentypen. Tabelle 2.2 zeigt die möglichen Datentypen, die durch SMIv1 bzw. SMIv2 definiert sind. Da SNMP abwärtskompatibel ist, unterstützt SNMPv3 immer noch SMIv1, jedoch sollte bei neuen Implementierungen immer SMIv2 verwendet werden. Weiterführende Informationen zu ASN.1 sind unter [Dubuisson, 2000, Larmouth, 1999] zu finden.

2.3.4. BER

Mit den in Tabelle 2.2 gezeigten ASN.1 Datentypen lassen sich nun sämtliche Managed Objects darstellen. Jedoch muss bei der Übertragung auch der Anfang bzw. das Ende des Datentyps festgelegt werden. Hierfür werden die Basic Encoding Rules (BER) eingesetzt. Es gibt hierfür zwei Möglichkeiten, *definite-length* oder *indefinite-length*. Bei der Verwendung von *definite-length* besteht ein BER Objekt aus drei Teilen: Type, Length und Data. Type spezifiziert den Daten Typ repräsentiert durch ein Byte mit speziellem Identifier (siehe dazu Tabelle 2.3). Length enthält die Länge des folgenden Datenblocks.

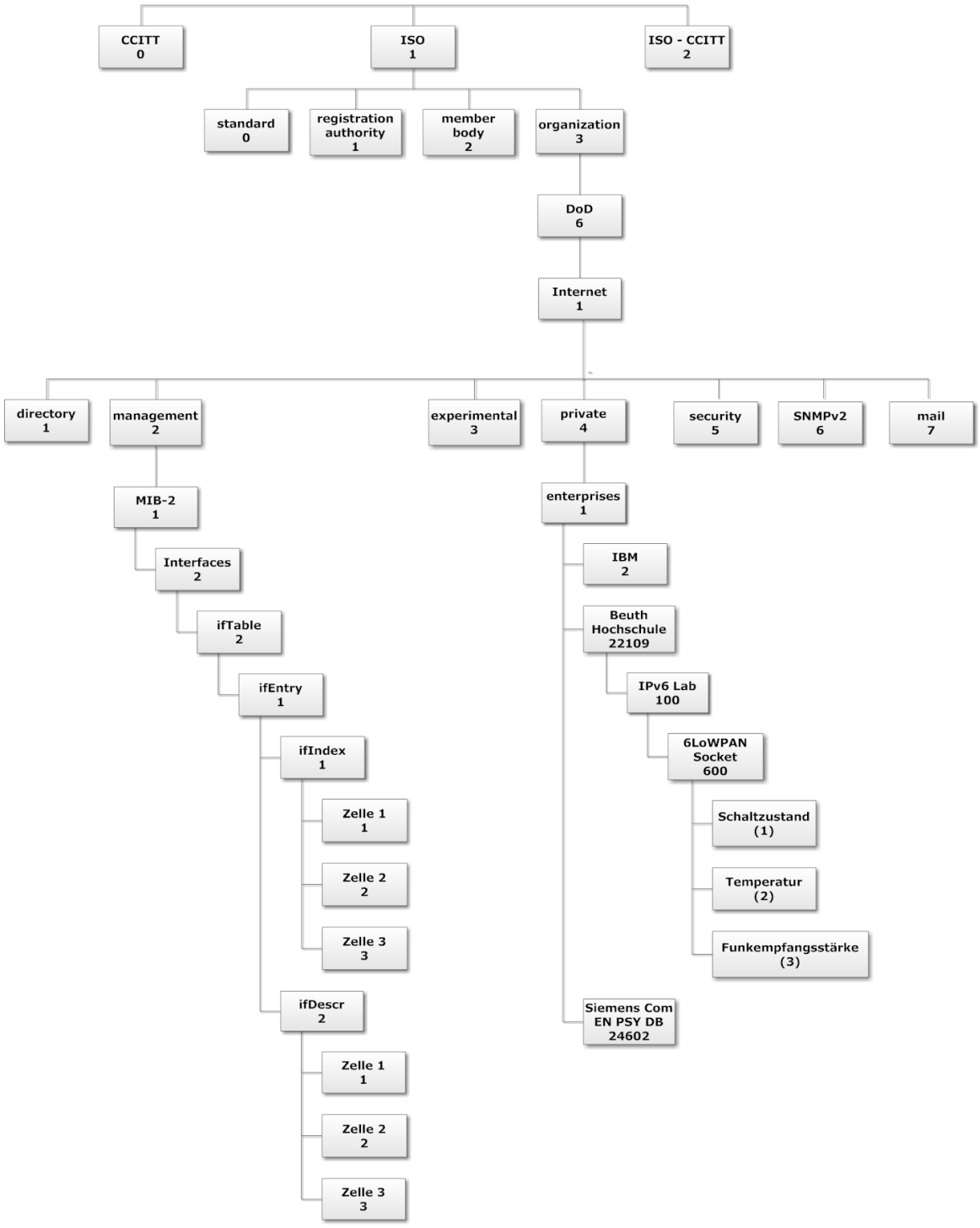


Abbildung 2.5.: SNMP MIB Baumstruktur

Tabelle 2.2.: Datentypen definiert durch SMIv1 und SMIv2 [McCloghrie et al., 1999a, Rose and McCloghrie, 1990]

Base Type	*SMIv1*	*SMIv2*	**Base Type**	*SMIv1*	*SMIv2*
INTEGER	ja	ja	TimeTicks	ja	ja
Integer32	nein	ja	OCTET STRING	ja	ja
Unsigned32	nein	ja	OBJECT IDENTIFIER	ja	ja
Gauge	nein	ja	IpAddress	ja	nein
Gauge32	nein	ja	NetworkAddress	ja	nein
Counter	ja	nein	Opaque	ja	ja
Counter32	ja	nein	BITS	nein	ja
Counter64	nein	ja			

Tabelle 2.3.: Datentypen und BER Identifier (Auswahl) [Bruey, 2005]

primitiver Datentyp	**Identifier**	**komplexer Datentyp**	**Identifier**
INTEGER	0x02	Sequence	0x30
Octet String	0x04	GetRequestPDU	0xA0
Null	0x05	ResponsePDU	0xA2
Object Identifier	0x06	SetRequest PDU	0xA3

Und Data enthält die eigentlichen Nutzdaten (Beispiel siehe Abbildung 2.6). Wird *indefinite-length* Encoding verwendet, so steht innerhalb des Feldes Length der Wert 0x80. Das Ende des Datenblocks muss dann aber durch ein weiteres Feld am Ende des Datenblocks markiert werden. Dieses Feld besteht aus zwei Byte mit dem Inhalt 0x0000. (siehe Abbildung 2.7) Unter SNMP wird jedoch immer das definite-length Encoding verwendet.

Abbildung 2.6.: BER codiertes Feld (primitiver Datentyp definite-length)

Als komplexe Datenfelder werden Datenfelder bezeichnet, die aus mehreren definite length codierten Datenfeldern bestehen. Ein Beispiel hierfür zeigt Abbildung 2.8. Die einzelnen Datenfelder werden dazu ineinander verschachtelt. So wird das Feld *Data* des ersten Datenfelds mit dem zweiten Datenfeld gefüllt usw.. Des Weiteren gibt es noch zwei spezielle Basic Encoding Rules, die zu beachten sind. Beide gelten im Zusammenhang mit der Codierung von Object Identifiern. So werden die ersten beiden Nummern der OID, hier genannt $(x.y)$ als ein Wert nach der Formel $(40 \cdot x) + y$ codiert [ITU, 2002]. Da die ersten beiden Nummern in SNMP eigentlich immer 1.3. lauten, bedeutet das also $(40 \cdot 1) + 3 = 43$. Wird 43_{10} hexadezimal codiert so erhält man $2B_{16}$. Alle folgenden Nummern werden jeweils einzeln mit einem Byte codiert.

Doch genau hier entsteht ein Problem, da ein Byte nur Zahlen von 0-255 fassen kann, Sub-OIDs aber einen Wert bis zu 2^{31} annehmen können [Presuhn, 2002a], z.B. lautet der

Abbildung 2.7.: BER codiertes Feld (primitiver Datentyp indefinite-length)

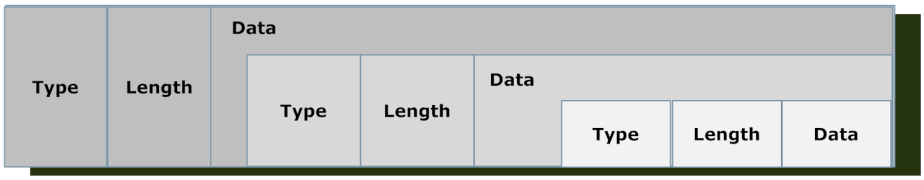


Abbildung 2.8.: BER codiertes Feld (komplexer Datentyp)

von der IANA vergebene *enterprises* Unterzweig der Beuth-Hochschule Berlin 22109, muss eine Regelung getroffen werden, um mehr als ein Byte verwenden zu können. Deshalb gibt es in [ITU, 2002] eine Regelung für dieses Problem. Sie besagt, dass nur die niederwertigsten sieben Bit eines Bytes verwendet werden um Daten zu speichern. Das höchstwertigste Bit wird verwendet, um anzuzeigen ob die momentane Zahl mehr als ein Byte nutzt (High bedeutet weitere Bytes folgen). Dies bedeutet jedoch, dass hierdurch nur noch sieben Bit zur Speicherung von Daten zur Verfügung stehen, also nur noch Zahlenwerte von 0-127. Zur Veranschaulichung wird diese Regel nun auf den Unterzweig der BHT Berlin angewendet. In Abbildung 2.9 ist der Ablauf dargestellt. Die Dezimalzahl 22109 wird zuerst in Binärform gewandelt und anschließend in sieben Bit Stücke aufgeteilt. Anschließend wird beim ersten und zweiten Byte das höchstwertigtste Bit auf 1 gesetzt und die restlichen Bits mit den zuvor berechneten Bits aufgefüllt. Die folgende Umrechnung in die hexadezimale Schreibweise ist optional.

22109_{10}	$\Longrightarrow$	01	0101100	1011101_2
		$\downarrow$	$\downarrow$	$\downarrow$
		1.Byte	2.Byte	3. Byte
		$\downarrow$	$\downarrow$	$\downarrow$
		10000001_2	10101100_2	01011101_2
		$\downarrow$	$\downarrow$	$\downarrow$
		81_{16}	AC_{16}	$5D_{16}$

Abbildung 2.9.: BER Codierung Sub-OID 22109

2.3.5. Nachrichtenformat

Mit dem Wissen aus den vorherigen Abschnitten lässt sich nun schon ein komplettes, funktionsfähiges SNMPv1 bzw. SNMPv2 Paket aufbauen. In Abbildung 2.10 ist das Nachrichtenformat eines SNMPv1 bzw. SNMPv2 Pakets zu sehen. Es ist gut zu erkennen, dass ein SNMPv1/2 Paket prinzipiell aus einem komplexen Datentyp *Sequence* besteht, welcher andere *komplexe* und *primitive* Datentypen enthält. Deshalb lässt sich durch das Umwandeln

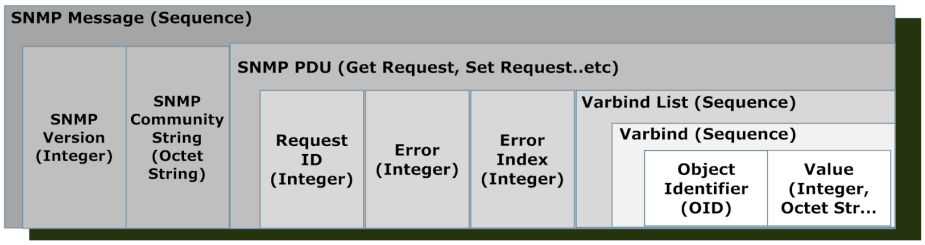


Abbildung 2.10.: SNMPv1 bzw. SNMPv2 Nachricht

jedes Feldes mithilfe der ASN.1 Norm und der anschließenden Codierung durch die BER das fertige SNMP Paket in Abbildung 2.11 erzeugen.

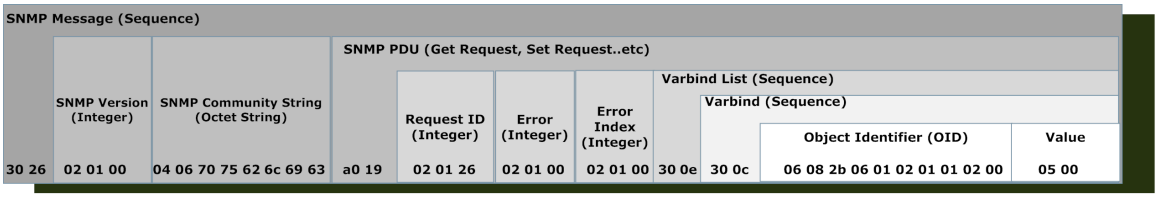


Abbildung 2.11.: BER codierte SNMPv1 Nachricht, Byte Stream in hexadezimaler Darstellung

Man erkennt, das Paket wird mit einer *Sequence* begonnen (Vergleich Tabelle 2.3), *Type Feld = 0x30 = Sequence*, *Length Feld = 0x26 = 38 Byte* Gesamtlänge des Datenfelds innerhalb des *Sequence* Datentyps, also 40 Byte Gesamtpaketlänge. Das nächste Feld im Paket ist die SNMP Version, hier ist Das *Type Feld 0x02*, also Datentyp *Integer* (Vergleich Tabelle 2.3), *Length Feld = 0x01*, also ein Byte das anschließende Datenfeld ist *0x00*, das bedeutet SNMPv1. So kann also Schritt für Schritt jedes Feld decodiert bzw. codiert werden.
Noch zu erklären sind die Felder *Request ID*, *Error* und *Error Index*. Das Feld *Request ID* beinhaltet eine Referenznummer, die es dem SNMP Manager ermöglicht, ankommende Antworten zu den versendeten Anfragen zuzuordnen. Das nun folgende *Error Feld* enthält im Falle eines Fehlers einen Fehlercode, der mithilfe von Tabelle 2.4 zugeordnet werden kann. *Error Index* enthält nach dem Auftreten eines Fehlers, einen Zeiger auf das verantwortliche Objekt.
Die nun folgende *Sequence*, *Varbind List*, ist ein komplexer Datentyp, welche die Objekte (Managed Objects) mit dem zugehörigen Wert (Value) beinhaltet. Die Objekte an sich bestehen wiederum aus einem komplexen *Sequence* Datentyp mit dem Namen *Varbind*. Hierin enthalten ist die *OID* des Objekts so wie deren Wert.
Wird nach einem Wert eines Objekts angefragt, so ist der Wert innerhalb des Feldes *Value* in der *Sequence Varbind* natürlich 0, bei einer Antwort, bzw. bei einer Setz Aufforderung, beinhaltet das Feld *Value* jedoch dann den gewünschten Wert, den das Objekt bekommen soll. Im Beispiel in Abbildung 2.11 handelt es sich um eine Anfrage nach einem Wert, um eine sogenannte Get-Anfrage[18], deshalb ist das zweite Byte innerhalb des Feldes *Value* = 0.
→ Innerhalb einer *Varbind List*, können mehrere *Varbind* Felder platziert werden.

[18] hierzu mehr in Abschnitt 2.3.7

Tabelle 2.4.: SNMP Error Codes [Presuhn, 2002a]

Fehlercode	*Fehlername*	**Fehlercode**	*Fehlername*
0x00	noError	0x10	wrongValue
0x01	tooBig	0x11	noCreation
0x02	noSuchName	0x12	inconsistentValue
0x03	badValue3	0x13	resourceUnavailable
0x04	readOnly	0x14	commitFailed
0x05	genErr	0x15	undoFailed
0x06	noAccess	0x16	authorizationError
0x07	wrongType	0x17	notWritable
0x08	wrongLength	0x18	inconsistentName
0x09	wrongEncoding		

Bei SNMPv3 wird dieser Ablauf jedoch etwas komplexer. Das Prinzip ist natürlich das Gleiche geblieben, jedoch sind einige Felder hinzugekommen. In Abbildung 2.12 ist das Nachrichtenformat einer SNMPv3 Nachricht abgebildet. (Der Aufbau wurde von RFC3412 [Case et al., 2002a] übernommen). Die Nachricht an sich besteht wie bei SNMPv1 bzw. 2 aus einer *Sequence*. Das erste Feld innerhalb der *Sequence* ist die *msgVersion*. Da dieses Feld identisch in allen drei SNMP Versionen ist, lässt sich hierdurch problemlos zwischen den Versionen unterscheiden. Das nächste Feld ist wieder eine *Sequence*, es beinhaltet die Header Daten *msgID*, *msgMaxSize*, *msgFlags* und *msgSecurityModel*. Das Feld *msgID* wird zwischen SNMP Manager und Agent genutzt, um Anfragen und Antworten zuordnen zu können. Rechts daneben sitzt das Feld *msgMaxSize*, es übermittelt dem Empfänger die maximale Nachrichtengröße des Senders. Die nun kommenden Flags sitzen in den drei letzten niederwertigsten Bits des Octetsstrings, zunächst gibt es das Flag *reportableFlag*, durch Setzen dieses Flags wird der Empfänger aufgefordert einen *Report* PDU zurückzusenden (das *reportableFlag* ist grundsätzlich bei allen Anfragen gesetzt). Die beiden noch verbleibenden Flags lauten *privFlag* und *authFlag*, sie geben Auskunft welche Sicherheitsmechanismen der Sender bei diesem Paket angewendet hat. Das letzte Feld in der Header Sequence *msgSecurityModel* enthält Informationen über das verwendete Sicherheitsmodell[19] [Stallings, 1999].

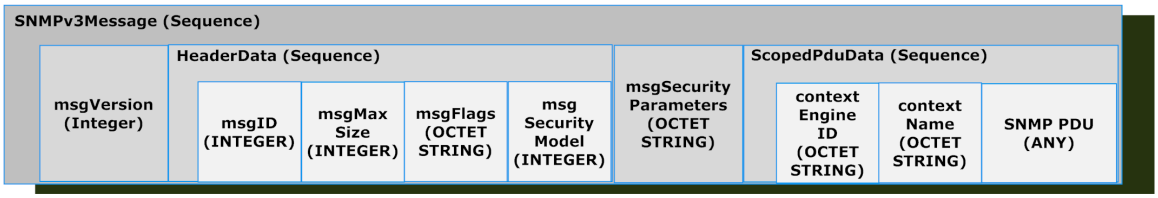


Abbildung 2.12.: SNMPv3 Nachricht

Wird nun der Header verlassen, so kommt als nächstes das Feld *msgSecurityParameters*. In diesem Bereich werden je nach Sicherheitsmodell Informationen gespeichert, die zwischen dem Sender Sicherheitsmodul und dem Empfänger Sicherheitsmodul ausgetauscht werden

[19] SNMPv1=1, SNMPv2c=2, USM=3, dazu mehr in Abschnitt 2.3.8

müssen. Die Inhalte dieses Feldes sind abhängig vom verwendeten Sicherheitssystem[20].
Es folgt zu nun, je nachdem ob das *privFlag* gesetzt wurde, entweder verschlüsselt oder nicht, das *Sequence* Feld *ScopedPduData*. Hierin enthalten sind die Felder *contextEngineID*, welches die eindeutige SNMP Engine ID der zugehörigen Anwendung enthält, das Feld *contextName*, worin der jeweilige Umgebungskontext innerhalb der zuvor definierten EngineID gespeichert ist, sowie die eigentlichen Daten im Feld *SNMP PDU* [Case et al., 2002a].

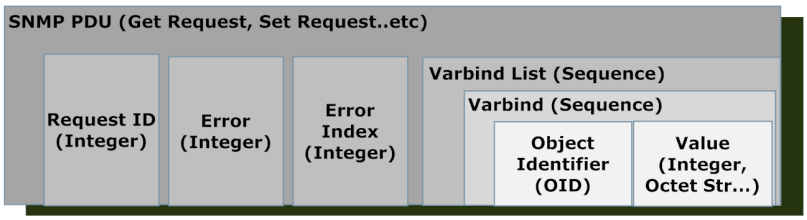


Abbildung 2.13.: SNMP PDU

Innerhalb des Feldes *SNMP PDU* befinden sich nun die PDU Sequence, die auch bei SNMPv1 bzw. SNMPv2 verwendet wurde (siehe Abbildung 2.13). Hier sind die zuvor besprochenen Felder *Request Identifier*, *Error Status*, *Error Index* sowie der *Variable Binding List*, welche die *Variable Bindings* beinhaltet, platziert.

2.3.6. Protokoll Architektur

Die SNMPv3 Architektur ist in RFC3411 [Harrington et al., 2002] vorgestellt. Sie besteht aus mehreren SNMP Einheiten, sogenannten Entities. Wobei diese Einheiten die Funktion des Agenten, des Managers oder beides zugleich übernehmen können. Jede SNMP Einheit besteht aus mehreren, voneinander abhängigen Modulen, welche sich gegenseitig bestimmte Dienste anbieten. Wobei alle Module so konstruiert sind, dass sie problemlos durch neuere oder bessere Modulversionen ausgetauscht werden können. Somit ist gewährleistet, dass beispielsweise das Sicherheitsmodul problemlos durch ein anderes, neues, eventuell besseres ausgetauscht werden kann, ohne das die anderen Module abgeändert werden müssen.

→ Da diese Arbeit sich mit der Entwicklung eines SNMP Agentensystems beschäftigt, wird der weitere Focus auf diese SNMP Einheit gelegt. Prinzipiell unterscheidet sich der Aufbau jedoch nur unwesentlich. Eine SNMP Manager Entity besitzt alle Module einer Agenten Entity jedoch kein Access Control Subsystem. Weitere Details zu SNMP Managern sind unter [Stallings, 1999, Harrington et al., 2002] zu finden.

Grundsätzlich besteht eine SNMP Einheit aus zwei Teilen, der SNMP Engine und den SNMP Applications. Die SNMP Engine beinhaltet Funktionsblöcke zum Senden und Empfangen von Nachrichten, zum Authentifizieren und Verschlüsseln bzw. Entschlüsseln von Nachrichten sowie zum Verwalten der Managed Objects. In Abbildung 2.14 ist der Aufbau einer Agenten Einheit sowie die Unterteilung in SNMP Engine und SNMP Applications zu sehen.
Die SNMP Engine enthält einen *Dispatcher*[21], ein *Message Processing Subsystem*, ein *Security Subsystem* und ein *Access Control Subsystem*. Die Schlüsselfigur innerhalb der SNMP

[20] auch hierzu mehr in Abschnitt 2.3.8
[21] to dispatch, engl. abfertigen

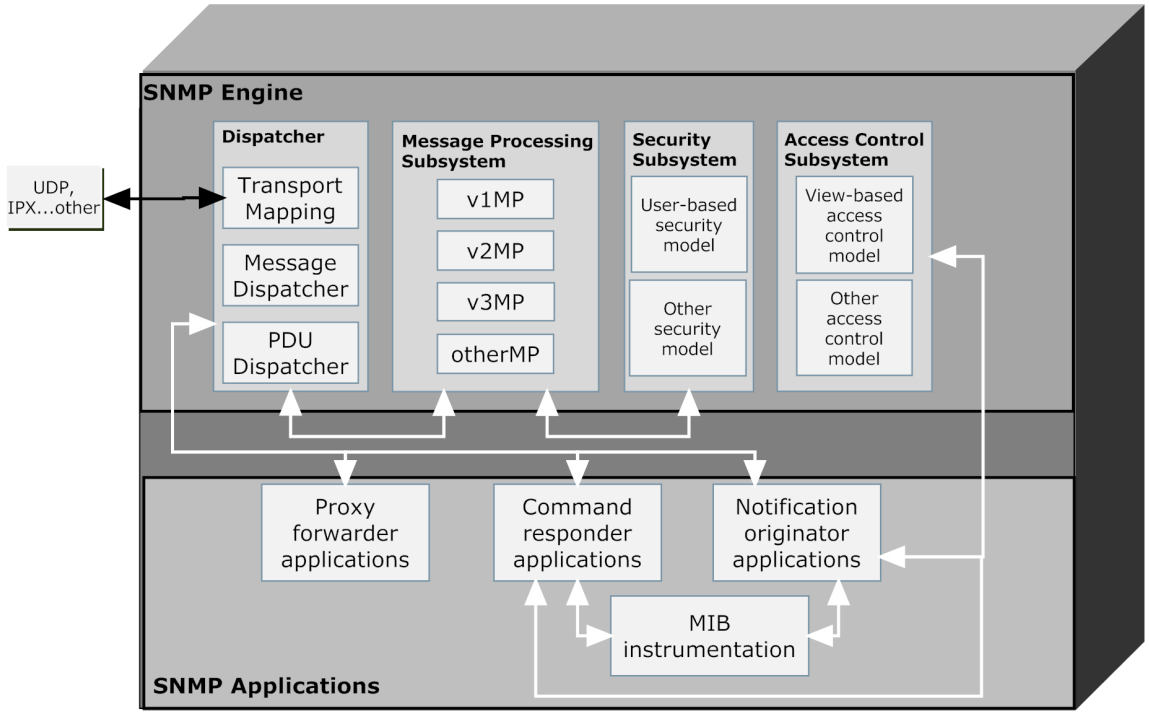


Abbildung 2.14.: SNMPv3 Agent Aufbau [Stallings, 1999]

Engine nimmt der Dispatcher ein. Er stellt eine Art Verkehrsmanager bzw. Router zwischen den SNMP Applications, dem Message Processing Subsystem und dem Netzwerk dar. Für ausgehende Nachrichten erhält der Dispatcher den PDU von der SNMP Applikation und leitet diese je nach benötigter SNMP Version an das versionsspezifische Message Processing Subsystem weiter. Das Message Processing Subsystem generiert die benötigten Header und leitet die Daten weiter an das Security Subsystem. Dieses verarbeitet die benötigten Sicherheitsoptionen, verschlüsselt die Nachricht oder generiert die Authentifizierungsdaten. Nach Abschluß der Bearbeitung übergibt das Security Subsystem die Nachricht zurück an das Message Processing Subsystem. Das Message Processing Subsystem leitet die Nachricht zurück an den Dispatcher. Nun kann die SNMP Nachricht versendet werden. Der Dispatcher übergibt dazu die Nachricht an die Transportschicht.

Für ankommende Nachrichten empfängt der Dispatcher die Nachricht über die Transportschicht und leitet die Nachricht zum versionsspezifischen Message Processing Subsystem weiter. Das Security Subsystem überprüft die Authentifizierung und entschlüsselt die Daten (wenn nötig). Die entschlüsselte Nachricht kommt zurück zum Message Processing Subsystem. Dort werden die Daten weiter extrahiert und zurück an den Dispatcher übergeben. Dieser leitet die Daten an den zuständigen Service weiter.

Zusätzlich zur SNMP Engine sind in einem SNMP Agent drei Services implementiert, der *Command Responder*, der *Notification Originator* und der *Proxy Forwarder*. Der wichtigste Service ist der *Command Responder*, er bietet Zugang zu den Managed Objects über das Access Control Subsystem. Der Command Responder bearbeitet die *Get* und *Set* Anfragen

(siehe Abschnitt 2.3.7) und setzt oder liest die gewünschten Objektdaten. Außerdem generiet er die Antwort PDU. Der *Notification Originator* ist zuständig für die unidirektionalen Nachrichten (siehe Abschnitt 2.3.1). Die letzte Applikation der *Proxy Forwarder* bietet die Möglichkeit Nachrichten an andere SNMP Einheiten (Entitys) weiterzuleiten.

2.3.7. Protokoll Operationen

Wie in Kapitel 2.3.5 besprochen, enthält eine SNMP Nachricht einen PDU, welcher den Aufbau in Abbildung 2.13 hat. Je nachdem welchen Identifier die PDU besitzt, wird von der SNMP Engine die jeweilige Aktion durchgeführt. In Abbildung 2.15 sind alle möglichen SNMP Operationen laut RFC3416 [Presuhn, 2002a] mitsamt Identifier und der jeweils zuständigen Service Applikation aufgeführt.

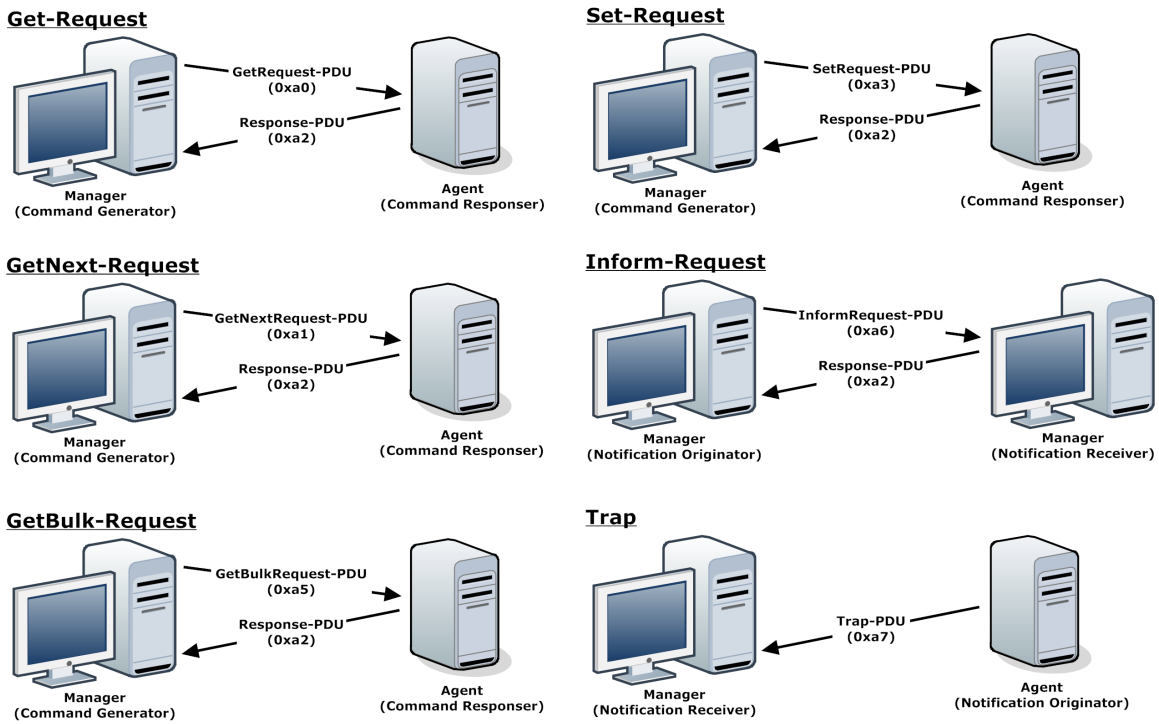


Abbildung 2.15.: SNMP Protokoll Operationen

Die ***Get-Request*** Nachricht ermöglicht es dem Manager auf ein oder mehrere Objekte des Agenten zuzugreifen und als *Response* Nachricht den Wert des Objekts oder der Objekte zu bekommen. Der Agent verarbeitet dann jedes Objekt innerhalb der *Variable Binding List*, also jedes *Variable Binding* und fügt in deren *Value* Feld den Wert des Objektes ein. Sollte während der Verarbeitung ein Fehler aufgetreten sein, so wird das *value* Feld mit dem Fehler beschrieben[22]. NoSuchObject wird eingefügt wenn die gesuchte OID nicht gefunden wurde, noSuchInstance wird eingefügt wenn kein Zugriff auf die Variable gewährt wurde. Bei allen anderen Fehlern wird das *value* Feld auf Null gesetzt und es werden die Felder *Error* und *Error Index* verwendet, wobei das Feld Error einen Fehlercode enthält und das Feld Error Index auf das Variable Binding innerhalb der Variable Binding List zeigt, welches für den

[22]mögliche Fehler: noSuchObject, noSuchInstance, endOfMIBView

Fehler verantwortlich ist, siehe Abbildung 2.13 [Presuhn, 2002a].
Der ***Get-Next-Request***-Befehl liefert immer den Wert des nächsten Objekts in der MIB zurück, dabei wird lexikographisch vorgegangen. Ist das Ende der MIB erreicht und existiert kein weiteres Objekt mehr, so wird das *value*-Feld mit dem Fehlercode *enofMIBView* beschrieben. Mithilfe mehrerer aufeinanderfolgender *Get-Next-Requests* können sogenannte MIB-Walks durchgeführt werden. Diese ermöglichen es, alle Objekte innerhalb einer MIB anzuzeigen ohne die MIB Struktur zu kennen.
Der Befehl ***GetBulk-Request*** wurde entwickelt, um das Datenaufkommen von GetNext Requests zu minimieren. Das Prinzip ist das Gleiche wie der GetNext Request, nur das dieses Mal so viele Variable Bindings wie möglich in ein SNMP Paket gepackt werden[23], statt für jedes Objekt ein separates Paket zu versenden. Die Anzahl der Objekte pro Paket ist begrenzt durch die maximale Nachrichtengröße des Empfängers[24]. Die Get-Bulk Nachricht besitzt als einziger PDU Typ zwei Felder, die in allen anderen PDUs nicht enthalten sind. Dies sind die Felder *non-repeaters* und *max-repetitions*. Mit einer Get-Bulk Nachricht können zwei unterschiedliche Aktionen ausgeführt werden. Zum Ersten können mehrere Start OIDs bestimmt werden, die vom Agenten wie eine Standard Get-Next Nachricht behandelt werden. Zum Zweiten können mehrere Start OIDs bestimmt werden, die vom Agenten wie mehrere Get-Next Nachrichten nacheinander behandelt werden. Hierbei gibt das Feld *non-repeaters* die Anzahl der OIDs innerhalb der *VariableBinding List* an, die wie eine Standard Get-Next Nachricht verarbeitet werden sollen[25]. Sind innerhalb der *VariableBinding List* noch weitere Variable Bindings, also Variable Bindings mit einem Index größer als *non-repeaters*, so werden für jedes dieser Objekte *max-repetitions* Get-Next Operationen durchgeführt[26]. Ist an irgendeiner Stelle das Ende der MIB erreicht und es gibt demnach keine weiteren Objekte, wird in das Value Feld des Variable Bindings der Fehlercode *endOfMIBView* eingetragen. Treten andere Fehler während der Verarbeitung auf, werden keine Objekte in der Antwort versendet, stattdessen wird das Feld *Error* auf *genErr*[27] gesetzt, und das Feld *Error-Index* mit dem Index des Problem Objekts beschrieben.
Die ***Set-Request*** Nachricht wird vom Manager genutzt, um den Wert einzelner oder mehrerer Objekte gleichzeitig zu verändern. Der Agent überprüft nach dem Empfang zuerst die Größe der Nachricht und berechnet ob eine Response Nachricht erstellt werden kann[28]. Wäre die Response Nachricht zu groß, wird der Fehlercode *tooBig*[29] eingefügt und das Variable Binding Feld leer gelassen. Es folgt der größte Aufwand des Set-Requests, alle Variable Bindings in der Variable Binding List müssen überprüft werden. Dazu werden 12 Prüfungen durchgeführt, wenn eine von ihnen fehlschlägt, wird in der Response Nachricht der dazugehörige Error Code in *Error*, sowie der Index auf das verursachende Objekt in *Error Index* eingefügt und die weitere Verarbeitung abgebrochen, siehe Tabelle 2.5. Waren alle Überprüfungen erfolgreich, so werden alle Werte innerhalb der Agent MIB mit denen in der Set Request Nachricht aktualisiert. Sollte bei diesem Vorgang ein weiterer Fehler auftreten, werden alle bisherigen Änderungen rückgängig gemacht und das *Error* Feld im Response PDU auf *com-*

[23] auch hier wird die lexikographische Reihenfolge wie bei der Get-Next Anfrage genutzt
[24] siehe Feld *msgMaxSize* in Abbildung 2.12
[25] Es wird für diese OIDs das lexikographisch nächste Objekt zurückgegeben
[26] Die Anzahl der Get-Next Requests pro OID (für jedes Variabel Binding mit Index größer als *non-repeaters*) entspricht *max-repetitions*
[27] 0x05, siehe Tabelle 2.4
[28] Die Response Nachricht enthält alle Variable Bindings der Orginal Nachricht, die maximale Größe ergibt sich aus dem Feld *msgMaxSize* des Set-Requests
[29] 0x01, siehe Tabelle 2.4

mitFailed gesetzt. Können nicht alle Änderungen rückgängig gemacht werden, so enthält die Response Nachricht das *Error* Feld mit dem Wert *undoFailed*.

Tabelle 2.5.: Überprüfungen vor Set-Request Ausführung [Stallings, 1999]

Überprüfung	*Errorcode bei Nichterfolg*
1. Zugriff auf Variable möglich?	noAccess
2. kann Variable erstellt oder geändert werden?	notWriteable
3. Datentyp in Request und MIB identisch?	wrongType
4. Länge in Request und MIB identisch?	wrongLength
5. Typ ASN.1 confrom?	wrongEncoding
6. Kann der gewünschte Wert der Variable zugewiesen werden?	wrongValue
7. Kann die Variable jemals erstellt werden?	noCreation
8. Kann die Variable momentan erstellt werden?	inconsistentName
9. Existiert die Variable und kann sie jemals geändert werden?	notWriteable
10. Kann die Variable momentan geändert werden?	inconsistentValue
11. Sind alle benötigten Resourcen verfügbar?	resourceUnavailable
12. Sind keine sonstigen Fehler vorhanden?	genErr

Trap Nachrichten werden von Agenten versendet, um dem Manager wichtige Ereignisse mitzuteilen, es wird das Gleiche PDU Format wie bei den anderen Protokoll Operationen verwendet[30]. In jeder Trap Nachricht müssen zwingend die beiden Variable Bindings *sysUpTime.0* und *snmpTrapOID.0* enthalten sein. Diese beiden Objekte sind Bestandteil der MIBII, welche in RFC 3418 [Presuhn, 2002c] definiert ist. Der Agent fügt nun die Variable Bindings in die Variable Binding List ein, welche die Objekte enthält, die dem Manager mitgeteilt werden sollen. Auf Trap Nachrichten wird keine Bestätigung gesendet, siehe Abbildung 2.15.

Eine andere Möglichkeit eine SNMP Einheit über Ereignisse zu informieren, ist der ***InformRequest***. Dieser wird jedoch von den SNMP Managern genutzt, um andere Managementstationen über wichtige Ereignisse zu informieren, siehe Abbildung 2.15. Die Pflicht Variable Bindings entsprechen denen, die auch bei der Trap Nachricht verwendet werden. Erhält ein Manager einen *InformRequest*, so überprüft er ob es möglich ist eine Response zu generieren. Ist dies nicht möglich, d.h. die Response Nachricht würde die maximale Paketlänge des Response Empfängers überschreiten, wird wieder die Fehlermeldung *tooBig* versendet. In allen anderen Fällen wird die Information in der InformRequest Nachricht an die zugehörige Applikation weitergeben und eine Response Nachricht ohne Fehlercode versendet.

2.3.8. User-Based Security Model

Dieser Abschnitt widmet sich der Protokollsicherheit von SNMPv3. Da die Vorgängerversionen SNMPv1 und SNMPv2c keine ausreichende Sicherheit bieten konnten[31], wurde für

[30] Ausnahme: GetBulk-Request

[31] Die Sicherheit bei SNMPv1 und SNMPv2c basiert auf einem sogenannten "Community String", dieser wird wie eine Art Passwort gehandhabt. Leider wird der Community String im Klartext übertragen und kann somit durch einfachste Paketsniffing Tools herausgefunden werden.

SNMPv3 das sogenannte User-Based Security Model[32] entwickelt.
Laut RFC 3414 [Blumenthal and Wijnen, 2002] besteht das User Based Security Model aus drei Modulen.

- *Authentication Modul*
  Dieses Modul nutzt als Authentifizierungsprotokoll HMAC-MD5-96[33] oder HMAC-SHA-96[34], möglich sind aber auch andere Authentifizierungsprotokolle, es muss jedoch garantiert sein, dass nur mit dem korrekten Schlüssel gültige Nachrichten erstellt werden können.

- *Timeliness Modul*
  Durch den Einsatz des Timeliness Moduls, ist die SNMP Entity gegen den Erhalt und die Verarbeitung duplizierter bzw. verzögerter Nachrichten geschützt. Das Modul nutzt hierzu das Feld *snmpEngineBoots*(enthält die Anzahl der Neuinitialisierungen der SNMP Entity) und das Feld *snmpEngineTime*(enthält die aktuelle Zeit seit der letzten Neuinitialisierung).

- *Privacy Modul*
  Momentan ist in RFC 3414 [Blumenthal and Wijnen, 2002] nur ein Verschlüsselungsalgorithmus definiert, dies ist der CBC-DES[35]. Es wird jedoch auch durch RFC 3414 empfohlen, wenn die Möglichkeit besteht einen sichereren Algorithmus zu verwenden, dieser auch verwendet werden soll. So ist es zum heutigen Stand sinnvoll den sichereren AES CFB 128 Algorithmus[36] zu verwenden. Das Privacy Module verschlüsselt den PDU und garantiert somit die Vertraulichkeit.

Das *Authentication Module* sowie das *Privacy Module* lassen sich nach Bedarf an- und ausschalten. Dies wird über die Flags, welche in Abschnitt 2.3.5 besprochen wurden, eingestellt. Das *privFlag* und *authFlag* aktiviert das jeweilige Modul. Somit ergeben sich 4 Kombinationen, von denen jedoch nur 3 zulässig sind, siehe Tabelle 2.6.

Tabelle 2.6.: USM Security Level

authFlag (Bit 7)	**privFlag (Bit 8)**	**USM Security Level**
0	0	noAuthNoPriv
0	1	authNoPriv
1	0	nicht erlaubt
1	1	authPriv

[32] 2009 wurde ein weiteres Sicherheitsmodell, das Transport Security Model für SNMP in [Harrington and Hardaker, 2009] vorgestellt, worin die Sicherheit auf die unterliegenden OSI Schichten übertragen wird. In dieser Arbeit soll jedoch der Schwerpunkt auf das User Based Security Model gelegt werden.

[33] Hased-Based Message Authentication Code [Krawczyk et al., 1997, of Standards and Technology, 2002a] in Kombination mit der Einweg Hashfunktion Message Digest 5 [Rivest, 1992], wobei die Ausgabe auf 96Bit abgeschnitten wird.

[34] Ebenfalls Hased-Based Message Authentication Code [Krawczyk et al., 1997, of Standards and Technology, 2002a] jedoch in Kombination mit dem etwas sicheren Secure Hash Algorithm 1 [of Standards and Technology, 2002b], auch hier wird die Ausgabe auf 96Bit abgeschnitten.

[35] Cipher Block Chaining Data Encryption Standard [of Cable Telecommunications Engineers, 2008]

[36] Advanced Encryption Standard [of Standards and Technology, 2001a] Standard Cipher FeedBack Mode 128bit keysize [Blumenthal et al., 2004]

In Abschnitt 2.3.5 wurde das SNMPv3 Nachrichtenformat besprochen, in Abbildung 2.5 ist das Nachrichtenformat einer SNMPv3 Nachricht abgebildet. Dort befindet sich auch das Feld *msgSecurityParameters*. Dieses Feld ist eigentlich eine *Sequence* und enthält weitere Felder, siehe Abbildung 2.16. Bevor jedoch die einzelnen Felder besprochen werden können, muss zuerst der Begriff *Authoritative Engine* erklärt werden. So wird immer dann, wenn ein Paket mit Datenanteil gesendet wird, also ein Get, GetNext, Set oder Inform PDU enthalten ist, welcher eine Response Nachricht erfordert, der Empfänger als *Authoritative Engine* bezeichnet. Des Weiteren wird bei Nachrichten, welche keine Response benötigen, also einen Trap, Response oder Report PDU enthalten, der Sender dieser Nachricht als *Authoritative Engine* bezeichnet.

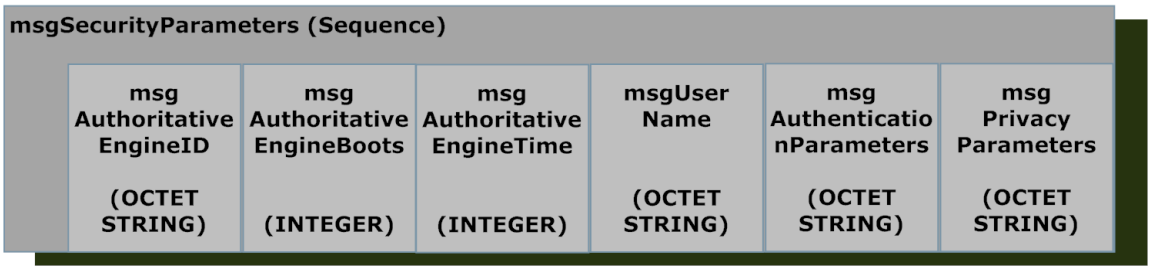


Abbildung 2.16.: USM Sicherheits Parameter innerhalb des SNMPv3 Pakets

Nun können die einzelnen Felder der *msgSecurityParameters* Sequence besprochen werden. Das Feld *msgAuthoritativeEngineID* enthält die Identifikationsnummer der SNMP Engine, welche in diesem Datenaustausch die Position der Authoritative Engine einnimmt. *msgAuthoritativeEngineBoots* beinhaltet, wie oft die Authoritative Engine seit dem letzten Softwareupdate neu gestartet wurde. Es sind Werte von 0 - $(2^{31} - 1)$ möglich. Hierzu gehört auch das Feld *msgAuthoritativeEngineTime*, es zeigt die Zeit an, die vergangen ist, seit die Authoritative Engine neu gestartet wurde, also die Zeit, seit msgAuthoritativeEngineBoots inkrementiert wurde. Diese beiden Felder ermöglichen es dem Timeliness Modul die Versandzeit zu bestimmen. Es folgt das Feld, welches vom Authentication Modul genutzt wird, das Feld *msgUserName*. Hier ist wie der Name schon sagt, der Name des Benutzers, welcher die Konversation initiiert hat, hinterlegt. Die beiden folgenden Felder beinhalten Parameter für die Sicherheitsmodule, das Feld *msgAuthenticationParameters*, enthält die benötigten Parameter für das Authentication Modul und das Feld *msgPrivacyParameters* enthält die Parameter für das Privacy Modul. Wird eines der beiden Module nicht genutzt, so hat das jeweilige Feld den Wert Null.

Timeliness Module

Das Timeliness Modul hat die Aufgabe, Schutz gegen Manipulation, Verzögerung, Wiederholung und Änderung der Reihenfolge von Paketen, zu garantieren. Es kontrolliert dazu die Zeit, die zwischen dem Versenden des Pakets und dem Erhalt des Pakets liegt.
Um die Zeit zwischen zwei unabhängig arbeitenden Geräten messen zu können, müssen sich der Agent und der Manager synchronisieren. Hier kommt wieder der Begriff der *Authoritative Engine* zum Einsatz. Es synchronisiert sich immer die *NonAuthoritative Engine* mit der *Authoritative Engine*. Dazu werden die vorhin erwähnten Felder *msgAuthoritativeEngineBoots* und *msgAuthoritativeEngineTime* benutzt. Um die SNMP Engines zu synchronisieren,

Listing 2.1: Aktualisierung der lokalen Variablen des Timeliness Moduls (Pseudo Code)

```
if ((msgAuthoritativeEngineBoots > snmpEngineBoots) or
((msgAuthoritativeEngineBoots = snmpEngineBoots) and
(msgAuthoritativeEngineTime > latestReceivedEngineTime)))
{
        snmpEngineBoots=msgAuthoritativeEngineBoots;
        snmpEngineTime=msgAuthoritativeEngineTime;
        latestReceivedEngineTime=msgAuthoritativeEngineTime;
}
//snmpEngineBoots, snmpEngineTime, latestReceivedEngineTime
// = lokaler Speicher, hier der der NAE
//msgAuthoritativeEngineBoots, msgAuthoritativeEngineTime
// = Paketdaten, hier der AE
```

wurde in [Blumenthal and Wijnen, 2002] ein **Discovery Prozess** festgelegt. Dieser besitzt folgenden Ablauf:

1. Die Nonauthoritative Engine (NAE) sendet eine Request Nachricht zur Authoritative Engine (AE), die Nachricht enthält im Feld *msgUserName* den Wert *initial* und im Feld *msgAuthoritativeEngineID* eine Null. Außerdem enthält die PDU eine leere *varBindList*.

2. Die AE antwortet darauf mit einer Report Nachricht, welche ihre *snmpEngineID* im Feld *msgAuthoritativeEngineID* enthält.

3. Es kommt nun der Schritt zur Zeitsynchronisierung. Dazu sendet die NAE einen weiteren Request mit gesetztem *authFlag* und der zuvor gelernten *msgAuthoritativeEngineID*. Die Felder *msgAuthoritativeEngineBoots* und *msgAuthoritativeEngineTime* sind auf Null gesetzt.

4. Als Antwort sendet die AE eine Report Nachricht mit ihren momentanen Werten für *msgAuthoritativeEngineBoots* und *msgAuthoritativeEngineTime*.

Die NAE speichert nun die Werte und inkrementiert *msgAuthoritativeEngineTime* jede Sekunde um Eins. Außerdem legt sie eine weitere Variable *latestReceivedEngineTime* an, in welcher der höchste Wert von *msgAuthoritativeEngineTime*, der jemals von der AE empfangen wurde gespeichert wird. Diese drei Werte werden von der NAE für jede AE, mit der sie Kontakt hat gespeichert. In jedem Paket, dass die AE versendet, sind ihre aktuellen Werte für *msgAuthoritativeEngineBoots* und *msgAuthoritativeEngineTime* enthalten. Die NAE aktualisiert ihre lokal gespeicherten Kopien dieser Werte immer dann, wenn die im Paket enthaltene *msgAuthoritativeEngineBoots* größer als der lokal gespeicherte Wert ist oder *msgAuthoritativeEngineBoots* der lokal gespeicherten Anzahl entspricht und *msgAuthoritativeEngineTime* größer als die lokal gespeicherte *latestReceivedEngineTime* ist. Tritt einer der beiden Fälle auf, so werden immer alle drei lokal gespeicherten Variablen aktualisiert. Das Listing 2.1 zeigt diesen Zusammenhang in Pseudo Code Darstellung.

→ Natürlich wird die Synchronisierung nur durchgeführt, wenn die Nachricht zuvor durch das Authentication Modul mittels HMAC validiert wurde.

Die eigentliche Aktualitätsüberprüfung findet beim ankommenden Paket nach unterschiedlichen Festlegungen statt. Ist die **AE der Nachrichtenempfänger**, so wird dieses immer dann als unglaubwürdig eingestuft, wenn eine der drei folgenden Bedingungen zutrifft:

- $snmpEngineBoots = 2^{31} - 1$
- $msgAuthoritativeEngineBoots \neq$ snmpEngineBoots
- ($msgAuthoritativeEngineTime$) größer oder kleiner ($snmpEngineTime \pm 150$s)

Ist die **NAE der Empfänger**, so wird die Nachricht immer dann als unglaubwürdig eingestuft, wenn eine der folgenden Bedingungen zutrifft:

- $snmpEngineBoots = 2^{31} - 1$
- $msgAuthoritativeEngineBoots < snmpEngineBoots$
- [$msgAuthoritativeEngineTime = snmpEngineTime$] &
  [$msgAuthoritativeEngineTime <$ ($snmpEngineTime$ -150s)]

→ Wobei auch hier wieder gilt:
$snmpEngineTime$, $snmpEngineBoots$ = lokale Variablen
$msgAuthoritativeEngineBoots$, $msgAuthoritativeEngineTime$ = Paketdaten

In beiden Fällen lässt sich also sagen, wenn die lokale Variable *snmpEngineBoots* ihren maximalen Wert erreicht hat, wird kein Paket mehr akzeptiert. Ist die AE der Empfänger, so muss die Anzahl der Bootvorgänge exakt mit den Bootvorgängen, die durch NAE übermittelt wurden, übereinstimmen. Außerdem darf die Zeit seit dem letzten Bootvorgang von der Zeit, die von der NAE übermittelt wurde, um maximal 150 Sekunden abweichen. Im Falle, dass die NAE der Nachrichtenempfänger ist, muss die übermittelte Anzahl der Bootvorgänge kleiner als die lokal gespeicherte Anzahl sein. Dies ermöglicht es der AE während zwei Paketen einen Reboot durchzuführen. Sollte die Anzahl der lokal gespeicherten Bootvorgänge gleich der übermittelten Anzahl sein, was den eigentlichen Normalfall darstellt, so wird zusätzlich überprüft, ob die Zeit seit dem letzten Bootvorgang größer als die lokal gespeicherte Zeit abzüglich 150 Sekunden ist. Die Nachricht darf also nicht älter als 150 Sekunden sein, jedoch wurde keine Festlegung getroffen, ob sie neuer sein darf. Diese Option wurde offen gelassen, um die Problematik zu lösen, dass der Timer der NAE schneller laufen könnte als der Timer der AE. Somit hat die NAE die Möglichkeit ihren Timer zu aktualisieren [Stallings, 1999, Blumenthal and Wijnen, 2002, Schwenkler, 2005].

User Verwaltung

Zur Verwaltung der Benutzer wird in RFC3414 [Blumenthal and Wijnen, 2002] eine MIB mit dem Namen usmUser Group beschrieben. Da es mehreren SNMP Einheiten möglich ist hier einen Benutzer anzulegen, bzw. Änderungen an den Benutzerdaten vorzunehmen, wurde hier eine Funktion eingerichtet, die den gleichzeitigen Datenzugriff kontrolliert. Es gibt hierzu den Datentyp *TestandIncr*, welcher in RFC2579 [McCloghrie et al., 1999b] beschrieben ist. Um

Variablen in der usmUser Group zu ändern, muss zuerst die *TestAndIncr* Variable *UserSpinLock* eingelesen (Get-Request) und danach mit dem eingelesenen Wert beschrieben werden (Set-Request). *UserSpinLock* erhöht sich durch diese Prozedur um Eins. Wurde *UserSpinLock* während dem Get- und Set-Request schon inkrementiert, d.h. entspricht der Wert im Set-Request nicht dem aktuellen Wert von *UserSpinLock*, so wird die Fehlermeldung *Inconsistent Value* zurückgegeben. Es wird nun in die Set Nachricht als zweite Variable Binding die zu setzende Variable gepackt. Somit ist das Setzen der Variable nur möglich, wenn nach dem Setzen von *UserSpinLock* keine Fehlermeldung zurück gesendet wurde. Durch diesen Mechanismus ist gewährleistet, dass zum Zeitpunkt der Änderungen kein Zugriff eines anderen Benutzers durchgeführt wird.

Die usmUser Group besteht aus dem skalaren Objekt *UserSpinLock* und aus der Tabelle *usmUserTable*. Innerhalb der Tabelle wird für jeden Benutzer eine Spalte angelegt, innerhalb derer Daten wie z.B. das benutzte Authorization Protocol, das Privacy Protocol, die geheimen Schlüssel sowie natürlich der Benutzername gespeichert sind. Eine komplette Darstellung der usmUserGroup ist im Anhang A.1) zu finden.

Kryptographische Funktionen

SNMPv3 enthält, wie anfangs erwähnt laut RFC3414 [Blumenthal and Wijnen, 2002] zwei kryptographische Funktionen, welche für die Authentifizierung und die Verschlüsselung verwendet werden. Dazu wird ein Privacy Key und ein Authentication Key benötigt. Für das Authentication Modul wird entweder HMAC-MD5-96 oder HMAC-SHA-96 verwendet.
Der HMAC-MD5-96 generiert aus dem *authKey* und der Nachricht eine 128 Bit Ausgangsfolge, welche im Anschluss auf 96 Bit gekürzt wird. Dieser Wert wird anschließend in das Feld *msgAuthenticationParameter* der SNMP Nachricht geschrieben, siehe Abbildung 2.12. Wird HMAC-SHA-96 verwendet, so muss ein 20 Byte *authKey* verwendet werden. Die Ausgangsfolge ist ebenfalls 20 Byte lang, wird aber im Anschluss ebenfalls auf 12 Byte gekürzt.
Für die Verschlüsselung, also das Privacy Modul wird CBC-DES empfohlen, es sind jedoch auch andere Verschlüsselungsfunktionen wie z.B. der AES CFB 128, welcher in RFC3826 [Blumenthal et al., 2004] beschrieben ist, zulässig.

→ Grundsätzlich sollte jedoch bedacht werden, dass DES schon seit längerer Zeit als nicht mehr sicher berurteilt wird. Schon 1998 wurde DES von der EEF[37] durch den Einsatz einer 250.000 Dollar Maschine [EFF, 1998] in weniger als drei Tagen geknackt. Der gleiche Rechner schaffte es im Jahr 1999 zusammen mit 100.000 weltweit vernetzten PCs von distributed.net[38] innerhalb von 22 Stunden [McNett, 1999]. Im Jahr 2006 kostete das Brechen des Algorithmus mithilfe der COPACOBANA Maschine [Kumar et al., 2006] nur noch knapp 9.000 Euro und dauerte etwa 6,5 Tage [Priplata and Stahlke, 2006]. Im Jahr 2009 schließlich schaffte es die Firma SciEngines GmbH durch einen ähnlichen FPGA Aufbau wie COPACOBANA den DES Algorithmus in weniger als einem Tag zu knacken [SciEngines-GmbH, 2012].

Je nach verwendetem Verschlüsselungsalgorithmus, werden unterschiedliche Kryptographische Modi durch die dazugehörigen RFCs empfohlen. Diese müssen eingesetzt werden, um

[37] Electronic Frontier Foundation `https://www.eff.org/`
[38] `http://www.distributed.net`

die Nachteile die durch monoalphabetische Chiffren auftreten, auszugleichen[39]. Durch die Verwendung des CBC Modus (Cipher Block Chaining) können diese Nachteile verhindert werden. Wird der CBC Modus genutzt, so wird jeder Klartextblock vor der Anwendung des Verschlüsselungsalgorithmus mit dem vorherigen schon verschlüsselten Block mittels XOR verknüpft. Da vor der Verschlüsselung des ersten Blocks kein vorheriger zur Verfügung steht, wird ein zufällig generierter Initialisierungsvektor genutzt. Der CFB Modus dient dem gleichen Zweck wie der CBC Modus, nämlich die Nachteile des ECB Modus zu beseitigen. Auch hier wird ein zufällig generierter Initialisierungsvektor genutzt, dieser wird jedoch zuerst verschlüsselt und anschließend mit dem Klartextblock über XOR verknüpft. Im Anschluß wird das Ergebnis der XOR Verknüpfung nochmals verschlüsselt. Dieses Ergebnis wird nun wieder über XOR mit dem nächsten Klartextblock verknüpft. Und dieses Ergebnis wird abermals verschlüsselt und immer so weiter. Auf diese Art entsteht durch die Anwendung von CFB ein Stromchiffre [Stallings, 2010].
Durch RFC3414 [Blumenthal and Wijnen, 2002] ist für DES der CBC Modus festgelegt, durch RFC3826 [Blumenthal et al., 2004] für AES der CFB Modus.
Bei DES-CBC werden nur die ersten 8 Byte des 16 Byte *privKey* Schlüssels verwendet. Da DES nur 56-Bit Schlüssel nutzt, wird das LSB jedes Bytes entfernt[40]. AES-CFB setzt den vollen 128 Bit *privKey* kein. Sowohl CBC als auch CFB benötigen einen Initialisierungsvektor (IV). Bei CBC werden, um diesen zu produzieren die letzten 8 Byte des *privKey* als sogenannter preIV genutzt. Bei CBC wird dieser preIV im Anschluß bitweise mit der sogenannten *saltValue*, welche bei CBC aus dem Wert der *snmpEngineBoots* sowie einer weiteren 4 Byte Integer Zahl, welche nach jeder Nutzung verändert werden sollte, generiert wird, verknüpft und bildet anschließend den Initialisierungsvektor. Bei CFB hingegen wird der IV komplett unabhängig vom *privKey* generiert. Es werden die ersten 4 Bytes der *snmpEngineBoots*(Most Significant Byte First) als die ersten 4 Byte des IV genommen. Die Bytes 4-8 des IVs bilden die ersten 4 Byte von *snmpEngineTime*. Die noch übrigen 16 Byte werden durch die ersten 16 Byte einer 64 Byte Integer Variablen gebildet. Diese Variable sollte einen zufälligen Wert haben und bei jeder neuen Nachricht um mindestens Eins inkrementiert werden. Bei CFB wird diese Variable *saltValue* genannt.

Der Absender fügt die *saltValue* vor dem Versenden in das *msgPrivacyParameters* Feld ein und ermöglicht es so dem Empfänger ebenfalls die korrekte IV zu berechnen [Stallings, 1999, Ertel, 2007, Spitz et al., 2011, Stallings, 2010, Blumenthal et al., 2004].

Auf eine detailliertere Erklärung der kryptographischen Funktionen wird an dieser Stelle verzichtet, da der Schwerpunkt dieser Arbeit nicht in der Kryptographie liegt, für detailliertere Erklärungen wird auf die Standards in Krawczyk et al. [1997], of Standards and Technology [2002a], Rivest [1992], of Standards and Technology [2002b], of Cable Telecommunications Engineers [2008], of Standards and Technology [2001a] sowie die Beschreibungen in [Stallings, 2010, Spitz et al., 2011] verwiesen.

[39] Zwar handelt es sich hierbei um ein monalphabetisches Chiffre mit Alphabet von 2^{64}, jedoch wäre es prinzipiell denkbar auch hierfür ein elektronisches Codebuch (ECB) zu erstellen, wodurch z.B. bei wiederholenden Vorgänge, wie z.B. einer Banküberweisung einzelne Blöcke ausgetauscht oder wiederholt werden könnten, und somit Daten manipulierbar wären. Wird kein besonderer Modus angewandt, so wird dieser Betriebsmodus auch ECB genannt [Ertel, 2007]

[40] $(8 \cdot 8) - 8 = 56$

2.3.9. View-Based Access Control Model

In RFC 3415 [Wijnen et al., 2002] ist das View-Based Access Control Model (VACM) beschrieben. Dieses regelt den Zugriff verschiedener Nutzer auf verschiedene Objekte der MIB. Da innerhalb des Contiki SNMP, welches in Abschnitt 3.5 beschrieben wird, kein VACM genutzt wird, wird auf eine detailierte Beschreibung in dieser Arbeit verzichtet. Es soll jedoch aus Gründen der Vollständigkeit erwähnt werden. Für weitere Informationen zum VACM wird auf [Wijnen, Presuhn, and McCloghrie, 2002, Stallings, 1999, Schwenkler, 2005] verwiesen.

2.4. Fazit

Es wurde zuerst der Standard *IEEE 802.15.4* behandelt, worin der Physical Layer und der MAC Layer definiert sind. Es wurden die möglichen Netzwerktopologien und der Rahmenaufbau besprochen, welcher bei deaktivierter Sicherheit 108 Byte an Daten tragen kann. Im Kapitel 2.2 wurde mit dem *6LoWPAN* Protokoll erläutert wie IPv6 über IEEE 802.15.4 durch Header für Fragmentierung, Komprimierung und Routing genutzt werden kann. Im darauffolgende Kapitel 2.3, wurde das *Simple Network Management Protokoll* detailliert beschrieben. Es wurden dort die Begriffe OID und MIB sowie die Datencodierung nach dem ASN.1 Standard mithilfe der BER erläutert. Das SNMP Nachrichtenformat wurde untersucht und die Protokollarchitektur sowie die Protokolloperationen der dritten SNMP Version wurden vorgestellt. Abschließend wurde das Sicherheitsmodell von SNMPv3 abgehandelt, welches ein Verschlüsselungsmodul (Privacy Module), ein Aktualitätsmodul (Timeliness Module) und ein Authentifizierungsmodul (Authentication Module) bietet.

3. Contiki OS

Contiki OS ist ein Open Source Betriebssystem für 8-Bit Mikrocontroller mit begrenzten Ressourcen. Die erste Contiki Version wurde im Jahr 2003 unter einer GNU Lizenz veröffentlicht. Contiki besitzt Möglichkeiten zur Multitasking Funktionalität, einmal über die sog. Protothreads bei denen alle Threads mit dem gleichen Stack[1] arbeiten, sowie über eine Multithreading Bibliothek, welche je nach Anwendung eingebunden werden kann und bei der jeder Thread seinen eigenen Stack nutzt. Der Betriebssystemkern ist ereignisgesteuert und bietet Interprozesskommunikation via Messagepassing durch Events sowie eine dynamische Prozessstruktur mit Unterstützung zum Laden und Entladen von Programmen zur Laufzeit an. Des Weiteren enthält Contiki einen IP Stack[2]. Der IP Stack nennt sich unter Contiki uIP (micro IP) und beinhaltet neben IPv4 und UDP Support auch IPv6 Unterstützung [Durvy et al., 2008a, Dunkels et al., 2004, Vasseur and Dunkels, 2010].

Contiki wurde bereits auf viele Plattformen wie z.B. den Atmel AVR oder den MSP430 von Texas Instruments portiert. Sogar Portierungen für den Commodore 64 sowie Atari 8 Bit oder den Game Boy wurden durchgeführt.
Die Anwendungsentwicklung unter Contiki wird unterstützt durch eine Entwicklungs- und Testumgebung, genannt Instant Contiki. Instant Contiki ist eine vorinstallierte Linux Debian Distribution als VMWare Image, welches das MSP430 gcc Compiler Toolset, AVR-gcc und die AVR toolchain sowie mehrere Simulatoren (Cooja, Netsim, MSPsim) und Entwicklungstools zur Anwendungsentwicklung enthält.

In diesem Kapitel werden zuerst die Contiki *Protothreads* erläutert, welche für die Anwendungsentwicklung unter Contiki essentiell sind. Es wird kurz die *Multithreading Library* vorgestellt mit der wirkliches Multitasking nach Bedarf genutzt werden kann. Es folgt Abschnitt 3.3. In ihm wird auf den Contiki *Kernel* und die *Prozessstruktur* eingegangen, welche ebenfalls eine fundamentale Wichtigkeit für das Grundverständnis des Contiki Betriebssystems darstellt. Der folgende Abschnitt 3.4 behandelt den *uIP Stack* und gibt praktische Hinweise im Umgang. Der letzte Abschnitt dieses Kapitels enthält die Beschreibung der Contiki Anwendung *Contiki SNMP*. Hier wird im Detail anhand des Quellcodes der Aufbau und die Funktionsweise des Programms beschrieben.

3.1. Protothreads

Bei Contiki OS handelt es sich um ein ereignisgesteuertes Betriebssystem. Man müsste also annehmen, dass deshalb auch als ereignisgesteuerte Programmiermethode, Statemachines (Zustandsautomaten) zum Einsatz kommen. Moderne Betriebssysteme basieren jedoch auf der Nutzung von Prozessen und Threads. Das bedeutet, sie nutzen Methoden um eine

[1] Stack engl. = Stapel, hier Stapelspeicher
[2] Stack engl. = Stapel, hier IP Protokollstapel

"Blocking and Wait"-Situation herzustellen, was wiederum bedeutet, die Ausführung eines Programmcodes zu stoppen und währenddessen einen zweiten Programmcode auszuführen. Diese Eigenschaft wird auch als simulierte Parallelverarbeitung bezeichnet. Eine reale Parallelverarbeitung ist durch den Einsatz mehrerer Prozessorkerne möglich, was jedoch bei 8 Bit Mikrocontrollern normalerweise nicht der Fall ist. Deshalb ist die einzige Möglichkeit um eine scheinbare Parallelverarbeitung zu erreichen, zwischen den einzelnen Programmcodes zu springen, also eine Art Zeitmultiplexing mit Prozessorressourcen durchzuführen. Hierbei wird jeder einzeln ablaufende Programmcode als Thread (Faden) bezeichnet. Grundsätzlich werden Threads in heutigen Betriebsystemen von Prozessen gestartet, welche wiederum ebenfalls eine Art von Paralellverarbeitung darstellen. So kann ein Prozess mehrere Threads besitzen. Damit Threads während ihres Ablaufs pausiert werden können und somit der Prozess einen anderen Thread ausführen kann, müssen sie wiedereintrittsfähig (reentrant) sein. Dies bedeutet jedoch, dass alle lokalen Variablen, Befehlszähler und Register irgendwo während der Wartezeit gespeichert werden müssen. Hierfür besitzt jeder Thread in modernen Betriebssystemen seinen eigenen Stack [Mandl, 2008]. Womit aber schon das Problem bei resourcenbeschränkten Geräten beginnt. Denn aufgrund ihrer Resourcenbeschränktheit, welche natürlich auch den vorhandenen Speicherplatz betrifft, ist es nicht möglich jedem Thread einen einzelnen Stack zuzuweisen. Und deshalb wird für gewöhnlich die Programmierung mit Threads auf 8 Bit Mikrocontroller nicht genutzt. Es wird die zuvor erwähnte ereignisgesteuerte Programmierung mit Zustandsautomaten eingesetzt. Da deren Programmierung meist komplizierter ist, wurde von den Contiki Entwicklern ein Verfahren entwickelt, das es ermöglicht Threads zu Nutzen und trotzdem nicht mehr Speicher als durch die Nutzung von Zustandsautomaten zu benötigen. Die sogenannten Protothreads. Protothreads bieten eine thread-ähnliche Programmierung, nutzen jedoch einen gemeinsamen Stack und sind auf einem ereignisgesteuerten Kernel lauffähig. Ermöglicht wird dieses Verhalten durch den simplen Einsatz von Präprozessor Makros, welche auf C `switch()` Routinen basieren. Dieses Verfahren wurde schon im Jahr 2000 von Simon Tatham auf dessen Webseite vorgestellt [Tatham, 2000] und ist im Grunde auf das sog. Duffs Device von Tom Duff [Duff, 1988] und die Assembler Coroutinen von Donald Knuth [Knuth, 1997](Section 1.4.2) zurückzuführen. Protothreads verursachen lediglich einen Overhead von 2 - 3 Byte pro Protothread[3][Dunkels et al., 2006b]. Dies macht sie gegenüber den herkömmlichen Multithreading Systemen mit separaten Stacks und daher großem Speicherbedarf, für ressourcenbeschränkte Geräte (Constrainted Devices), sehr vorteilhaft.

3.1.1. Funktionsweise

Sobald ein Protothread in eine Warteschleife gerät, wird diese mit der Funktion `PT_WAIT_UNTIL()` gekennzeichnet und an eine Bedingung geknüpft. Der Protothread wird verlassen und ein anderer Protothread wird solange ausgeführt, bis dieser wieder auf eine Wartebedingung gerät. Der Eventhandler überprüft nun während dem Wechsel zwischen den Threads, ob eine der zuvor gesetzten Bedingungen erfüllt wurde. Ist dies der Fall, so wird der dazugehörige Thread an der vorherigen Austrittsstelle weiter ausgeführt. Dies ermöglicht es zwischen den einzelnen Threads zu wechseln und diese scheinbar gleichzeitig auszuführen. Zur Veranschaulichung wird das Beispiel aus dem Bericht [Dunkels et al., 2006b] herangezogen. In Listing 3.1 ist anhand eines Pseudo Codes zur Ansteuerung eines Funkchips ein

[3] 2Byte->MSP430, 3Byte->AVR

Listing 3.1: Funkchip Ansteuerung mit ereignisgesteuerter Programmierung (Pseudo Code)

```
1  state: {ON, WAITING, OFF}
2
3  radio wake eventhandler:
4  if (state = ON)
5          if (expired(timer))
6                  timer  t_sleep
7                  if (not communication complete())
8                          state  WAITING
9                          wait timer  t_wait_max
10                 else
11                         radio off()
12                         state  OFF
13 elseif (state = WAITING)
14         if (communication complete() or
15                 expired(wait timer))
16                 state  OFF
17                 radio off()
18 elseif (state = OFF)
19         if (expired(timer))
20                 radio on()
21                 state  ON
22                 timer  t_awake
```

ereignisgesteuertes Programmierbeispiel dargestellt. Dieser Code wird nun in Listing 3.2 mit Protothreads umgesetzt.

Es ist deutlich zu sehen, dass sehr viele Codezeilen gegenüber der ereignisgesteuerten Lösung eingespart wurden. Außerdem erscheint der Protothread Ansatz übersichtlicher und leichter verständlich. Um jedoch die Umsetzung zu verstehen, ist es notwendig die Definitionen der Funktionen `PT_BEGIN()`, `PT_END()` sowie `PT_WAIT_UNTIL()` zu verstehen. In den Listings 3.3 und 3.4 sind die Präprozessor Anweisungen zu betrachten.

Mithilfe eines weiteren Beispiels aus [Dunkels et al., 2006b] soll nun gezeigt werden, wie diese Anweisungen genau umgesetzt werden. Dazu muss Listing 3.5 betrachtet werden. Es handelt sich hierbei um einen sehr vereinfachten Sender in C-Quellcode. Die Funktion enthält eine `dowhile` Schleife in der sich wiederum die `PT_WAIT_UNTIL()` Funktion befindet. Des Weiteren wurde der Funktionsbeginn und das Funktionsende durch die Macros `PT_BEGIN()` und `PT_END()` definiert.

Werden diese Macros nun durch ihre Definitionen in Listing 3.3 und 3.4 ersetzt, so erhält man den Quellcode in Listing 3.6.

Bei detaillierter Betrachtung der einzelnen Zeilen ist zu sehen, dass das Macro `PT_BEGIN(pt)` durch die Zeilen `switch(pt->lc) {` und `case 0:` ersetzt wurde (Zeile 2 und 3). Es wird also eine Switch Routine eröffnet und der Fall das `pt->lc == 0` ist wird festgelegt.

Das Makro `PT_WAIT_UNTIL(pt, cond1)` durch `LC_SET(c)` (Listing 3.4) sowie die Zeilen `if(!cond1) return PT_WAITING;` (Zeilen 8 und 9).

Listing 3.2: Funkchip Ansteuerung mit Protothreads (Pseudo Code)

```
radio wake prototthread:
PT_BEGIN
while (true)
        radio on()
        timer  t_awake
        PT_WAIT_UNTIL(expired(timer))
        timer  t_sleep
        if (not communication complete())
                wait_timer  t_wait_max
                PT WAIT UNTIL(communication complete() or
                                expired(wait timer))
        radio off()
        PT_WAIT_UNTIL(expired(timer))
PT_END
```

Listing 3.3: Definition der Protothread Funktionen 1

```
struct pt { lc_t lc };
#define PT_WAITING 0
#define PT_EXITED 1
#define PT_ENDED 2
#define PT_INIT(pt) LC_INIT(pt->lc)
#define PT_BEGIN(pt) LC_RESUME(pt->lc)
#define PT_END(pt) LC_END(pt->lc); \
return PT_ENDED
#define PT_WAIT_UNTIL(pt, c) LC_SET(pt->lc); \
if(!(c)) \
return PT_WAITING
#define PT_EXIT(pt) return PT_EXITED
```

Listing 3.4: Definition der Protothread Funktionen 2

```
typedef unsigned short lc_t;
#define LC_INIT(c) c = 0
#define LC_RESUME(c) switch(c) { case 0:
#define LC_SET(c) c = __LINE__; case __LINE__:
#define LC_END(c) }
```

Listing 3.5: Sender als Protothread Funktion

```
1  int sender(pt) {
2          PT_BEGIN(pt);
3          do {
4                  PT_WAIT_UNTIL(pt, cond1);
5          } while(cond);
6          PT_END(pt);
7  }
```

Listing 3.6: Sender als Protothread Funktion mit eingesetzten Präprozessor Anweisungen

```
1  int sender(pt) {
2          switch(pt->lc) {
3          case 0:
4                  do {
5                          pt->lc = 8;
6          case 8:
7                  if(!cond1)
8                  return PT_WAITING;
9                  } while(cond);
10 }
11 return PT_ENDED;
12 }
```

Listing 3.7: Makro für PT_YIELD

```
1 #define PT_BEGIN(pt) { int yielded = 1; \
2                       LC_RESUME(pt->lc)
3 #define PT_YIELD(pt) yielded = 0; \
4                       PT_WAIT_UNTIL(pt, yielded)
5 #define PT_END(pt) LC_END(pt->lc); \
6                       return PT_ENDED; }
```

Listing 3.8: Makro für PT_SPAWN

```
1 #define PT_SPAWN(pt, child, thread) \
2                       PT_INIT(child); \
3                       PT_WAIT_UNTIL(pt, thread != PT_WAITING)
```

Der Befehl `__LINE__` wird bei der Kompilierung durch die aktuelle Zeilennummer ersetzt, in diesem Fall in Listing 3.5 durch 8. Somit entsteht ein Unikum welches während der gesamten Kompilierung kein zweites mal enstehen kann. Diese Codezeile `case 8:` kennzeichnet den Wiedereintrittspunkt der Funktion bei erneutem Aufruf. Durch den Befehl `if(!cond1)` wird die Funktion immer beendet solange die Bedingung noch nicht erreicht wurde.

Das Makro `PT_ENDED()` schließt am Ende die Klammer der `switch()` Routine.

3.1.2. Erweiterungen

Da beim Umschreiben von Automaten in Protothreads oftmals Blockierungsmechanismen ohne Bedingungen benötigt werden, wurde zusätzlich zu `PT_WAIT_UNTIL()` die Funktion `PT_YIELD()` eingeführt [Dunkels et al., 2006b]. In Listing 3.7 ist das Präprozessor Makro zu sehen. `PT_YIELD()` entspricht größtenteils dem Makro `PT_WAIT_UNTIL()`, der Unterschied jedoch ist, dass zusätzlich die Variable `yielded` eingeführt wurde. Diese wird zusätzlich in `PT_BEGIN()` deklariert und initialisiert.

Somit wird nun erkannt, ob der Protothread bereits angehalten hat oder ob er dies noch tun soll. In PT_BEGIN() wird die Variable `yielded` immer auf Eins gesetzt. Dies sichert die Initialisierung der Variable bei jedem Aufruf. `PT_YIELD()` setzt die Variable auf Null und platziert einen `PT_WAIT_UNTIL()` Aufruf darunter. Somit wird beim ersten Aufruf der Protothread pausiert, und beim zweiten Aufruf weiter ausgeführt.

Eine weitere Funktion der Protothreads ist das Multithreading, also das Starten eines Threads aus einem Thread. Dazu gibt es das Makro `PT_SPAWN()`, siehe Listing 3.8, es wird innerhalb eines Protothreads gestartet und bewirkt das Starten eines neuen Child-Protothreads. Bei jedem Starten des Protothreads wird auch der Child-Protothread gestartet. Der Haupt-Protothread wird solange blockiert bis der Child-Protothread mit `PT_EXIT()` oder `PT_END()` beendet wurde.

3.1.3. Einschränkungen

Ein Problem bei der Verwendung von Protothreads stellt die Flüchtigkeit von funktionslokalen Variablen dar. Um Werte aus Protothreads beim nächsten Wiedereintritt zur Verfügung

zu haben, müssen diese explizit vor dem Starten einer wait-Anweisung gespeichert werden.

Eine weitere Möglichkeit ist die Verwendung von Variablen, welche mit `static local` deklariert wurden. Diese Variablen werden nicht im Stack sondern im Datenspeicher abgelegt und sind somit beim nächsten Wiedereintritt verfügbar.

Weitere Einschränkungen enstehen durch die von Protothreads unsachgemäße `switch`-Routinen Nutzung. Werden `switch`-Routinen im Quellcode von Protothreads genutzt, kann dies unerwartete Auswirkungen auf den Programmablauf sowie Kompillierungsfehler verursachen. `Switch`-Routinen können daher mit Protothreads nicht genutzt werden[4] [Dunkels et al., 2006b, 2005].

3.2. Multithreading Library

Zusätzlich zu den Protothreads bietet Contiki eine Bibliothek an, welche echtes, POSIX ähnliches, Multithreading erlaubt. Durch das Einbinden der Bibliothek `sys/mt.h` in den Quellcode, steht dem Programmierer die sog. Multithreading API zur Verfügung. Diese erlaubt es Threads zu nutzen, welche jeweils ihren eigenen Stack verwenden, und nicht wie die Protothreads, auf einen gemeinsamen Stack zurückgreifen [SICS, 2012]. Die Multithreading Library benötigt zusätzlich prozessorabhängigen Quellcode um zwischen den Threads zu wechseln. Sie sind im Quellcode Verzeichnis unter `\cpu\"Prozessor"\mtarch.c`, zu finden. Threads die über die Multithreading Library ausgeführt werden, sind nicht in der Lage Contiki Events zu empfangen und werden deshalb nur für unabhängige, freistehende Programme empfohlen [Dunkels, 2011a].

→ Contiki SNMP nutzt in seiner derzeitigen Implementierung die Multithreading Bibliothek nicht, es wird deshalb in dieser Arbeit nicht weiter darauf eingegangen. Weitere Details sind unter [Dunkels, 2011a] und [SICS, 2012] zu finden.

3.3. Kernel und Prozesse

Alle Prozesse in Contiki werden mit Protothreads aus Abschnitt 3.1 programmiert, selbst der Contiki Kernel ist ein Protothread. Für die Prozessprogrammierung wurden die Protothread Makros aus Kapitel 3.1 etwas abgeändert. Inhaltlich haben sich die Makros jedoch nicht geändert, der Aufruf geschieht durch die Makrobezeichnung in Listing 3.9.

3.3.1. Codeausführung

Ein Code kann in Contiki entweder preemptiv oder kooperativ ausgeführt werden. Das bedeutet hier, dass ein preemptiv ausgeführte Code jederzeit den kooperativ ausgeführten Code unterbrechen kann. Während ein kooperativ ausgeführter Code immer komplett ausgeführt wird bis der nächste kooperative Code gestartet wird [Dunkels, 2011b].

Prozesse in Contiki laufen immer kooperativ ab, der preemptive Ablauf ist hier für Gerätetreiber und für Echtzeitanwendungen vorgesehen. In Abbildung 3.1 ist hierfür ein Beispiel angegeben.

[4] Die bisher einzige Möglichkeit `switch`-Routinen zu Nutzen ist der Austausch der `switch` Makros durch GCC Makros siehe dazu [Dunkels et al., 2006b]

Listing 3.9: Prozessspezifische Makro Bezeichnungen

```
1  PROCESS_BEGIN()
2  //     Declares the beginning of a process' protothread.
3  PROCESS_END()
4  //     Declares the end of a process' protothread.
5  PROCESS_EXIT()
6  //     Exit the process.
7  PROCESS_WAIT_EVENT()
8  //     Wait for any event.
9  PROCESS_WAIT_EVENT_UNTIL()
10 //     Wait for an event, but with a with condition.
11 PROCESS_YIELD()
12 //     Wait for any event, equivalent to PROCESS_WAIT_EVENT().
13 PROCESS_WAIT_UNTIL()
14 //     Wait for a given condition; may not yield the process.
15 PROCESS_PAUSE()
16 //     Temporarily yield the process.
```

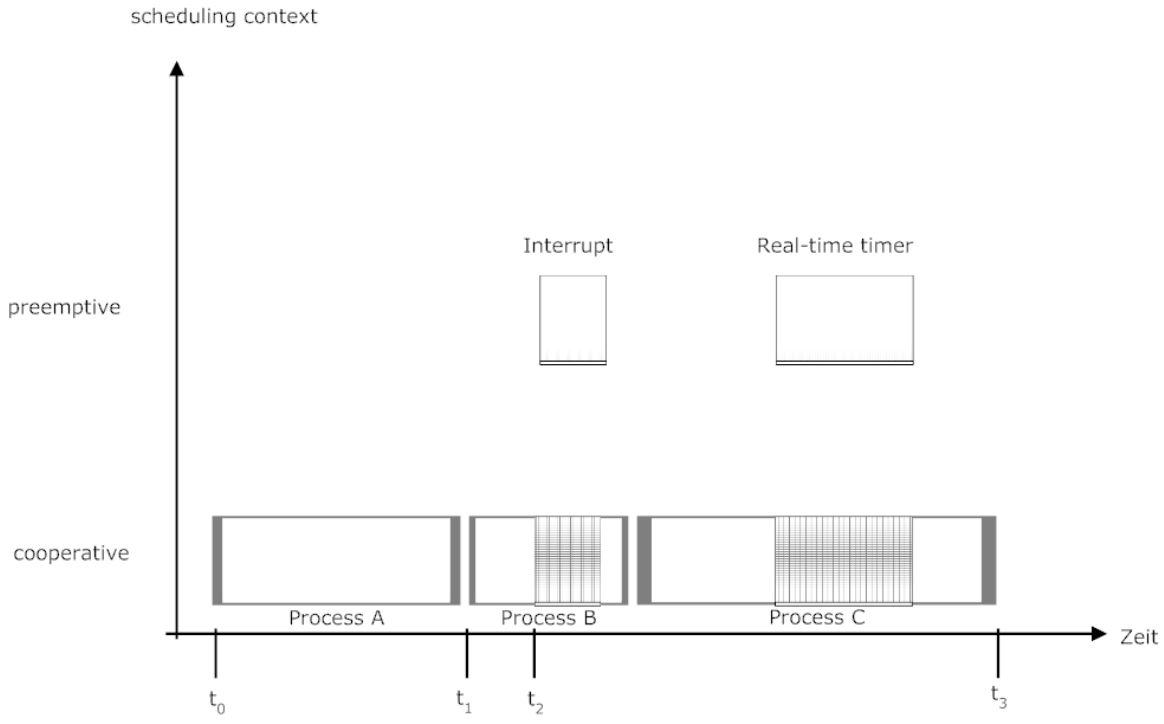


Abbildung 3.1.: präemptive und kooperative Prozesse [Dunkels, 2011b]

Listing 3.10: Prozesskontrollblock in Contiki

```
1 struct process {
2   struct process *next;
3   const char *name;
4   int (* thread)(struct pt *, process_event_t, process_data_t);
5   struct pt pt;
6   unsigned char state, needspoll;
7 };
```

Im unteren Bereich der Abbildung werden mehrere Prozesse kooperativ, also nacheinander ausgeführt. Prozess B wird nun zum Zeitpunkt t_2 durch einen Interrupt unterbrochen. Dieser Interrupt kann z.B. durch das Netzwerkgerät verursacht werden, wenn dort ein neues Datenpaket eingetroffen ist. Der Interrupt wird nun bearbeitet und Prozess B gestoppt. Erst nach der Beendigung der Interrupt Bearbeitung wird Prozess B weiter ausgeführt. Tritt während der Prozessausführung kein preemptiv ausgeführter Code auf, so wird wie zuvor für die Protothreads beschrieben, die simulierte Parallelverarbeitung der einzelnen Prozesse durchgeführt.

3.3.2. Prozesskontrollblock

Jeder Prozess in Contiki besteht aus zwei Teilen, dem Prozesskontrollblock und dem Prozess Thread [Dunkels, 2011b]. Der Prozesskontrollblock enthält die Laufzeitinformationen des Prozesses, also z.B. den Namen, momentanen Zustand usw., der Prozess Thread enthält den Quellcode des Prozesses. In Listing 3.10 ist die Struktur des Prozesskontrollblocks zu sehen.

Jeder Prozess kehrt bei Beendigung bzw. Unterbrechung zum Prozess Planer des Contiki Kernels zurück. Dieser verwaltet die Prozesskontrollblöcke der laufenden Prozesse. In Listing 3.10 ist die erste Variable `*next` ein Zeiger auf die nächste auszuführende Prozesskontrollstruktur. Und in der Variablen `pt` wird der aktuelle Zustand des Prozesses gespeichert. Über den Funktionszeiger `thread` ist der Zugriff auf den Prozess Thread, also den Quellcode möglich [SICS, 2012].

3.3.3. Interprozesskommunikation

Die Interprozesskommunikation unter Contiki wird über Events gesteuert. Das bedeutet, dass jeder Prozess die Möglichkeit besitzt einem anderen Prozess eine Nachricht bestehend aus einer Codenummer, sowie eventuell einer zusätzlichen Speicheradresse, zukommen zu lassen. Dabei wird zwischen asynchronen und synchronen Events unterschieden [Dunkels, 2011b].

Asynchrone Events Wird von einem Prozess ein asynchrones Event abgesetzt, so wird dieses nicht sofort übermittelt, sondern zuerst in die Warteschleife des Kernels eingereiht. Der Kernel arbeitet dann nacheinander die Events ab und übermittelt dem zugehörigen Empfangsthread die Eventinformationen. Asynchrone Events können als Empfänger entweder einen einzelnen Prozess oder alle Prozesse haben. Dem Programmierer steht hierzu die Funktion `process_post()` zur Verfügung [Dunkels, 2011b] [SICS, 2012].

Listing 3.11: reservierte Event Identifier in Contiki

```
1  #define PROCESS_EVENT_NONE            128
2  //This event identifier is not used.
3  #define PROCESS_EVENT_INIT            129
4  //sent to new processes when they are initiated.
5  #define PROCESS_EVENT_POLL            130
6  //sent to a process that is being polled.
7  #define PROCESS_EVENT_EXIT            131
8  //sent to a process that is being killed by
9  //the kernel
10 #define PROCESS_EVENT_CONTINUE        133
11 //sent by the kernel to a process that is
12 //waiting in a PROCESS_YIELD() statement.
13 #define PROCESS_EVENT_MSG             134
14 //sent to a process that has received a communication message.
15 //It is typically used by the IP stack
16 #define PROCESS_EVENT_EXITED          135
17 //sent to all processes when another process is about to exit.
18 #define PROCESS_EVENT_TIMER           136
19 //sent to a process when an event timer (etimer) has expired.
```

Synchrone Events Synchrone Events umgehen die Kernel Warteschleife und werden direkt an den Empfangsprozess übermittelt. Damit entspricht ein synchrones Event einem eigentlichen Funktionsaufruf. Synchrone Events können nur an einen bestimmten Prozess übermittelt werden und der Event-Sender-Prozess wird bis zur vollständigen Übermittlung angehalten [Dunkels, 2011b].

Synchrone Eventübermittlung wird mit der Funktion `process_post_synch()` erledigt [SICS, 2012].

Polling Das Polling ist eine spezielle Form der Eventübermittlung. Sie wird normalerweise von Gerätetreibern und anderen preemptiven Routinen genutzt [SICS, 2012]. Das Polling unter Contiki ist im eigentlichen Sinne ein Interrupt. Die Übermittlung erfolgt so schnell wie möglich und enthält keine weiteren Informationen [Dunkels, 2011b]. Der Aufruf geschieht mit der Funktion `process_poll()` [SICS, 2012].

Event Identifier Events werden durch sog. Event Identifier unterschieden, diese sind einfache Integer Zahlen. Zur Anwendungsentwicklung können alle Zahlen unterhalb 128 genutzt werden. Zahlen über 128 werden vom Contiki Betriebssystem genutzt, sind statisch alloziert und dienen unterschiedlichen Standard Aktionen, siehe Listing 3.11 [Dunkels, 2011b]. Befindet man sich in einem Prozess, so kann auf den Event Identifier, welcher den Prozess ausgelöst hat, über die Variable `ev` zugegriffen werden. Auf die Speicheradresse, welche zusätzlich übermittelt werden kann, ist der Zugriff über die Variable `data` möglich.

3.3.4. Prozessplaner

Die Steuerung sämtlicher Prozesse übernimmt das Kernel Prozess Modul[5], es ist das eigentliche Herz des Betriebssystems. Dort sind auch die zwei wichtigsten Daten Objekte der Prozesssteuerung zu finden, die Prozessliste und die Asynchrone Event FIFO Warteschleife. Die *Prozessliste* ist als einfache verlinkte Liste programmiert und enthält die Prozessstrukturen aller gestarteten Prozesse. Die Länge kann während der Laufzeit angepasst werden. Die *Asynchrone Event FIFO Warteschleife* enthält alle asynchronen Events, die noch zu erledigen sind. Implementiert ist die Warteschleife durch einen Ring Buffer mit fester Länge. Der Prozessplaner ruft, wenn ein Event bzw. ein Poll ihn erreicht den dazugehörigen Prozess auf bzw. arbeitet die Event Liste ab. Wird ein Prozess gestartet, so wird dieser solange ausgeführt bis er beendet, gestoppt oder pausiert wurde. Danach sucht der Prozessplaner den nächsten Prozess und führt diesen aus. Aus der Sicht des Contiki Prozessplaners haben Prozesse, auf die eine Polling Nachricht wartet, die höchste Priorität. Zur Überprüfung ob und wie viele Prozesse eine Polling Nachricht bekommen, kann über die globale Variable `poll_requested` herausgefunden werden. Das Ablaufdiagramm in Abbildung 3.2 zeigt den Ablauf des Prozessplaners wenn `poll_requested` ungleich Null war [Ashtawy et al., 2012].

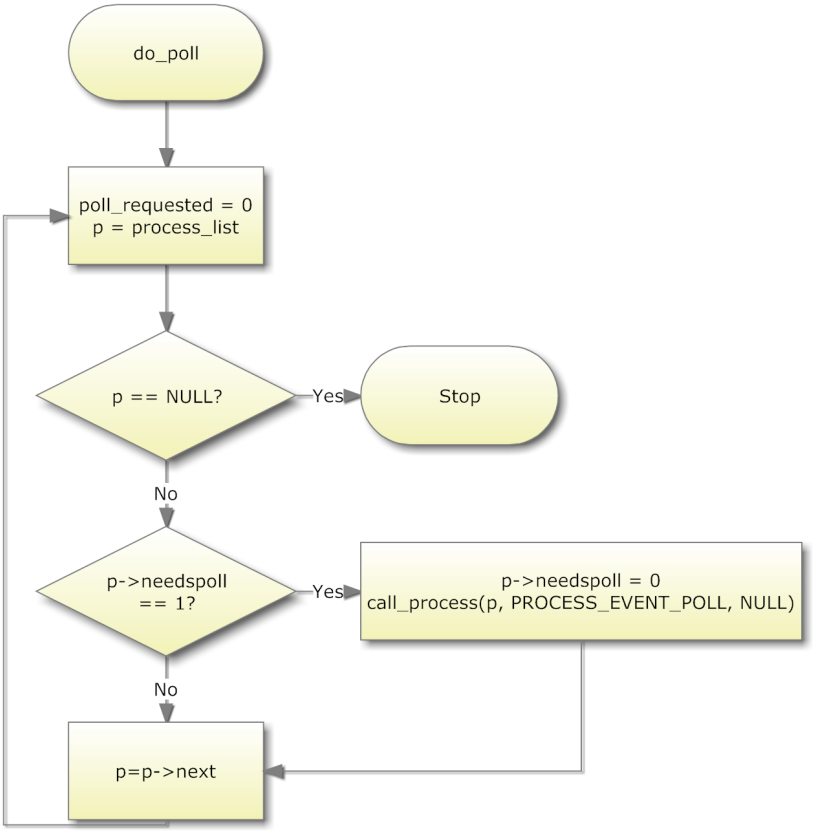


Abbildung 3.2.: Prozessplaner Polling Prozess [Ashtawy et al., 2012]

Wenn ein Poll abgesetzt wurde, wandert der Prozessplaner durch die System Prozess Liste und verarbeitet alle Prozesse bei denen das Polling Flag gesetzt ist. Wurden alle Polls

[5]Quellcode in der Datei `process.c`

ausgeführt, so wendet sich der Prozessplaner an die Event Warteschleife und führt dort das nächste anstehende Event aus. Ist dies erledigt überprüft er sofort wieder auf anstehende Polling Anfragen.

3.4. uIP

Die TCP/IP Protokollfamilie ist die am meisten genutzte Protokollimplementierung im Bereich des Internets sowie in lokalen Netzen. uIP (micro Internet Protokoll) ist die Implementierung für Embedded Devices mit eingeschränkten Ressourcen. Lange Zeit wurde die Meinung vertreten, dass es nicht möglich wäre die TCP/IP Protokollfamilie auf Embedded Geräten mit eingeschränkten Ressourcen, RFC konform zu implementieren. Erst im September 2001 gelang es dem Swedish Institute of Computer Science unter der Leitung von Adam Dunkels mit dem uIP Stack eine TCP/IP Implementierung bereitzustellen, welche die Anforderungen von Geräten mit beschränktem Speicher erfüllt und trotzdem RFC konform ist. uIP benötigt in der Standard Konfiguration nur 1 kByte RAM und wenige kByte ROM. Dabei sind alle Protokolle wie IP, ICMP, UDP und TCP enthalten. Dies wird ermöglicht durch ereignisgesteuerte Programmierung, ein einfaches Buffer Management und eine speichereffiziente TCP Implementierung [Vasseur and Dunkels, 2010]. Im Jahr 2008 entwickelten Julien Abeille und Mathilde Durvy (beide Cisco Systems) eine uIP Erweiterung, welche IPv6 Support auf Basis von 6LoWPAN ermöglicht [Durvy et al., 2008b].
uIP bietet dem Programmierer zwei Möglichkeiten zum Zugriff, einmal über die sogenannten Protosockets, welche eine ähnliche Handhabung wie BSD-Sockets auf Basis von Protothreads bieten, sowie über eine ereignisgesteuerte Raw-API. Da Protosockets nur für TCP Verbindungen verwendet werden können [SICS, 2012], SNMP jedoch auf UDP basiert, beschäftigt sich diese Arbeit mit der ereignisgesteuerten RAW-API. Genauere Informationen zu Protosockets sind der Contiki Dokumentation in [SICS, 2012] zu entnehmen.
uIP unterstützt nur die minimalsten Anforderungen der Protokoll Standards, so wurden alle Bedingungen, die in den RFCs mit MUST [Bradner, 1997] gekennzeichnet wurden und die für Host zu Host Kommunikation benötigt werden, erfüllt. Lediglich einige Funktionen, welche die Kommunikation zwischen Anwendung und Stack betreffen, wurden eingeschränkt [Dunkels, 2003]. Dadurch, dass uIP eine ereignisgesteuerte API nutzt, unterscheidet sich diese absolut von den TCP/IP BSD Sockets, welche nach dem POSIX Standard arbeiten [IEEE, 2008].

3.4.1. uIP Raw API

Bei der uIP Raw-API benachrichtigt der IP-Stack die zutreffende Anwendung sobald Daten bereitstehen. Will eine Anwendung Daten senden, so muss sie warten bis sie dazu von uIP aufgerufen wird. Da Speicherplatz eines der am wenigsten vorhandenen Dingen bei Geräten welche uIP verwenden ist, bietet uIP keine Möglichkeit den Paketbuffer während der Laufzeit dynamisch anzupassen. UIP verwendet einen statischen Buffer, der genau ein IP Paket speichern kann. Dieser Buffer wird gleichzeitig für das Senden und Empfangen genutzt, d.h. Anwendungen müssen ihre Daten zuerst abholen bevor sie etwas senden können. Wurde das momentan gespeicherte IP Paket nicht von der Anwendung abgeholt und somit der Buffer nicht von uIP freigegeben, können keine weiteren IP Pakete empfangen werden. Lediglich der Eingangsbuffer der Empfangshardware kann eventuell weitere Pakete speichern. Ist dieser

jedoch auch voll, so werden weitere ankommende Pakete nicht angenommen. Außerdem entsteht durch den kleinen Buffer das Problem, dass Anwendungen die Daten senden, im Falle eines Paketverlusts auf der Übertragungsstrecke selbst dafür sorgen müssen, die zuvor versendeten und nun eventuell verloren Daten nochmals bereitzustellen[6] [Vasseur and Dunkels, 2010].
Wenn ein Paket auf dem Gerät ankommt, so benachrichtigt der Netzwerkgerätetreiber den TCP/IP Stack. Der TCP/IP Stack verarbeitet danach das Paket und benachrichtigt, sofern Daten im Paket enthalten sind, die passende Anwendung. Da die Daten im Buffer beim nächsten eintreffenden Paket sofort wieder überschrieben werden, muss sich die Anwendung darum kümmern, die Daten zwischenzuspeichern. Natürlich wird der Paketbuffer erst mit einem neuen Paket überschrieben, wenn die Anwendung die Daten verarbeitet hat [SICS, 2012]. Sollte das IP Paket fragementiert worden sein, so besitzt uIP einen zweiten Buffer, welcher für das Wiederzusammensetzen verwendet wird. Es kann jedoch immer nur ein Paket zusammengesetzt werden, eine simultane Multidefragmentierung ist nicht möglich. Sollte also ein IP Fragment eines anderen Pakets eintreffen während zeitgleich ein IP Paket zusammengesetzt wird, so wird das IP Fragment des anderen Pakets nicht angenommen [Vasseur and Dunkels, 2010].
Diese Eigenschaften von uIP sorgen natürlich für einen geringeren Datendurchsatz im Vergleich zu bisher bekannten IP Stacks, besonders wenn ein uIP-Gerät mit einem PC kommuniziert [Dunkels, 2005].

3.4.2. uIPv6

Um ein wirkliches Internet of Things zu erhalten, wird natürlich IPv6 benötigt. Der große Adressraum (2^{128} anstatt 2^{32} Adressen) und der Autokonfigurationsvorgang machen IPv6 perfekt für das Internet der Dinge. Durch die Entwicklung und Standardisierung von 6LoWPAN, siehe Kapitel 2.2, wurden die IPv6 Header reduziert. Dies ermöglichte eine uIPv6 Implementierung mit nur 11,5 kByte Quellcodegröße und einem RAM Verbrauch von weniger als 2 kByte. Damit ist uIPv6 der kleinste IPv6 Stack der bisher verfügbar ist [Durvy et al., 2008b]. uIPv6 wurde von dem IPv6 Forum[7] im IPv6 Ready Programm[8] mit dem Phase 1 Logo ausgezeichnet und erfüllt die Bedingungen laut RFC4294 [Loughney, 2006].
Wird beim Kompilieren des Contiki Quellcodes das Flag `UIP_CONF_IPV6` gesetzt, so wird der IPv4 Stack durch den IPv6 Stack ersetzt. uIPv6 übernimmt alle weiteren Protokolle wie UDP, TCP usw. von uIPv4. Es ist außerdem sinnvoll das Flag `UIP_CONF_IPV6_CHECKS` zu setzen, dies gewährleistet, dass alle ankommenden IPv6 Pakete vor der Bearbeitung auf Korrektheit überprüft werden [SICS, 2012].
Nach dem Start von uIPv6 wird zuerst die Netzwerkschnittstelle initialisiert. Dazu kreiert der Knotenpunkt die link-lokale IPv6 Adresse aus der Kombination des `fe80 ::0/64` Präfix und der 802.15.4 MAC Adresse. Im nächsten Schritt wird der Duplicate Adress Detection (DAD) Vorgang ausgeführt um die Einzigartigkeit der generierten Adresse zu überprüfen. Gleichzeitig wird eine Router Solication Nachricht versendet um den Router zum Senden von Router Advertisments zu bewegen. Anschließend werden die Informationen aus dem erhaltenen Router Advertisment genutzt um die globale IPv6 Adresse zu generieren und die

[6] Bei Verwendung der Protosockets ist ein separater Buffer vorhanden, welcher bei TCP Verbindungen einen Retransmissions Buffer besitzt, dies führt jedoch zum doppelten Speicherverbrauch
[7] http://www.ipv6forum.com/
[8] http://www.ipv6ready.org

Netzwerkeinstellungen zu konfigurieren [Durvy et al., 2008b].
Ankommende Pakete werden von der Funktion `uip_process` verarbeitet. Nachdem die Header überprüft wurden, beginnt die Funktion die Extension Header zu verarbeiten. Es werden momentan vier Arten Extension Header unterstützt.(definiert in der Datei `uip.h`)

- Hop-by-Hop Options
- Routing
- Fragment
- Destination Options

→ uIPv6 kann zwar die Extension Header empfangen und verarbeiten, das Senden von Extension Headers wird jedoch nicht unterstützt.

Sollten die Pakete in fragmentierter Form eintreffen, so werden wie bei IPv4 die einzelnen Fragmente zusammengesetzt[9]. Die maximale Paketgröße des zusammengesetzten Paket beträgt 1280 Byte und die maximale Zeit die zwischen dem ersten und dem letzten Fragment eines Pakets liegen darf sind 60s [SICS, 2012].

3.4.3. 6LoWPAN Implementierung

Die 6LoWPAN Implementierung sitzt zwischen dem MAC Layer und dem uIPv6 Layer. Die Implementierung unter Contiki hält sich an die Vorgaben aus [Hui, 2008a,b, Montenegro et al., 2007].

→ Die Contiki 6LoWPAN Implementierung unterstützt momentan nur 802.15.4 64-Bit Adressen.

Wird vom MAC Prozess ein neues 6LoWPAN Paket empfangen, so kopiert dieser den Datenbereich in den Paket Buffer. Anschließend wird die Sicslowpan Input Funktion aufgerufen. Das gleiche passiert in umgekehrter Richtung, wenn der IPv6 Layer ein Paket versenden will. Dieser speichert das Paket in den uip Buffer und ruft die Sicslowpan Output Funktion auf. Ist ein IP Paket zu groß, um in einen 802.15.4 Frame zu passen, so wird es fragmentiert wie in [Montenegro et al., 2007] beschrieben. Im ersten Fragment befindet sich immer der komprimierte bzw. unkomprimierte IP/UDP Header. Bei ankommenden fragmentierten Paketen wird das original Paket wieder zusammengesetzt. Wie bei der IP Defragmentierung kann auch hier nur ein Paket verarbeitet werden. Kommen andere Pakete oder Fragemente anderer IP Pakete während einer laufenden Defragmentierung an, so werden diese verworfen. Die maximale Zeit vom ersten bis zum letzten Fragment eines Pakets beträgt 20 Sekunden. Die maximale Größe eines Pakets ist beschränkt durch den statischen Buffer. Dieser kann bis zu 1280 Byte speichern. Durch setzen des Kompilierungs-Flags `SICSLOWPAN_CONF_FRAG` wird die Fragmentierung eingeschaltet. Nach dem Zusammensetzen des Pakets werden die Daten in den `uip_buf` verschoben.

[9] Aktiviert wenn Compilierungsflag `UIP_CONF_REASSEMBLY` gesetzt ist.

Durch das Kompilierungsflag SICSLOWPAN_CONF_COMPRESSION wird das Kompressionsschema festgelegt. Unterstützt wird HC1, HC01[10] sowie IPv6, wobei IPv6 bedeutet, dass keine Kompression genutzt wird [SICS, 2012].

3.4.4. uIPv6 Paketverarbeitungsablauf

Dieser Abschnitt beschreibt einen kompletten Paketdurchlauf, der bei Nutzung von uIPv6 abläuft, wenn ein IPv6 Paket empfangen bzw. versendet wird. Es werden hierzu die wichtigsten genutzten Funktionen sowie ihre Position im Quellcode angegeben.

Initialisierung

Als unterste Schicht dient der sogenannte Sicslowmac, welcher in der Datei siclowmac.c zu finden ist. Der dort definierte Prozess mac_process() kommuniziert direkt mit dem Radio Treibern, welche sich z.B. bei dem in dieser Arbeit verwendeten Atmel AT86RF230 im Ordner \cpu\avr\radio\ befinden. Über den Aufruf der Funktion mac_init() (mac.c) wird dem verwendeten Interface die MAC Adresse zugewiesen und über die Funktion sicslowpan_init() (sicslowpan.c) die 6LoWPAN Implementierung initialisiert. Dort wird unter anderem die Funktion input() (sicslowpan.c) als die Funktion festgelegt, welche später bei eingetroffenen Daten durch die Funktion sicslowmac_dataIndication() (sicslowmac.c) aufgerufen wird, außerdem wird die Funktion output() (sicslowpan.c) als Standard Output Funktion zum Senden ausgehender Daten festgelegt. Im letzten Schritt wird schließlich noch der IEEE-15-4 Manager[11] über die Funktion ieee_15_4_init() (ieee-15--4-manager.c) initialisiert. Über die Funktion mac_task() (sicslowmac.c) wird dann die Hauptschleife gestartet, welche bei jedem eintreffendem MAC Event (MAC_EVENT_RX) überprüft ob der eintreffende Frame ein Datenframe ist.

Ankommendes IPv6 UDP Paket

Handelt es sich um einen Datenframe, so werden durch die Funktion sicslowmac_dataIndication() (sicslowmac.c) die Paketbuffer Funktionen gestartet. Die Paketbuffer Funktionen packetbuf_copyfrom() und packetbuf_set_datalen() (beide packetbuf-.c) haben zum einen die Aufgabe die ankommenden Daten in den Paketbuffer zu kopieren und außerdem den Datenbereichs vom Headerbereich zu trennen indem die Länge des Datenbereichs gespeichert wird[12]. Nachdem die Paket-Funktionen ihre Arbeit erledigt haben, ruft sicslowmac_dataIndication() die zuvor festgelegte input() Funktion aus sicslowpan.c auf. Diese setzt nun die einzelnen Fragmente zusammen, dekomprimiert Header und schreibt am Ende ein fertiges IPv6 Paket in den UIP_IP_BUF definiert in sicslowpan.c). Danach benachrichtigt sie den IP Stack indem sie die Funktion

[10] entspricht IPHC, vorgestellt in Kapitel 2.2, wobei nicht RFC 6282 [Hui and Thubert, 2011] als Definition verwendet wurde sondern draft-hui-6lowpan-hc-01 [Hui, 2008b]

[11] Der IEEE-15-4 Manager gehört zu den Radio Treibern und enthält Funktionen zur Einstellung des Physical Layers, alle diese Einstellungen werden in einer ieee_15_4_manager Struct Struktur gespeichert, diese Einstellungsbibliothek wird auch PID (Physical Layer Information Base) genannt.

[12] Hierbei muss beachtet werden, dass wenn die ankommenden Daten größer als der Paket Buffer sind, nur soviel abgespeichert wird, wie maximal in den Paket Buffer passt.(Die standardmäßige PACKETBUF_SIZE ist 128) (packetbuf.h)

tcpip_input() aus der Datei tcpip.c aufruft. Die Funktion tcpip_input() meldet über die Funktion process_post_synch (aus process.c) ein synchrones Event mit der Meldung PACKET_INPUT an den tcpip_process, welcher in der Datei tcpip.c definiert ist. Der Prozess tcpip_process, ruft nachdem er die Meldung empfangen hat, den eventhandler, ebenfalls in tcpip.c, auf. Dieser wiederum erkennt an der Meldung, dass es sich um ein eingehendes Paket handelt und startet die dazu vorgesehene Funktion packet_input()(tcpip.c), diese wiederum die Funktion uip_input()(uip.h), welche ihrerseits die Funktion uip_process()(uip.c) mit dem Parameter UIP_DATA aufruft. Die Funktion uip_process() überprüft den IPv6 Header sowie die Version und führt über eine goto Routine je nach überliegendem Protokoll weitere Schritte aus[13]. Handelt es sich bei dem aktuellen Paket um ein UDP Paket, so wird im Quellcode zu der Marke udp_input: gesprungen, bei der dann der zuständige Prozess informiert wird[14], dass Daten für ihn vorliegen.

Senden von eines IPv6 UDP Pakets

Soll ein UDP Paket gesendet werden, so kann hierfür die Funktion uip_udp_packet_send() (uip-udp-packet.c) genutzt werden. Diese kopiert die zu sendenden Daten in den uip_buf(eine typdef union in der Datei uip.h). Dann wird auch die Funktion uip_process() gestartet, diesmal jedoch mit dem Flag UIP_UDP_SEND_CONN, was innerhalb der uip_process() Funktion dazu führt, dass mit der goto Anweisung zur Marke udp_send: gesprungen wird. Dort werden die nötigen UDP Header und eventuell die Checksumme berechnet. Nach Beendigung befindet man sich wieder zurück in der Funktion uip_udp_packet_send(), welche darauf die Funktion tcpip_ipv6_output()(tcpip.c) startet. Diese erstellt alle IPv6 Header und startet danach die zuvor durch sicslowpan_init()(sicslowpan.c) festgelegte Standard Output Funktion output()(sicslowpan.c). Hier werden die nötigen 6LoWPAN Header Komprimierungen und Fragementierungen vorgenommen. Im Anschluß wird über die Funktion send_packet()(sicslowpan.c) schlußendlich die Funktion sicslowmac_dataRequest()(sicslowmac.c) gestartet, die das Paket dem Radio Treiber übergibt.

3.5. Contiki SNMP

Contiki SNMP wurde im Jahr 2010 an der Jacobs Universität von Siarhei Kuryla [Kuryla, 2010] entwickelt, der Quellcode wurde unter einer GNU Lizenz veröffentlicht. Er kann unter der Google Code Seite heruntergeladen werden[15]. Contiki SNMP hält sich weitgehend an die Struktur, die in RFC 3411 [Harrington et al., 2002] empfohlen wurde[16], somit ist es modular aufgebaut und über festgelegte Schnittstellen verknüpft. Momentan werden von Contiki SNMP die Nachrichtenformate SNMPv1 und SNMPv3 unterstützt. Bei Verwendung von SNMPv3 unterstützt der SNMP Agent das User Based Security Model mit HMAC-MD5 96 Authentication sowie AES-CFB 128 Verschlüsselung (siehe Kapitel 2.3.8).

[13] sollte es sich beim empfangenen IPv6 Paket um ein Fragment handeln, so wird die Funktion uip_reass()(auch uip.c) gestartet, welche die Defragmentierung vornimmt.
[14] Die Zuständigkeit wird über die Portnummer ermittelt.
[15] http://code.google.com/p/contiki-snmp
[16] siehe Kapitel 2.3, bzw. Abbildung 2.14

In diesem Kapitel sollen die einzelnen Module sowie die dazugehörigen Schnittstellen des SNMP Agenten detailliert erläutert werden. Es wird empfohlen während des Lesens dieses Kapitels den Contiki SNMP Quellcode zu studieren um Zusammenhänge einfacher begreifen zu können. Dieser ist, auf der dieser Arbeit beigefügten CD, im Unterverzeichnis *Doxygen_ Doku* zu finden.

3.5.1. UDP Handler

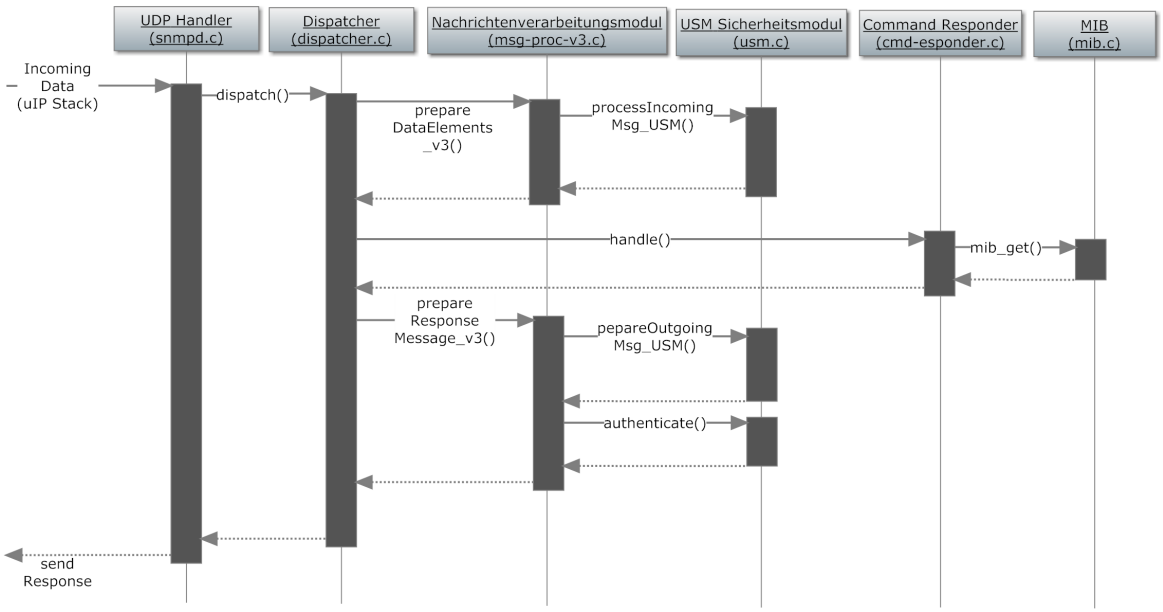


Abbildung 3.3.: Module und Schnittstellen des Contiki SNMP mit prinzipieller zeitlicher Ansteuerung und Lebenszeit

Der eigentliche SNMP Prozess befindet sich in der Datei `snmpd.c` (siehe Listing 3.12). Er wird dort mit der Zeile `PROCESS_THREAD(snmpd_process, ev, data)` gestartet. Die in Abschnitt 3.3 bzw. in Listing 3.9 dargestellten Makros für die Protothread Verwendung ist in Listing 3.12 in Zeile 2, 21 und 27 wiederzufinden. Der Prozess öffnet zuerst eine UDP Verbindung mit der Funktion `udp_new()` und bindet die geöffnete UDP Verbindung mit `udp_bind()` an Port 161[17]. Anschließend wird die MIB mit der Funktion `mib_init()` initialisiert (siehe dazu Abschnitt 3.5.6) und schließlich eine `while()`-Schleife geöffnet, welche mit `PROCESS_YIELD()` zuerst gestoppt wird und bei erneutem Aufruf die UDP Handler Funktion startet.

Der UDP Handler überprüft nun ob der ihm übermittelte Event Identifier für "eingetroffene TCP/IP Nachricht" steht (siehe dazu Listing 3.11) und ob neue Daten im UIP Buffer vorliegen (Funktion `uip_newdata()`, Rückgabewert≠0). Ist dies der Fall, so wird der Dispatcher gestartet.(siehe Abschnitt 3.5.2).

[17]Es wird zusätzlich noch die Funktion `HTONS()` verwendet, diese wandelt die Bitreihenfolge von Hostbyteorder zu Networkbyteorder

Listing 3.12: SNMP Daemon Prozess

```
1  PROCESS_THREAD(snmpd_process, ev, data) {
2          PROCESS_BEGIN();
3          #ifndef CONTIKI_TARGET_AVR_RAVEN
4          systemStartTime = clock_time();
5          #endif
6          #if CHECK_STACK_SIZE
7          u16t i = 0;
8          u32t pointer;
9          u32t* p = &pointer;
10         for (i = 0; i < 1000; i++) {
11             *p = 0xAAAAAAAA;
12             p--;
13         }
14         marker = &pointer;
15         #endif
16         udpconn = udp_new(NULL, HTONS(0), NULL);
17         udp_bind(udpconn, HTONS(LISTEN_PORT));
18         /* init MIB */
19         if (mib_init() != -1) {
20             while(1) {
21                 PROCESS_YIELD();
22                 udp_handler(ev, data);
23             }
24         } else {
25             snmp_log("error occurs while initializing the MIB\n");
26         }
27         PROCESS_END();
28 }
```

Der Dispatcher verarbeitet die Nachricht, benachrichtigt den Command Responder (Abschnitt 3.5.5), und generiert mithilfe des Nachrichtenverarbeitungsmodul (Abschnitt 3.5.3 eine Response Nachricht. Diese steht dem UDP Handler über den Zeiger `response` zur Verfügung.
Hat der Dispatcher seine Arbeit erledigt, so muss im letzten Schritt nur noch die Response Nachricht versendet werden, dazu wird mithilfe der Contiki Funktion `uip_udp_packet_send()` die Response Nachricht abgesendet.
Die UDP Handler Funktion wird nun beendet. Das Programm befindet sich nun wieder in der `while()` Schleife innerhalb des `snmpd` Prozesses. In Abbildung 3.3 ist der Programmablauf mit den einzelnen Modulen für eine SNMPv3 Nachricht mit *Get-Request* PDU dargestellt. Die einzelnen Module werden in den folgenden Abschnitten erläutert.

3.5.2. Dispatcher

⇒ Schnittstelle:

```
s8t dispatch (u8t *const input, const u16t input_len,
u8t * output, u16t *output_len, const u16t max_output_len)
```

Aufruf von: `udp_Handler()` (snmpd.c), UDP Handler Abschnitt 3.5.1

Der Dispatcher stellt die Hauptfigur in Contiki SNMP dar, er erhält von der UDP_Handler Funktion einen Zeiger `u8t *const input` auf das frisch angekommene SNMP Paket. Die Länge des Pakets wird durch die Variable `const u16t input_len` übergeben. Des weiteren erhält der Dispatcher vom UDP Handler einen Speicherplatz in Form eines Zeigers `u8t * output`, sowie die maximale Größe des Speicherplatzes `const u16t max_output_len`. Um nachher die Länge des Output Pakets angeben zu können, wird ihm der Zeiger `u16t *output_len` übergeben.

Der Dispatcher überprüft nun zuerst ob die übergebenen Daten in den vorhanden Buffer passen. War die Überprüfung erfolgreich, so kann mit dem Decodieren mithilfe der BER begonnen werden, siehe Abschnitt 2.3.4.

Decodieren der Nachricht unter Nutzung der BER

Dazu wird die Funktion `s8t ber_decode_sequence (const u8t *const input, const u16t len, u16t *pos)`, definiert in der Datei `ber.c` gestartet. Die Funktion `ber_decode_sequence()` überprüft zuerst den Typ des ersten BER codierten Objekts, siehe Tabelle 2.3, Abbildung 2.12 und 2.11, dieses muss 0x30 (Sequence) lauten. Zur Überprüfung startet `ber_decode_sequence()` die Funktion `ber_decode_type_length()`, welche wiederum die Funktion `ber_decode_type()` aufruft (alle in `ber.c`). Die Funktion `ber_decode_type()` nimmt die eigentlich BER Typ Zuordnung vor und speichert den aktuellen Typ in der Variablen `u8t type`. Es folgt die Längencodierung, wozu die Funktion `ber_decode_type_length()` die Funktion `ber_decode_length()` startet. Diese überprüft zuerst, ob die Längencodierung in der *definite-length* Codierung oder in der *indefinite-length* vorgenommen wurde und ob die Länge der folgenden Nutzdaten größer als 128 Byte ist. Siehe hierzu Abschnitt 2.3.4. Contiki SNMP unterstützt in der momentanen Implementierung jedoch nur Längenfelder, die nicht mehr als zwei Byte nutzen. Die Länge wird dann im Anschluß über den übermittelten Zeiger in die Variable `u16t length`

geschrieben. Die Funktion `ber_decode_length()` ist damit beendet und innerhalb der Funktion `ber_decode_sequence()` wird nun überprüft ob das Type Feld dem Datentyp *Sequence* entspricht, siehe Abbildung 3.4.

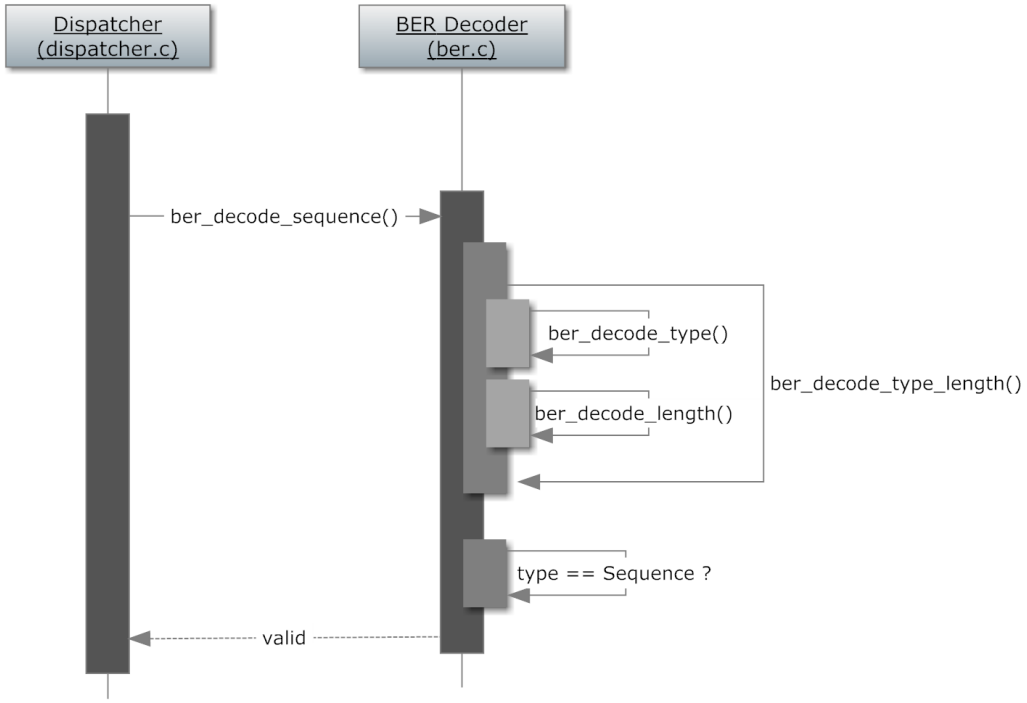


Abbildung 3.4.: Ablauf des Decodiervorgangs des Sequence Feldes

Im nächsten Schritt wird die SNMP Version decodiert, dazu wird die Funktion `s8t ber_decode_integer(const u8t *const input, const u16t len, u16t *pos, s32t *value)` verwendet, diese verwendet auch die Funktion `ber_decode_type_length()` um den Datentyp sowie die Länge zu decodieren, anschließend wird der decodierte Integerwert in die Variable `s32t tmp` geschrieben.
Über die Variable `tmp` wird nun die SNMP Version ermittelt, und eine `message_t` bzw. `message_v3_t` Struktur angelegt, die den Aufbau in Abbildung 3.5 bei SNMPv1 und den Aufbau in Abbildung 3.6 bei SNMPv3 hat. Zum Vergleich siehe Abbildungen 2.10 und 2.12. Bei SNMPv3 und auch im weiteren Verlauf wird die Struktur `typedef struct ptr_t` oft verwendet, diese besteht aus einem Zeiger, sowie einer dazugehörigen Längenangabe. Sie ist in `snmpd-types.h` in Zeile 64 definiert und ihr Aufbau ist in Abbildung 3.9 zu sehen.

1. Aufruf der versionsspezifischen Nachrichtenverarbeitungsfunktion

Es wird nun je nach ermittelter SNMP Version die zugehörige Nachrichtenverarbeitungsfunktion aufgerufen, also für SNMPv1 `prepareDataElements_v1()` und für SNMPv3 `prepareDataElements_v3()`. Der jeweiligen Funktion wird der aktuelle Zeiger auf die BER codierten Daten der SNMP Nachricht, die Gesamtlänge der Nachricht, ein Pointer auf die aktuelle Position (an dieser Stelle zeigt dieser Zeiger auf das erste Feld nach der SNMP Version), sowie einen Zeiger auf die `message_t` bzw. `message_v3_t` Struktur. Nach Ab-

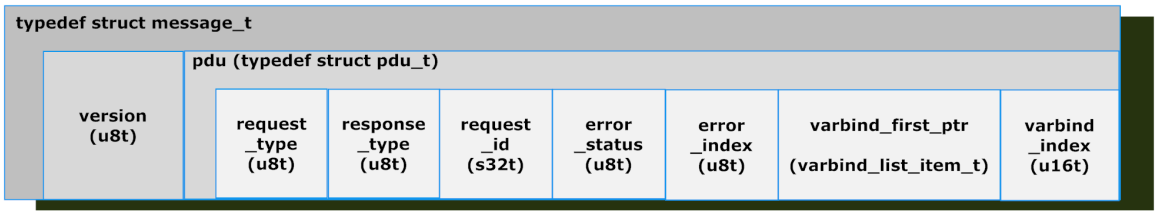


Abbildung 3.5.: Aufbau der message_t Struktur

schluß der Nachrichtenverarbeitungsfunktion steht dem Dispatcher nun eine decodierte und unverschlüsselte SNMP Nachricht in Form einer `message_t` bzw. `message_v3_t` Struktur zur Verfügung.

→Detailliertere Beschreibung der Nachrichtenverarbeitungsfunktion in Kapitel 3.5.3, Abschnitt *Decodieren ankommender Nachrichten mithilfe der BER*.

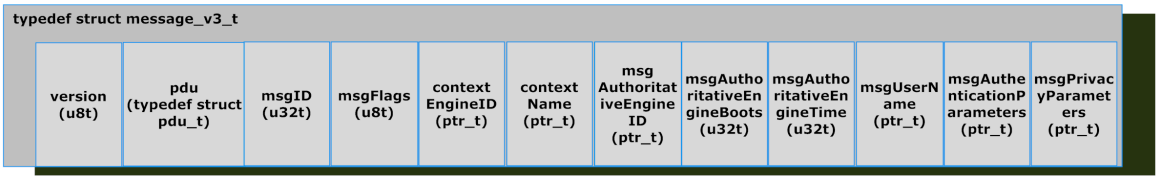


Abbildung 3.6.: Aufbau der message_v3_t Struktur

Aufruf des Command Responders

Die decodierte und unverschlüsselte Nachricht wird im nächsten Schritt an den Command Responder über die Funktion `handle()` weitergegeben, dieser verarbeitet die in der Nachricht enthaltenen Befehle (unterstützt werden *Get*, *Get-Next* und *Set Befehle*) und schreibt alle benötigten Daten für die *Response* Nachricht über den Zeiger `msg_ptr` in die `message_t` Struktur (Details hierzu siehe Abschnitt 3.5.5).

2. Aufruf der versionsspezifischen Nachrichtenverarbeitungsfunktion

Ist die `handle()` Funktion beendet, muss noch die *Response* Nachricht vorbereitet werden. Dazu wird wieder das Nachrichtenverarbeitungsmodul gestartet, diesmal jedoch über die SNMP versionsspezifische Funktion `prepareResponseMessage_vX()`, wobei X für die SNMP Version steht, 1 oder 3. Die Funktion nimmt nun die Codierung mithilfe der BER vor, ruft das Sicherheitsmodul bei Bedarf auf und liefert schließlich die fertige *Response* Nachricht über den Zeiger `output`.

→ Detaillierte Beschreibung der Nachrichtenverarbeitungsfunktion in Kapitel 3.5.3, Abschnitt *Codieren ausgehender Nachrichten mithilfe der BER*.

Der Dispatcher hat nun seine Arbeit erledigt und gibt den verwendeten Speicherplatz der `message_t` Struktur wieder frei. Das Programm kehrt nun zum `udp_handler` zurück.

3.5.3. Das Nachrichtenverarbeitungsmodul

Das Nachrichtenverarbeitungsmodul wird vom Dispatcher aufgerufen und hat die Aufgaben alle Elemente aus einer ankommenden SNMP Nachricht mithilfe der BER zu decodieren `s8t prepareDataElements_vX()` bzw. eine zu sendende SNMP Nachricht mithilfe der BER zu codieren `s8t prepareResponseMessage_vX()`. Wobei X hier für die jeweilige SNMP Version steht, also 1 oder 3.

Decodieren ankommendender Nachrichten mithilfe der BER

⇒ Nachrichtenverarbeitungsmodul Schnittstelle 1:

```
s8t prepareDataElements_vX(u8t *const input, const u16t len, u16t
*pos, message_v3_t *request)
```

Aufruf von: `dispatch()` (dispatcher.c), Dispatcher Modul Abschnitt 3.5.2

Je nach verwendeter SNMP Version wird vom Dispatcher die dazugehörige Funktion zum Decodieren gestartet, das Nachrichtenverarbeitungsmodul erhält die zuvor besprochenen Daten über die Schnittstelle. Das Vorgehen ist jedoch SNMP versionsspezifisch.

SNMPv1: Wurde zuvor vom Dispatcher SNMPv1 ermittelt, so wird von der Funktion `prepareDataElements_v1()` zuerst der Community String mithilfe der Funktion `ber_decode_string()`, welche sich wieder in der Datei `ber.c` befindet ermittelt. Die Funktion `ber_decode_string()` basiert auch wiederum auf der zuvor verwendeten `ber_decode_type_length()` aus dem vorherigen Abschnitt, siehe Abbildung 3.4. Das Nachrichtenverarbeitungsmodul überprüft nun mit einem einfachen `memcmp()` Befehl ob der Community String der Nachricht mit dem lokal gespeicherten Communitystring, abgelegt in `snmpd-conf.h` unter `#define COMMUNITY_STRING`, übereinstimmt. War die Überprüfung erfolgreich, so wird die Funktion **`ber_decode_pdu()`**, welche die Decodierung des PDUs (siehe Abbildung 3.5 pdu_t Struktur) vornimmt und auch in der Datei `ber.c` zu finden ist. Zuerst werden die Felder `request_type, request_id, error_status` und `error_index` auf die bereits zuvor besprochene Weise decodiert. Die Varbind Elemente werden jedoch speziell behandelt. Das Feld `varbind_index` wird zuerst auf den momentanen Stand der Variable `ptr` gesetzt, dieser beinhaltet dadurch die Startadresse der `varbind_list`. Danach werden die einzelnen *Varbind Objekte*, welche in der Nachricht enthalten sind verarbeitet, dazu wird die in der `message_t` Struktur enthaltene Struktur `varbind_list_item_t` verwendet, siehe Abbildung 3.7. Diese enthält eine weitere Struktur `varbind` vom Typ `typedef struct varbind_t`, sowie einen Zeiger auf das nächste `varbind_list_item_t`, siehe Abbildung 3.8. Die `typedef struct varbind_t` wiederum besteht aus drei Feldern, `oid_ptr` vom Typ `ptr_t*`, `value_type` (`u8t`) und dem eigentlich Wert `value` welcher aus einer `typedef union` besteht, welche ebenfalls in Abbildung 3.7 zu sehen ist. Die Funktion `ber_decode_pdu()` liest jetzt alle Variable Bindings aus der empfangenen Nachricht und speichert diese in die `varbind_list_item_t` Strukturen, jedesmal wenn ein weiteres Variable Binding in der Nachricht enthalten ist, wird mithilfe der Funktion `varbind_list_append()`, enthalten in `utils.c` durch `malloc()` Speicherplatz alloziert und damit ein Speicherplatz für das neue Variable Binding geschaffen. Der Zeiger auf das nächste Variable Binding wird immer in die vorherhige `varbind_t` Struktur im Feld `next_ptr` gespeichert.

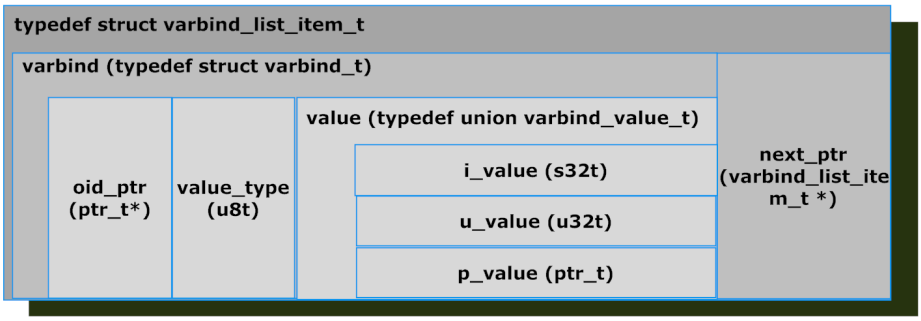


Abbildung 3.7.: Aufbau der varbind_list_item Struktur

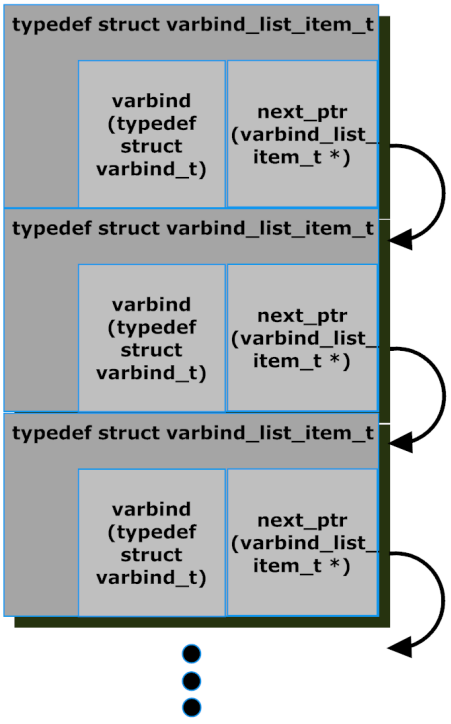


Abbildung 3.8.: Verbindungen zwischen den einzelnen Varbind Strukturen

Zum Decodieren der OID wird zuerst die Funktion `ber_decode_oid()` eingesetzt, welche überprüft ob der BER Typ OID ist und die Länge des Datenfelds zurückliefert. Im zweiten Schritt wird die Funktion `ber_decode_value()` eingesetzt, welche den BER codierten Wert in das Variable Binding speichert.

→ OIDs werden innerhalb des Agenten direkt BER codiert gespeichert, dies macht eine Decodierung der OID überflüssig.

SNMPv3: Ist der Nachrichtentyp SNMPv3, so wird vom Dispatcher nicht die Funktion `prepareDataElements_v1()` sondern die Funktion `prepareDataElements_v3()` gestartet. Es werden nun zuerst alle unverschlüsselten benötigten Felder der `message_v3_t` Struktur in Abbildung 3.6 aus der Nachricht decodiert(`msgID, msgMaxSize, msgFlags` und `msgSecurityModel`) und in die erzeugte Struktur gespeichert (nur `msgID` und `msgFlags`, die anderen werden nur überprüft). Anschließend wird die restliche Nachricht dem Sicherheitsmodul über die Funktion `s8t processIncomingMsg_USM(u8t *const input, const u16t input_len, u16t *pos, message_v3_t *request)` übergeben. Diese schreibt die unverschlüsselten Felder in die Struktur `message_v3_t *request` (detailiiertere Beschreibung in Abschnitt 3.5.4). Jetzt können die noch fehlenden Felder der Nachricht decodiert und in die `message_v3_t` Struktur geschrieben werden.[18] Der PDU wird mit der im vorherigen SNMPv1 Abschnitt besprochenen Funktion `ber_decode_pdu()` decodiert.

Das Programm kehrt nach Beendigung der Nachrichtenverarbeitungsfunktion wieder zum Dispatcher zurück.

Codieren ausgehender Nachrichten mithilfe der BER

⇒ Nachrichtenverarbeitungsmodul Schnittstelle 2:

```
s8t prepareResponseMessage_vX(message_t* message, u8t* output,
u16t* output_len, const u8t* const input, u16t input_len, const
u16t max_output_len)
```

Aufruf von: `dispatch()` (dispatcher.c), Dispatcher Modul Abschnitt 3.5.2

Auch beim Codieren der *Response* Nachrichten muss versionsspezifsch unterschiedlich vorgegangen werden. Es wird deshalb vom Dispatcher, wie im vorherigen Abschnitt, entweder die Funktion `prepareResponseMessage_v1` bei SNMPv1 oder die Funktion `prepareResponseMessage_v3` bei SNMPv3 gestartet, siehe Überschrift dieses Kapitel.

SNMPv1: Durch den Dispatcher wurde erkannt, dass es sich um eine ausgehende Nachricht mit der SNMP Version 1 handelt. Es wird deshalb die Funktion `prepareResponseMessage_v1` aufgerufen. Diese startet zuerst die Funktion `encode_v1_response()`, welche wiederum die Funktion `ber_encode_pdu()`, definiert in `ber.c`, ausführt.

Diese ist, wie der Name schon sagt, zuständig für die Codierung des PDUs mithilfe der BER. Nur wird bei der Codierung diesmal anders herum vorgegangen, im Gegensatz zur

[18] `contextName` und `contextEngineID`

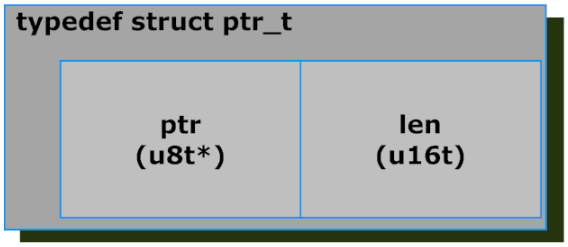


Abbildung 3.9.: ptr_t Struktur

Decodierung werden nun zuerst die Variable Bindings codiert. Dazu wird die Funktion `ber_encode_var_bind()`, ebenfalls `ber.c`, verwendet. Innerhalb dieser wird zu Beginn der Typ des Variable Bindings festgestellt, und danach die typspezifische BER Codierungsfunktion gestartet (Zugriff über `varbind->value_type` siehe Abbildung 3.7). Entweder `ber_encode_integer()` bei den Typen INTEGER, COUNTER, GAUGE oder TIME_TICKS, `ber_encode_fixed_string()` wenn der Typ OCTET_STRING ist, `ber_encode_oid()` wenn es sich bei dem Variable Binding um eine OID handeln sollte, oder die letzte Möglichkeit BER Type NULL. Nach der Typ Behandlung, wird zum Schluss noch die OID Struktur beschrieben. Diese besteht aus einer Struktur des Typs `ptr_t`, siehe Abbildung 3.9. Da die Länge der einzelnen Sub-OIDs größer als 128 sein können, müsste die Regel aus Abschnitt 2.3.4 beachtet werden, bei der mehrere Bytes für Sub-OIDs größer 128 genutzt werden müssen. Jedoch werden, wie zuvor erwähnt, die OIDs lokal schon BER codiert gespeichert. Dies macht eine Umwandlung überflüssig. Die OID kann also direkt in den Output Buffer kopiert werden (Funktion `ber_encode_oid()` (`ber.c`)), aufgerufen von `ber_encode_var_bind()`.
Innerhalb der Funktion `ber_encode_pdu()` wird die Funktion `ber_encode_var_bind()` für jedes Variable Binding, das in der Nachricht enthalten ist, aufgerufen. Kontrolliert wird dies durch den Zeiger innerhalb der Variable Binding Item Struktur `next_ptr`, siehe Abbildung 3.8. Wurden alle Variable Bindings bearbeitet, so werden die verbleibenden PDU Felder noch mit den typspezifischen Funktionen aus `ber.c` codiert und anschließend wird die Funktion `ber_encode_pdu()` beendet.
Zurück in der Funktion `encode_v1_response()`, fehlt noch die Codierung des Community Strings sowie des SNMP-Typ-Feldes. Dies geschieht ebenfalls über die typspezifischen BER Codierungsfunktionen aus `ber.c`. Zuletzt muss noch das Sequence Feld am Anfang der SNMP Nachricht codiert werden.

SNMPv3: Ist die zu versendende Nachricht vom Typ SNMPv3, so wird die Funktion `prepareResponseMessage_v3()` gestartet. Die Funktionsweise entspricht anfangs prinzipiell der von SNMPv1, so wird ebenfalls wie im vorherigen Kapitel unter Nutzung der Funktion `ber_encode_pdu()`, zuerst der PDU codiert. Nach Abschluß dieser Funktion, sind jedoch einige zusätzliche Schritte notwendig. So müssen zusätzlich die Felder *contextEngineID* und *contextName* mit der Funktion `ber_encode_fixed_string()` codiert werden. Anschließend muss das USM Sicherheitsmodul gestartet werden. Dies geschieht über die Funktion `prepareOutgoingMsg_USM()`, siehe Abschnitt 3.5.4. Die Funktion setzt die benötigten Felder für das Timeliness Modul, verschlüsselt bei aktivierter Privacy Option den gesamten PDU und codiert diesen wieder unter Nutzung der BER.

Nach Abschluß der USM Funktion, werden die noch fehlenden SNMPv3 Felder mithilfe der BER und der jeweils typspezifischen Funktionen aus `ber.c` codiert[19]. Zum Schluß muss die Nachricht noch authentifiziert werden[20]. Dies wird durch die Funktion `authenticate()` (`usm.c`) erledigt. Sie berechnet mittels der *HMAC* Funktion die Parameter `msgAuthenticationParameters`, Abbildung 3.6, für das noch verbleibende Feld innerhalb der *msgSecurityParameters*, Abbildung 2.16 (Details zur Funktion `authenticate()` in Abschnitt 3.5.4).

3.5.4. Das USM Sicherheitsmodul

Das USM Sicherheitsmodul besitzt drei Schnittstellen, die erste wird für das Entschlüsseln und Authentifizieren einkommender Nachrichten verwendet, außerdem wird eine Timeliness Überprüfung durchgeführt. Die zweite Schnittstelle dient zum Verschlüsseln ausgehender Nachrichten, die dritte Schnittstelle zum Authentifizieren von ausgehenden Nachrichten.

Entschlüsseln und Authentifizieren von eingehenden Nachrichten

⇒ USM Sicherheitsmodul Schnittstelle 1:

```
s8t processIncomingMsg_USM(u8t *const input, const u16t input_len,
u16t *pos, message_v3_t *request)
```

Aufruf von: `prepareDataElements_v3()` (msg-proc-v3.c) Nachrichtenverarbeitungsmodul, Abschnitt 3.5.3

Wird das USM Sicherheitsmodul für das Verarbeiten einer ankommenden Nachricht aufgerufen, so werden zuerst die *msgSecurityParameters*, siehe Abbildung 2.16, mithilfe der Funktion `decode_USM_parameters()` und den typspezifischen BER Funktionen decodiert. Es werden dann die *AuthoritativeEngineID* und der *msgUserName* überprüft. War die Überprüfung erfolgreich und ist in den *msgFlags* Authentication aktiviert, so wird mittels der Funktion `isBadHMAC()` die Nachricht authentifiziert. Dazu generiert die Funktion `isBadHMAC()` unter Nutzung der Funktion `hmac_md5_96()` die HMAC Ausgangsfolge (Abschnitt 2.3.8). `Hmac_md5_96()` verwendet zur MD5 Hashwertgenerierung die Funktionen aus der Datei `md5.c` sowie den Authentication Key aus der Variable `u8t authKul[16]`. Dieser ist in der Datei `keytools.c` hinterlegt und über die Funktion `getAuthKul()` abrufbar. Da laut Abschnitt 2.3.8 zur vollständigen Authentifizierung eine Timeliness Überprüfung gehört, wird bei erfolgreicher HMAC Überprüfung diese durchgeführt. Dazu wird zuerst der Wert der lokalen Variable `AuthoritativeEngineBoots` mit dem Wert `msgAuthoritativeEngineBoots` der eingehenden Nachricht verglichen. Des Weiteren muss der Wert `msgAuthoritativeEngineTime` der Nachricht mit der aktuellen Systemzeit verglichen werden. Dieser darf nur um 150s abweichen (Kapitel 2.3.8). Umgesetzt wird dies durch eine Absolutwertbildung von `msgAuthoritativeEngineTime - getSysUpTime()` und anschließenden Vergleich mit 150[21]. Die Funktion `getSysUpTime()` wurde in der Datei

[19] *msgSecurityModel, msgFlags, msgMaxSize, msgID, msgVersion* und natürlich der Nachrichtenbeginn mit *Sequence*

[20] Wenn Authentication innerhalb der *msgFlags* aktiviert ist, siehe Tabelle 2.6

[21] Leider ist an dieser Stelle im Programmcode ein kleiner Fehler unterlaufen und die Überprüfung von msgAuthoritativeEngineTime ist in der momentanen Implementierung nicht funktionsfähig, dies wurde in Abschnitt 4.2.3 jedoch behoben.

snmpd.c definiert und liefert die aktuelle Zeit die seit dem Systemstart vergangen ist in ms zurück.
Liegt der Wert der msgAuthoritativeEngineTime innerhalb der Grenzen, so kommt nun das Privacy Modul zum Einsatz. Contiki SNMP verwendet den AES-CFB 128, es ist deshalb erforderlich zu Entschlüsselung des PDUs zuerst den sogenannten Initialisierungsvektor zu generieren (siehe Kapitel 2.3.8). Hierzu wird unter anderem die Funktion convert_-2_octets() verwendet. So wird aus den Werten von msgAuthoritativeEngineBoots, msgAuthoritativeEngineTime und der in den msgPrivacyParameters.ptr enthaltenen *saltValue* der Initialisierungsvektor generiert. Danach wird zum Beginn der Entschlüsselung die Funktion aes_process() gestartet. Sie ruft die Funktion AES_set_encrypt_-key(), welche die für AES notwendige Schlüsselexpansion durchführt, auf. Außerdem startet sie im Anschluss die Funktion AES_cfb128_encrypt(). Beide Funktionen sind definiert in der Datei aes_cfb.c. Die Funktion AES_cfb128_encrypt() ist die eigentliche Verbzw. Entschlüsselungsfunktion, über den Parameter const u8 enc kann der jeweilige Modus gewählt werden, wobei High für Encryption steht und Low für Decryption. In diesem Fall, da eine Nachricht entschlüsselt werden soll, muss der Parameter 0 sein. Dies wird über das Makro AES_DECRYPT erledigt[22]. Die verwendeten Funktionen und Konstanten in der Datei aes_cfb.c entsprechen Eins zu Eins den Originalen welche durch die OpenSSL Library zur Verfügung gestellt werden.

Auf eine detailliertere Beschreibung der AES CFB OpenSSL Library wird an dieser Stelle verzichtet, da dies den Rahmen der Arbeit überschreiten würde. Auch im folgenden Abschnitt "Verschlüsseln und Authentifizieren von ausgehenden Nachrichten" wird diese Beschränkung beibehalten. Für detaillierte Informationen wird auf die Website des Open SSL Projects[23] sowie auf Literatur von [Stallings, 2010] und die Standards der NIST[24] [of Standards and Technology, 2001b,a] verwiesen.

Das verwendete Privacy Passwort wird mithilfe der Funktion getPrivKul() abgefragt. Definiert ist das Privacy Passwort in der Datei keytools.c in der Variablen u8t privKul[16]. Nach Beendigung der Funktion processIncomingMsg_USM() ist innerhalb des Input Buffers an der Stelle, an der zuvor der verschlüsselte PDU zu finden war, nun der entschlüsselte PDU zu finden. Dieser ist jedoch noch BER codiert und muss von der unterliegenden Funktion decodiert werden.

→ Hinweis: laut RFC3826 [Blumenthal et al., 2004], muss die 64Bit Integer Variable zur Generierung des IV eine Zufallszahl sein, welche beim Bootvorgang der SNMP Engine erstellt wird. Contiki SNMP nutzt in seiner derzeitigen Implementierung jedoch zwei fest programmierte hexadezimale Zahlenfolgen mit je 32Bit (Datei snmpd-conf.c, Zeilen 32 und 33 u32t privacyLow = 0xA6F89012;, u32t privacyHigh = 0xF9434568;).
Des Weiteren ist die in RFC3414 [Blumenthal and Wijnen, 2002] beschriebene *msgAuthoritativeEngineBoots* in Contiki SNMP nicht korrekt umgesetzt worden. Laut RFC3414 muss diese Variable als Wert die Anzahl der Bootvorgänge der SNMP Engine seit dem letzten Softwareupdate beinhalten. In Contiki SNMP wird dieser Wert über die Funktion u32t

[22] Datei aes.h, #define AES_DECRYPT 0
[23] http://www.openssl.org
[24] National Institute of Standards and Technology

getMsgAuthoritativeEngineBoots(), definiert in der Datei snmpd-conf.c, abgefragt. Diese Funktion liefert als Rückgabewert jedoch immer Null.
Ebenfalls ist in RFC3414 [Blumenthal and Wijnen, 2002] die Überprüfung des Sicherheitslevels passend zum Benutzernamen (securityLevel) definiert. In der momentanen Implementierung von Contiki SNMP wurde diese Überprüfung nicht umgesetzt. Das einzige Kriterium ist momentan der korrekte Benutzername. Ist der Agent und der Benutzer mit der Sicherheitseinstellung *AuthPriv* konfiguriert, so ist es trotzdem möglich ohne Authentication Passwort und ohne Privacy Passwort MIB Objekte zu setzen und zu empfangen. Das Gleiche gilt für das Sicherheitslevel *AuthNoPriv*, auch hier ist ein MIB Zugriff ohne Authentication Passwort möglich.

In Kapitel 4 wurden diese Implementierungen RFC konform korrigiert.

Verschlüsseln und Authentifizieren von ausgehenden Nachrichten

⇒ USM Sicherheitsmodul Schnittstelle 2:

```
s8t prepareOutgoingMsg_USM(message_v3_t *message, u8t *output,
u16t output_len, s16t *pos)
```

Aufruf von: encode_v3_response() (msg-proc-v3.c) Nachrichtenverarbeitungsmodul, Abschnitt 3.5.3

⇒ USM Sicherheitsmodul Schnittstelle 3:

```
s8t authenticate(message_v3_t *message, u8t *output, u16t
output_len)
```

Aufruf von: encode_v3_response() (msg-proc-v3.c) Nachrichtenverarbeitungsmodul, Abschnitt 3.5.3

Zuerst wird die Schnittstelle 2 aufgerufen, sie bietet die Möglichkeit einen SNMP PDU zu verschlüsseln. Dazu werden im ersten Schritt die Werte der Variablen msgAuthoritativeEngineID, msgAuthoritativeEngineBoots und msgAuthoritativeEngineTime in die message_v3_t Struktur gespeichert. Im zweiten Schritt wird die Funktion encode_USM_parameters(), auch definiert in usm.c, gestartet. Diese generiert den IV wie in Kapitel 2.3.8 beschrieben mithilfe der Funktion convert_2_octets(), definiert in utils.c und startet im Anschluß die Funktion aes_process(), welche auch im vorherigen Abschnitt verwendet wurde. Diesmal jedoch wird als Modus Parameter const u8 enc der Wert AES_ENCRYPT 1(Verschlüsselung) übergeben. Der weitere Ablauf entspricht dem besprochenen aus dem vorherigen Abschnitt "*Entschlüsseln und Authentifizieren von eingehenden Nachrichten*".
Im letzten Schritt codiert die Funktion encode_USM_parameters() den nun verschlüsselten PDU sowie die anderen USM Parameter, die Bestandteil des *Sequence* Feldes *msgSecurityParameters* sind (siehe Abbildung 2.16), mithilfe der BER und den typspezifischen Codierfunktionen aus der Datei ber.c. Das Nachrichtenverarbeitungsmodul hat nun den fertig verschlüsselten PDU, sowie das Feld *msgSecurityParameters* zurückerhalten.

Ist das Authentication Flag innerhalb der *msgFlags* gesetzt, so wird vom Nachrichtenverarbeitungsmodul über die Schnittstelle 3 die Funktion authenticate() gestartet. Diese wie-

derum startet die HMAC Funktion `hmac_md5_96()`, wodurch die Berechnung der HMAC Ausgangsfolge begonnen wird. Nach Abschluß wird der Wert über den Zeiger innerhalb der `msgAutenticationParameters`, welche eine `ptr_t` Struktur darstellt, in den Output Buffer geschrieben. Damit ist die Aufgabe des Sicherheitsmoduls beendet.

3.5.5. Das Command Responder Modul

⇒ Schnittstelle:

```
s8t handle(message_t * message)
```

Aufruf von: `dispatch()` (dispatcher.c) Dispatcher, Abschnitt 3.5.2

Gestartet vom Dispatcher, übernimmt der Command Responder nun die Aufgabe je nach PDU Typ die zugehörige Funktion aufzurufen. Unterstützt werden in der derzeitigen Implementierung die PDU Typen *Set*, *Get* und *Get-Next*, (siehe Kapitel 2.3.7). Die Zuordnung erfolgt über die als Übergabeparameter übermittelte `message_t` Struktur und die dort beinhaltete Variable `message->pdu.request_type`.

Set-Request

Handelt es sich bei dem zu verarbeitenden PDU Typ um einen *Set-Request*, so wird die Funktion `snmp_set()` gestartet. Die Funktion `snmp_set()` erstellt zuerst eine Liste der Variable Bindings, die in der SNMP Nachricht enthalten sind. Dazu werden eine `varbind_t` Struktur und zwei Zeiger auf eine `mib_object_list_t` Struktur, einen Zeiger auf eine `varbind_list_item_t` Struktur sowie ein Zeiger auf eine `mib_object_t` Struktur deklariert.(Aufbau der Strukturen siehe Abbildungen 3.10 und 3.7).
Der `varbind_list_item_t` Zeiger bekommt die Startadresse des ersten Variable Bindings innerhalb der Nachricht. Er dient als Kriterium für die folgende `while()` Schleife, welche solange durchlaufen wird bis keine weiteren Variable Bindings innerhalb der SNMP Nachricht enthalten sind. Dazu wird er am Ende jedes Schleifendurchlauf auf die `next_ptr` Adresse gesetzt, er zeigt also am Ende auf das nächste `varbind_list_item_t`. Das aktuell bearbeitete Variable Binding wird in die zuvor deklarierte `varbind_t` Struktur kopiert und dann über die Funktion `mib_get()` der Management Information Base übergeben (Abschnitt 3.5.6). Die MIB liefert nun die Adresse des gesuchten MIB Objekts zurück und die `snmp_set()` Funktion kopiert diese in den zu Funktionsbeginn deklarierten `mib_object_t` Zeiger.
Dieser `mib_object_t` Zeiger wird an die Funktion `mib_object_list_append()` übergeben, diese ist in der Datei `utils.c` definiert. Sie alloziert über die Funktion `malloc()` Speicherplatz für das MIB Objekt und liefert als Rückgabe einen Zeiger auf eine `mib_object_list_t` Struktur. Beim ersten Schleifendurchlauf wird die Adresse dieses Zeigers in den ersten `mib_object_list_t` Zeiger kopiert. Beim zweiten und N-ten Durchlauf wird die Adresse immer in den zweiten `mib_object_list_t` Zeiger kopiert, wobei innerhalb der zuvor bearbeiteten Struktur in das `next_ptr` Feld, die Adresse der aktuellen `mib_object_list_t` Struktur gespeichert wird. Durch diesen Ablauf erhält man eine zusammenhängende Liste von `mib_object_list_t` Strukturen, ähnlich der Variable Binding Liste, dargestellt in Abbildung 3.8. Durch den ersten `mib_object_list_t` Zeiger ist der Anfang der Liste bekannt und durch die Verknüpfung der einzelnen `mib_object_list_t`

Strukturen über das next_ptr Feld kann ein Zusammenhang hergestellt werden. Das Ende der Liste wird durch den Zeiger next_ptr in der letzten mib_object_list_t Struktur durch den Wert Null markiert.
Zum Abschluss werden über die MIB Funktion mib_set() die Änderungen innerhalb der MIB vorgenommen.

Get-Request

Die Bearbeitung eines *Get-Requests* ist im Gegensatz zur Bearbeitung des *Set-Requests* viel einfacher. Es wird zu Anfang ein varbind_list_item_t Zeiger deklariert, welcher die Adresse des ersten Variable Bindings innerhalb der SNMP Nachricht bekommt. Dieser Zeiger wird wieder als Kriterium für die folgende while()-Schleife gesetzt und bei jedem Schleifendurchlauf auf sein next_ptr Feld gesetzt. Die Schleife wird also solange durchlaufen bis im letzten Variable Binding das Feld next_ptr den Wert 0 enthält. Bei jedem Durchlauf wird über die MIB Funktion mib_get() der geforderte Wert zur passenden OID zurückgeliefert und in das Variable Binding der Anfrage gespeichert.

Get-Next-Request

Der Ablauf bei der Bearbeitung eines *Get-Next-Requests* durch den Command Responder entspricht dem des *Get-Requests* mit dem Unterschied, dass anstatt der MIB Funktion mib_get() die MIB Funktion mib_get_next() aufgerufen wird. Diese liefert das lexikographisch nächste Objekt zum jeweils übermittelten Variable Binding.

→ Die detailiertere Beschreibung der MIB Funktionen folgt in Kapitel 3.5.6

3.5.6. MIB

Innerhalb der MIB sind die Objekte gespeichert, welche vom SNMP Agent verwaltet werden. Die MIB bietet diverse Schnittstellen um diese zu Setzen und Abzufragen. Die folgenden Abschnitte bieten einen Einblick über die Funktionen der vorhanden Schnittstellen. Sämtliche MIB Schnittstellen sind in der Datei mib.c definiert. Die Konfiguration der MIB, also das Hinzufügen bzw. das Entfernen von Objekten wird in Abschnitt 3.5.7 beschrieben.

Zurückliefern eines MIB Objekts passend zur OID

⇒ MIB Schnittstelle 1:

```
mib_object_t * mib_get(varbind_t *req)
```

Aufruf von: snmp_get() (cmd-responder.c) Command Responder, Abschnitt 3.5.5

Um einen Wert zur passenden OID zu bekommen, startet der Command Responder die oben angegebene Funktion. Je nachdem wie die MIB initialisiert wurde, wird für die Objekte entweder dynamisch oder statisch der Speicherplatz alloziert. Dies wird durch das Makro MIB_SIZE innerhalb der Datei mib.h festgelegt. Bei MIB_SIZE 0 wird der Speicherplatz dynamisch alloziert, bei 1 wird statischer Speicher verwendet.
Die Funktion mib_get() startet die sogenannte Iterator Funktion mib_iterator_init-(), wodurch der Zeiger ptr auf das erste MIB Objekt gesetzt wird. Als nächstes wird eine

`while()`-Schleife gestartet, deren Bedingung an die Funktion `mib_iterator_is_end()` geknüpft ist. Solange der Rückgabewert der Funktion `!true` ist, wird die Schleife weiter ausgeführt. Wurde der Speicher der MIB dynamisch alloziert, so ist der Wert immer 0. Bei statischer MIB jedoch wird überprüft ob der aktuelle Index des MIB Objekt Arrays unterhalb der MIB Objekt Gesamtgröße liegt. Es wird im Anschluss kontrolliert, ob es sich beim momentanen MIB Objekt um ein skalares oder ein tabulares Objekt handelt (siehe Kapitel 2.3.2). Bei tabularen Objekten wird jede OID der Tabelle mit der an die Funktion übergebenen OID verglichen. Dies wird durch die Funktion `oid_cmp()`, welche auf `memcmp()` basiert und in der Datei `utils.c` definiert ist, erledigt. Das nächste Tabellenobjekt wird jeweils durch den Zeiger `get_next_oid_fnc_ptr` innerhalb der aktuellen `mib_object_t` Struktur, zugänglich über `ptr`, erreicht, siehe Abbildung 3.10. Skalare Objekte werden mit der gleichen Funktion verglichen. Das nächste MIB Objekt wird über die Funktion `mib_iterator_next()` aufgerufen und anschließend wiederholt sich die oben besprochene Prozedur. Der Aufruf des nächsten MIB Objekts gestaltet sich je nach Speicherallozierung wieder unterschiedlich. Bei dynamischem Speicher wird das nächste MIB Objekt durch den Zeiger `next_ptr` innerhalb der `mib_object_t` Struktur aufgerufen. Bei statischem Speicher wird die Indexvariable des MIB Arrays erhöht.
Wenn die passende OID gefunden wurde, wird die `while`-Schleife unterbrochen und der aktuelle Wert des MIB Objekts wird über die zugehörige get-Funktion abgerufen und in das Variable Binding des *Requests* gespeichert. Der Zugang zur GET Funktion erfolgt über den Funktionszeiger `get_fnc_ptr` des Typs `get_value_t`, siehe Listing 3.13.

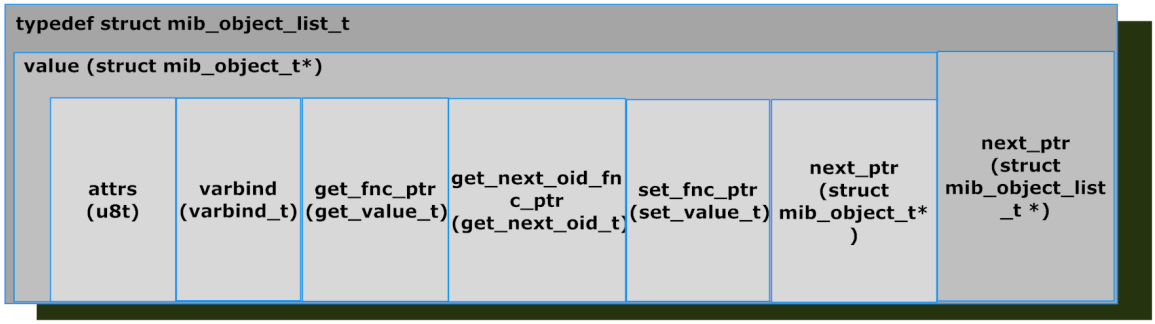


Abbildung 3.10.: Darstellung eines MIB Listenobjekts

Zurückliefern des lexikographisch nächsten MIB Objekts

⇒ MIB Schnittstelle 2:

```
mib_object_t * mib_get_next(varbind_t *req)
```

Aufruf von: `snmp_get_next()` (cmd-responder.c) Command Responder, Abschnitt 3.5.5

Das prinzipielle Vorgehen unterscheidet sich kaum von dem Vorgehen, welches bei der Funktion `mib_get()` im vorherigen Abschnitt angewandt wurde. Da alle MIB Objekte in aufsteigender OID Reihenfolge in die MIB gespeichert werden müssen, kann also, wenn beim ersten MIB Objekt begonnen wird, immer die aktuelle OID der MIB mit der OID des *Requests* verglichen werden. Ist die OID des *Requests* kleiner als die aktuell aufgerufene OID des MIB

Listing 3.13: Funktionszeiger Deklarationen MIB, Auszug mib.h

```
1  //Funktionszeiger Deklaration GET Funktion
2  typedef s8t(*get_value_t)(mib_object_t* object, u8t* oid, u8t len);
3
4  //Funktionszeiger Deklaration GET-NEXT OID (Tabelle) Funktion
5  typedef ptr_t* (*get_next_oid_t)(mib_object_t* object,
6          u8t* oid, u8t len);
7
8  //Funktionszeiger Deklaration SET Funktion
9  typedef s8t(*set_value_t)(mib_object_t* object, u8t* oid,
10         u8t len, varbind_value_t value);
```

Objekts, so muss dies das nächste Objekt innerhalb der MIB sein.
Hier kommt nochmals die Funktion `oid_cmp()` zum Einsatz, sie bietet nämlich eine weitere Funktion außer der Vergleichs-Rückmeldung. Ist die erste übergebene OID kleiner als die zweite, so liefert sie den Wert -1 zurück. Bei gleichen OIDs wird 0 zurückgeliefert und in allen anderen Fällen +1. Wurde das lexikographisch nächste Objekte auf diese Weise gefunden, so wird wieder über den GET Funktionszeiger der Wert des MIB Objekts in das Variable Binding innerhalb des *Requests* kopiert.

Verändern des Wertes eines MIB Objekts

⇒ MIB Schnittstelle 3:

```
s8t mib_set (mib_object_t *object, varbind_t *req)
```

Aufruf von: `snmp_set()` (cmd-responder.c) Command Responder, Abschnitt 3.5.5

Das Setzen eines MIB Objekts gestaltet sich relativ einfach, so wird zuerst überprüft ob im enthaltenen MIB Objekt ein SET Funktionszeiger definiert wurde. Ist dies der Fall, so wird die zugehörige Funktion aufgerufen (SET Funktionszeiger Definition siehe Listing 3.13). Die Definition der objektspezifischen Set Funktion ist in der Datei `mib-init.c` enthalten. Dies wird in Kapitel 3.5.7 detaillierter erläutert.
Sollte kein SET-Funktionszeiger festgelegt worden sein, so muss zuerst der BER Type ermittelt werden, dies kann über die *Request* Nachricht herausgefunden werden.(`req->value_type`) Ist der BER Type entweder *GAUGE*, *INTEGER* oder *OCTET STRING*, so kann ohne SET Funktion der Wert geändert werden. In allen anderen Fällen wird die Fehlermeldung *BAD_VALUE* zurückgegeben.

Hinzufügen eines MIB Objekts

⇒ MIB Schnittstelle 4:

```
s8t add_scalar(ptr_t* oid, u8t flags, u8t value_type, const void*
const value, get_value_t gfp, set_value_t svfp)
```

Aufruf von: `mib_init()` (mib.c) Management Information Base, Abschnitt 3.5.6

⇒ MIB Schnittstelle 5:

```
s8t add_table(ptr_t* oid_prefix, get_value_t gfp, get_next_oid_t
gnofp, set_value_t svfp)
```

Aufruf von: `mib_init()` (mib.c) Management Information Base, Abschnitt 3.5.6

Da es laut Kapitel 2.3.2 zwei Typen von managed Objects geben kann, nämlich tabulare und skalare, ist für jeden Typ eine einzelne Funktion zur Integration eines neuen Objekts in die MIB vorhanden. Schnittstelle 4 bietet die Funktionalität für skalare Objekte und über Schnittstelle 5 ist es möglich tabulare MIB Objekte in die MIB einzupflegen.

Schnittstelle 4, also die Funktion `add_scalar()` legt beim Start je nach Speicherallozierung ein neues MIB Objekt an, siehe Tabelle A.4. Bei statischer Speicherung wird dem Zeiger `object` die Adresse des nächsten Objekts innerhalb des Objektarrays zugewiesen. Bei dynamischer Speicherallozierung wird über die Funktion `mib_object_create()`, welche wiederum `malloc()` nutzt, die Adresse des neuen Speicherplatzes in `object` gespeichert. Im weiteren Funktionsablauf werden der `mib_object_t` Struktur, auf welche der Zeiger `object` zeigt, die Adressen der GET- und SET-Funktion zugewiesen. Außerdem wird bei aktivierter Tabellen Option, siehe ebenfalls Tabelle A.4, der Zeiger `object->get_next_oid_fnc_ptr` auf Null gesetzt[25]. Über das Feld `attrs` innerhalb der `mib_object_t` Struktur werden außerdem die Flags zugewiesen. Wobei über das 7. Bit der Schreib- bzw. Lesezugriff festgelegt wird[26]. Was noch fehlt ist der eigentliche Wert des Objekts bei dessen Initialisierung. Dazu wird die Variable `value` verwendet. Je nach Typ wird entweder die `u_value`(unsigned integer 32-bit), `p_value`(string 32-bit) oder die `i_value`(integer 32-bit) der `typedef union` Struktur, siehe Abbildung 3.7, innerhalb der `varbind_t` Struktur, welche wiederum in der `mib_object_t` Struktur, Abbildung 3.10, liegt, verwendet. Nach dieser Zuweisung, werden noch die Zeiger für OID und Value Typ übergeben und anschließend mithilfe der Funktion `mib_add()` das neue Ende der MIB in der Variablen `mib_tail` gespeichert. Außerdem setzt die Funktion `mib_add()` den Zeiger `next_ptr` innerhalb der `mib_object_t` Struktur des vorherigen MIB Objekts auf das neue MIB Objekt und den `next_ptr` des aktuellen MIB Objekts auf Null.

→ Der Zeiger `next_ptr` innerhalb der `mib_object_t` Struktur zeigt immer auf das folgende MIB Objekt, da das aktuell hinzugefügte MIB Objekt das letzte innerhalb der MIB ist, muss der Zeiger auf Null gesetzt werden um zugreifenden Funktionen zu signalisieren das dies das Ende der MIB ist.

[25] Dieser dient bei tabularen Objekten als Zeiger auf den nächsten Tabelleneintrag.
[26] 7.Bit High,READ-WRITE, 7.Bit Low, READ-ONLY

Soll ein tabulares Objekt der MIB hinzugefügt werden, so kommt die Funktion `add_table()` zum Einsatz. Sie wird hier als **Schnittstelle 5** der MIB bezeichnet. Nach dem Start der Funktion, wird wieder mithilfe von `mib_object_create()` Speicherplatz alloziert oder der nächste Array Eintrag zugewiesen[27]. Es werden danach die Funktionszeiger gespeichert und innerhalb des Objekts in der `varbind_t`-Struktur wird der Typ auf Null gesetzt.

→ Hinweis zur Definition neuer Tabellen MIB Objekte:

Contiki SNMP überlässt es dem "MIB Einrichter" die einzelnen Einträge der Tabelle zu verwalten. Das heisst, dass z.B. innerhalb der *Get*-Funktion zwischen der Basis OID und der weiteren Spalten bzw. Zeilenaufteilung und der darauf folgenden Zellen OID unterschieden werden muss. Am einfachsten lässt sich dies mithilfe von Switch Case Routinen durchführen. Ein Beispiel hierzu ist in der Datei `mib-init.c` für die Tabelle `oid_if_table` zu finden. Die *Get*-Funktion hierfür lautet `getIf()`. Da die Get Funktionen von tabularen Objekten nicht die gesamte OID sondern einen Zeiger auf das BER codierte OID Array an die Stelle nach der Basis OID bekommen, kann ab dort die Tabelle weiter verarbeitet werden. Abbildung 3.11 verdeutlicht diesen Zusammenhang. OIDs werden immer innerhalb einer `ptr_t` Struktur BER codiert gespeichert, übermittelt wird der Tabellen-Get-Funktion jedoch nicht ein `ptr_t` Zeiger, sondern ein Zeiger des Datentyps `u8t`. Die Funktion `getIf()` verwendet

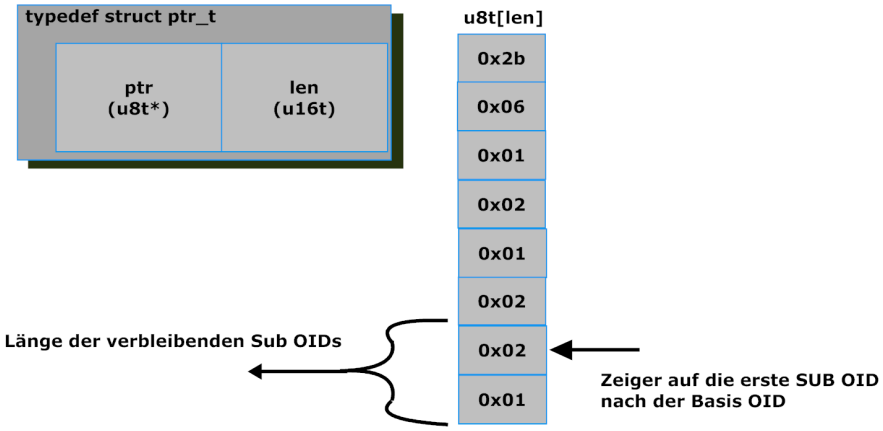


Abbildung 3.11.: Übermittelte Parameter an die Tabellen Get-Funktion

zum Decodieren mithilfe der BER die Funktion `ber_decode_oid_item`. Diese decodiert genau eine Zahl innerhalb der OID und liefert diesen Wert, sowie deren Länge zurück. Somit können auch komplexe Tabellen Stück für Stück aufgebaut werden.
Es ist außerdem zu beachten, dass für jede Tabelle eine Get_Next-Funktion geschrieben werden muss, in dem vorherigen Beispiel der `oid_if_table` ist dies die Funktion `getNextIfOid()`, das Prinzip ist ähnlich zu der Get Funktion, jedoch muss hier immer das Tabellenobjekt nach der übermittelten Sub-OID gefunden werden.

[27] Unterscheidung durch Speicherallozierungsvariante, siehe dazu vorherigen Abschnit

3.5.7. Konfigurationsmöglichkeiten

Contiki SNMP ist vielseitig konfigurierbar, durch den modularen Aufbau ist es grundsätzlich möglich einzelne Module abzuschalten bzw. zuzuschalten. So kann z.B. die SNMPv1 bzw. SNMPv3 Unterstützung einfach deaktiviert werden, wenn sie nicht benötigt wird. Des Weiteren können auch einzelne Teile der SNMPv3 Unterstützung deaktiviert und aktiviert werden, wie z.B. das Authentication Modul oder das Privacy Modul. Eine Übersicht über die möglichen Konfigurationsparameter ist im Anhang unter A.2 zu finden. Um die Passwörter für das Authentication und das Privacy Modul zu ändern, wird ein Passwort Generator benötigt, da die Passwörter auf dem Agenten als MD5 Hashwert gespeichert werden. Ein Passwort Generator ist in der Datei `keygen.c` zu finden.

Logging

In der Datei `logging.c` steht eine Debugging bzw. Logging Möglichkeit zur Verfügung die es erlaubt Fehlermeldungen über eine UDP Verbindung zu versenden. Bei Contiki SNMP wird zwischen zwei Modi unterschieden, einmal der Debug Modus, welcher für die Minimal Platform geschrieben wurde, sowie der Info Modus der für alle anderen Platformen verwendet werden kann. Die Aktivierung des jeweiligen Modus erfolgt über das Compilerflag `#define DEBUG` bzw. `#define INFO` (siehe Kapitel A.2).

MIB Verwaltung

MIB Objekte werden unter Contiki SNMP sofort BER codiert gespeichert, dies spart Resourcen, da die Decodierung und Codierung nicht auf dem Agenten ablaufen muss. In der Datei `oid-encoder.c` ist der Quellcode für einen OID BER Codierer enthalten. Es wird hierin die Vorgehensweise aus Kapitel 2.3.4 umgesetzt.
Sämtliche MIB Objekte sowie deren zugehörige Set und Get Funktionen sind innerhalb der Datei `mib-init.c` definiert. Ein Beispielablauf über das Hinzufügen eines MIB Objekts ist im Anhang unter A.5 zu finden.

3.6. Fazit

In diesem Kapitel wurde zuerst das Betriebssystem Contiki OS beschrieben, es wurden die Funktionen, welche für die Anwendungsentwicklung eines SNMP Agenten von Bedeutung sind besprochen und detailliert erläutert. Speziell die Protothreads, welche auf `switch()` Argumenten basieren, und der IP Stack uIP, der in der neusten Version uIPv6 ein wirkliches Leichtgewicht mit nur 2 Kbyte RAM Verbrauch und nur 11 kByte Quellcode Größe darstellt, wurden diskutiert. Im weiteren Verlauf dieses Kapitels wurde als weiterer Schwerpunkt die Contiki Anwendung *Contiki SNMP* in allen Einzelheiten untersucht und die genaue Funktionsweise, mit Bezug auf das SNMP Grundlagen Kapitel 2.3, dargestellt.

4. 6LoWPAN SNMPv3 Socket

In diesem Kapitel wird der praktische Anteil der Arbeit beschrieben. Im folgenden Abschnitt 4.1 ist eine Übersicht über die angefertigte *6LoWPAN SNMPv3 Steckdose* zu finden, diese wurde auf Basis des in den vorherigen Kapiteln ermittelten Wissens angefertigt und bietet die Möglichkeit 230V Verbraucher über IPv6, gesichert und gesteuert durch SNMPv3, ein- und auszuschalten. Abschnitt 4.2 gibt Auskunft über die benötigten Softwareänderungen des Contiki SNMP aus Kapitel 3.5. Und im darauffolgenden Abschnitt 4.3 werden die durchgeführten Tests der 6LoWPAN Steckdose erläutert und analysiert.

4.1. Übersicht

Abbildung 4.1.: 6LoWPAN 802.15.4 Steckdose

Durch das Wissen aus den vorherigen Kapiteln, ist es nun möglich einen SNMP Agenten mit Contiki SNMP auf einer Hardwareplattform zu implementieren. Erste Versuche wurden im Laufe dieser Arbeit auf der AVR Raven Plattform (siehe Anhang A.12.2) durchgeführt, wofür Contiki SNMP ursprünglich programmiert war. Das Ziel war jedoch die Entwicklung einer über SNMPv3 steuerbaren Funksteckdose mit möglichst kleinen Dimensionen. Aus diesem Grund wurde ein *AVR Zigbit Modul* mit der Typenbezeichnung ATZB-24-A2 (siehe Anhang

A.12.3) verwendet. Das AVR Zigbit Modul enthält einen 8 Bit *ATmega1281* Prozessor mit 8 Kbyte RAM und einem *AT86RF230* IEEE 802.15.4 Transceiver mit integrierter Antenne, verbaut in kleinste Abmessungen von nur 24mm x 13,5mm.
Um Zugriff auf die Pins des Zigbit Moduls zu bekommen, wurde ein Break-Out-Board in Zusammenarbeit des IPv6 Labors und des Elektroniklabors der Beuth Hochschule entwickelt (siehe Abbildung 4.2)(Platinenlayout, Schaltplan Anhang A.12.4). Dieses wurde zusätzlich mit einem *CP2102* Chip der Firma Silabs (siehe Anhang A.12.5) bestückt. Dieser bietet über einen Mini USB Anschluss einen virtuellen seriellen Kommunikationsport. Da unter Contiki OS printf()-Ausgaben standardmäßig über die serielle Schnittstelle ausgegeben werden, ist es mithilfe des Chips und einer Terminalanwendung möglich Debugging Informationen des Contiki Betriebssystems über USB zu empfangen[1]. Das Break-Out-Board bietet zusätzlich einen 10 poligen JTAG Anschluß über den der Microcontroller programmiert werden kann.

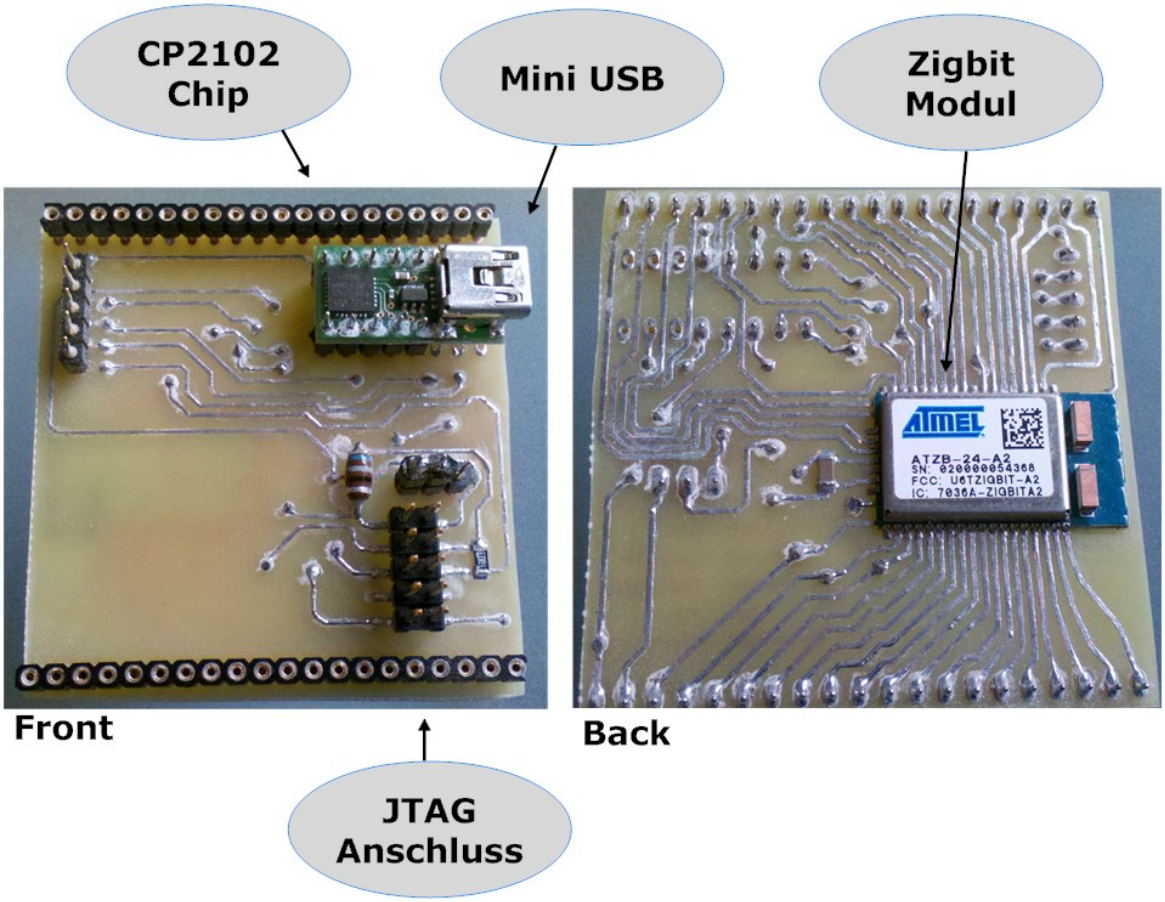


Abbildung 4.2.: Zigbit Platine

Diese Platine wurde wiederum in das Gehäuse einer Analog Funksteckdose eingebaut. Die Stromversorgung erfolgt über ein 5V Netzteil und nutzt den Festspannungsregler des CP2102 Chips um die vom Microcontroller benötigten 3,3V zu liefern. Die Steuerung der Steckdose,

[1]Download der erforderlichen Treiber unter `http://www.silabs.com/products/mcu/Pages/USBtoUARTBridgeVCPDrivers.aspx`

also das Ein- bzw. Ausschalten des Verbrauchers, übernimmt ein Optokoppler der Firma Sharp des Typs *S216S02* (siehe Anhang A.12.6), welcher mit einem zusätzlichen Snubber RC-Glied[2] ausgestattet ist. Angesteuert wird der Optokoppler durch einen Ausgangspin des Mikrocontrollers. Zur Temperaturüberwachung wurde zusätzlich ein NTC Widerstand an den Analog Digital Converter des ATmega1281 angeschlossen. Die Steckdose enthält außerdem einen Reset Taster und LEDs für die Betriebs- und Schaltzustandsanzeige.
Zur Isolierung der spannungsführenden Bauteile wurde Heißkleber eingesetzt. Das Ergebnis des Aufbaus ist in Abbildung 4.3 zu sehen, der komplette Schaltplan ist im Anhang unter A.12.1 enthalten.

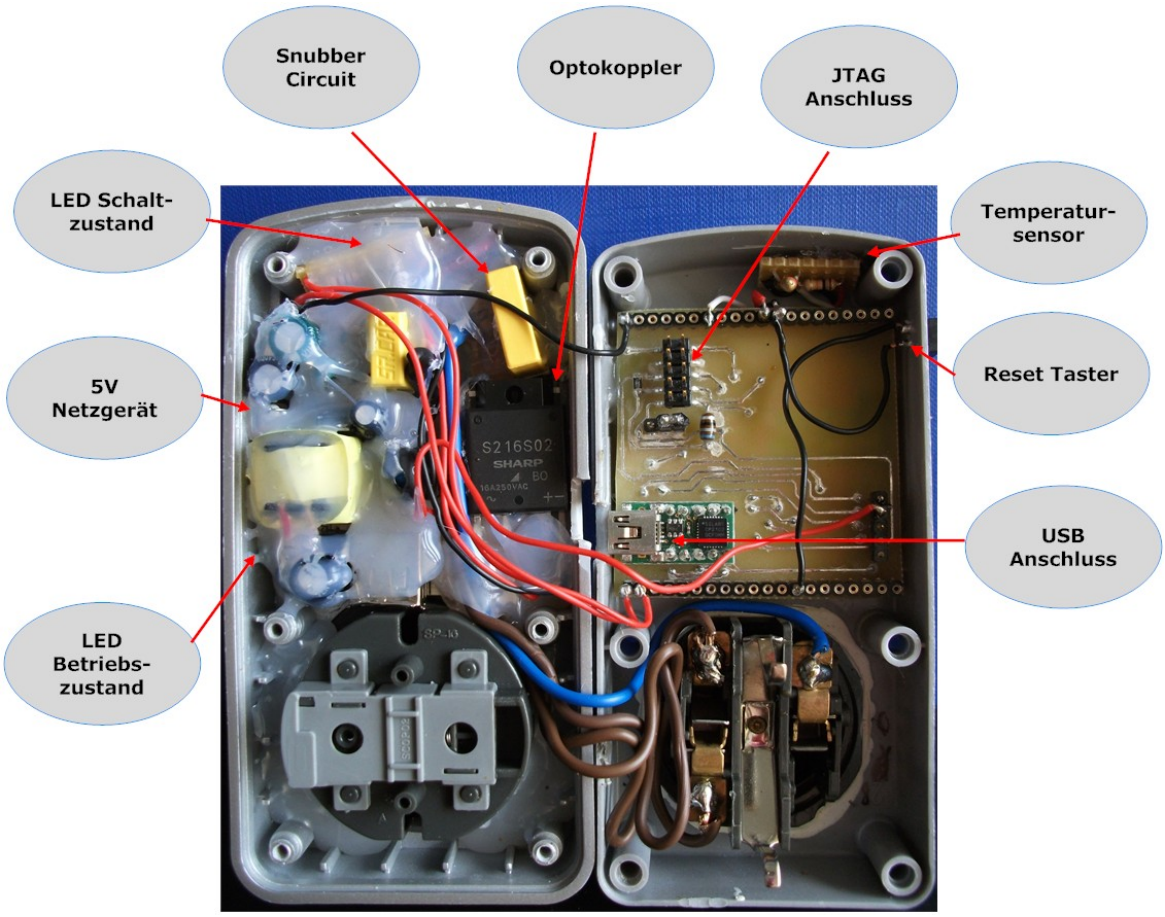


Abbildung 4.3.: Innenansicht der 6LoWPAN SNMP Steckdose

[2]snubber engl. = Dämpfer, auch genannt Boucherot-Glied oder RC-Löschglied, dient zur Beseitigung von Hochfrequenzen oder Spannungsspitzen und zur Begrenzung der Spannungsanstiegsgeschwindigkeit. Hier hauptsächlich aus letzterem eingesetzt, da eine zu hohe Spannungsanstiegsgeschwindigkeit zu der sog. "Über-Kopf" Zündung führen kann, welche den im Optokoppler eingesetzten Triac zerstören könnte.(siehe dazu Datenblatt S216S02 A.12.6)

4.2. Erweiterungen des Contiki SNMP Agents

Die folgenden Seiten dieses Abschnitts enthalten die Dokumentation und Erklärung zu den benötigten Erweiterungen bzw. Anpassungen des Contiki SNMP Agenten aus Kapitel 3.5, welche zur Durchführung dieser Arbeit nötig waren. Sie sind ebenfalls im Internet, im Wiki des IPv6 Labors der Beuth Hochschule Berlin[3], zu finden.

4.2.1. Portierung auf Contiki 2.5 und Contiki 2.6

Das original Contiki SNMP von der Google Code Seite[4] wurde für Contiki 2.4 geschrieben. Da die momentan neuste Version jedoch Contiki in der Version 2.6 ist musste die Software für die neuste Version angepasst werden. Dazu müssen lediglich die Aufrufe der Funktion `HTONS()` angepasst werden. In Contiki 2.5 wurde aus der Funktion `HTONS()` die Funktion `UIP_HTONS()`. Dies muss in den Dateien `logging.c` und `snmpd.c` angepasst werden. Um Contiki SNMP für Contiki 2.6 lauffähig zu kompilieren, muss außerdem in der Datei `snmpd.c` in Zeile 100 der Datentyp `u_8t` in `u8t` geändert werden. Eine komplette Erklärung zur Installation des Agenten mit der genannten Änderung auf der AVR Raven Plattform ist im Anhang unter A.6 nachzulesen.

4.2.2. Portierung auf die AVR Zigbit Plattform

Mit der oben genannten Änderung ist es nun möglich Contiki SNMP auf Contiki 2.5 bzw. auf Contiki 2.6 einzusetzen. Als Hardwareplattform ist jedoch momentan nur der AVR Raven möglich. Da aber das AVR Zigbit Modul verwendet werden soll, sind einige Anpassungen nötig.
Weil die Konstanten des AES Algorithmus und die BER codierten OIDs um Speicherplatz zu sparen vom Datenspeicher in den Programmspeicher verschoben werden sollen, kommen innerhalb des Quellcodes AVR-GCC spezifische Funktionen zum Einsatz. Diese können natürlich nur verwendet werden, wenn auch ein AVR Microcontroller verwendet wird. Aus diesem Grund ist im Quellcode immer dann, wenn AVR spezifische Programmspeicher Funktionen verwendet werden, die Präprozessor Direktive `#if CONTIKI_TARGET_AVR_RAVEN` zu finden. Da aber bei der Kompilierung für das Zigibt Modul diese Bedingung nicht wahr ist, wird anstatt dem Programmspeicher der Datenspeicher verwendet. Dadurch beträgt die Auslastung des Datenspeichers 99,3 Prozent. Dies ist selbstverständlich nicht wünschenswert. Deshalb müssen die Präprozessor Direktiven angepasst werden. Dazu genügt es, die oben genannte Direktive durch `#if CONTIKI_TARGET_AVR_RAVEN || CONTIKI_TARGET_AVR_ZIGBIT` zu ersetzen. Dies ist in den Dateien `utils.c`, `aes_cfb.c` und `mib.c` notwendig.
Außerdem werden in der Datei `snmpd.c` weitere Präprozessor Direktiven verwendet. Hier muss `#if DEBUG && CONTIKI_TARGET_AVR_RAVEN` durch `#if DEBUG && (CONTIKI_TARGET_AVR_RAVEN || CONTIKI_TARGET_AVR_ZIGBIT)` ersetzt werden. Um Zeile 197 innerhalb des `snmpd process` für das Zigbit Modul kompilierbar zu machen, muss `#ifndef CONTIKI_TARGET_AVR_RAVEN` durch `#if !(CONTIKI_TARGET_AVR_RAVEN || CONTIKI_TARGET_AVR_ZIGBIT)` ersetzt werden, da `ifndef` den Ausdruck nicht auswertet und somit bei mehreren Bedingungen nicht verwendet werden kann muss die `if !`

[3] `https://wiki.ipv6lab.beuth-hochschule.de/contiki/snmp`
[4] `http://code.google.com/p/contiki-snmp`

Listing 4.1: UART Einstellungen

```
1  rs232_init(RS232_PORT_1, USART_BAUD_57600,
2          USART_PARITY_NONE | USART_STOP_BITS_1 | USART_DATA_BITS_8);
```

Anweisung verwendet werden.

→ Durch diese Änderungen kann die Nutzung des Datenspeichers von den zuvor genannten 99,3 Prozent auf 57,5 Prozent reduziert werden[5].

Da das Zigbit Break-Out-Board aus Abschnitt 4.1 über USB eine serielle Schnittstelle besitzt, muss diese unter Contiki noch eingerichtet werden. Dazu muss die Datei `contiki--avr-zigbit-main.c` im Contiki Verzeichnis *\platform\avr-zigbit* editiert werden. In Zeile 98 kann die serielle Schnittstelle konfiguriert werden. Die hier eingetragenen Parameter müssen danach beim verwendeten Terminal-Programm natürlich übereinstimmen. Empfohlen werden laut der Contiki Developers Mailing List[6] die folgenden Einstellungen in Listing 4.1, die sich auch bei Versuchen im Rahmen dieser Arbeit als funktionsfähig herausgestellt haben.

Um im Anschluss den neuen Quellcode für das AVR Zigbit Modul zu kompilieren, muss das Makefile angepasst werden, dieser Vorgang ist im Anhang in Listing **??** allgemein beschrieben. Damit das AVR Zigbit Modul genutzt werden kann muss das Kompilierungsziel `TARGET=avr-raven` durch `TARGET=avr-zigbit` ersetzt werden. Danach kann wie im Anhang in A.5 beschrieben für das AVR Raven Board die Kompilierung in gleicher Weise für das Zigbit Modul durchgeführt werden. Wird das Break-Out-Board aus dem vorherigen Abschnitt genutzt, so ist im Anhang in Abbildung A.16 und A.17 der korrekte Anschluss des JTAG-Programmieradapters zu sehen.

4.2.3. Erweiterungen innerhalb des USM

Wie in Kapitel 3.5.4 festgestellt wurde, sind die Implementierungen für die 64Bit Integer Variable zur Generierung des IV und für die Sicherheitsvariable *msgAuthoritativeEngineBoots* nicht mit den zugehörigen RFCs konform[7]. Außerdem fehlte in der original Contiki SNMP Implementierung die Überprüfung des Benutzersicherheitslevels und die Überprüfung der Variablen `msgAuthoritativeEngineTime` wurde nicht korrekt umgesetzt.

64Bit Integer Variable zur Generierung des IV

Laut RFC 3826 [Blumenthal et al., 2004] muss die 64 Bit Integer Variable beim Bootvorgang pseudo zufällig generiert werden. In der momentanen Implementierung ist diese Variable durch zwei 32 Bit Integer `u32t privacyLow` und `u32t privacyHigh` in der Datei `snmpd-conf.c` mit festen Werten definiert. Um für diese Variablen nun zufällige Werte beim Bootvorgang zu generieren, wird die Funktion `random()` der AVR-GCC Library genutzt. Diese wird zuerst mit einem Startwert (Seed) durch die Funktion `srandom()` gesetzt.

[5] Die Programmspeicher Nutzung erhöhte sich durch die Verschiebung von 56,1 Prozent auf 65,7 Prozent. Als Testbetriebssystem wurde Contiki 2.6 verwendet.

[6] `https://lists.sourceforge.net/lists/listinfo/contiki-developers`

[7] 64Bit Integer Variable zur Generierung des IV →RFC3826 [Blumenthal et al., 2004], *msgAuthoritativeEngineBoots* →RFC3414 [Blumenthal and Wijnen, 2002]

Listing 4.2: Funktion u32t get_seed()

```
1  u32t get_seed()
2  {
3      u32t seed = 0;
4      u32t *p = (u32t*) (RAMEND+1);
5      extern u32t __heap_start;
6      #if PDEBUG
7          printf("RAMEND: %X\n__heap_start: %X",RAMEND, &__heap_start)
               ;
8      #endif
9      while (p >= &__heap_start + 1)
10         seed ^= *(--p);
11
12     return seed;
13 }
```

Zur Startwert Generierung wird die Funktion `u32t get_seed()` genutzt. Die Funktion `u32t get_seed()`, siehe Listing 4.2, generiert aus dem Datenspeicher, genauer gesagt aus dem SRAM Bereich des Datenspeichers in dem sich der Heap und der Stack befindet, einen 32 Bit Anfangswert. Da dies beim Start geschieht, ist der Inhalt des SRAMs zufällig. Der Beginn des Heap Speichers wird über die Variable `__heap_start`, welche im zugehörigen Linker Skript definiert ist, markiert. Das Ende des SRAMS wird über das Makro `RAMEND` (`avr/io.h`) gekennzeichnet, siehe Abbildung 4.4.
Es wird nun der gesamte Speicherbereich zwischen `__heap_start` und `RAMEND` stückweise immer wieder mit einer XOR Verknüpfung durchwandert und der Variable `seed` zugewiesen.

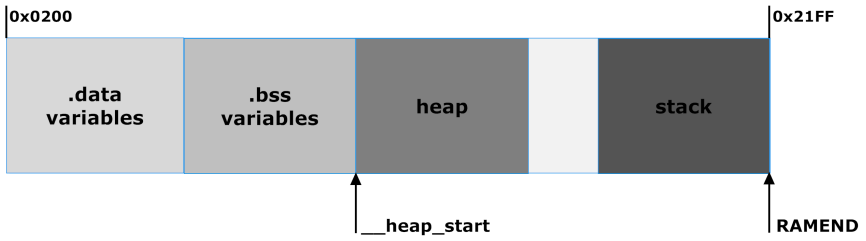


Abbildung 4.4.: Aufteilung des internen SRAM Speichers mit Zeigern auf den Beginn des Heap Speichers und auf das Ende des Stacks

Die Funktion `random()` wird nun nacheinander den beiden Variablen `u32t privacyLow` und `u32t privacyHigh` zugewiesen, wodurch diese nun pseudozufällig sind.

→ Die Variablen sowie die dazugehörigen `get-Funktionen` wurden von der Datei `snmpd--conf.c` bzw. `snmpd-conf.h` in die Dateien `snmpd.c` bzw. `snmpd.h` verschoben, da diese beim Programmstart generiert werden müssen.

Listing 4.3: Funktion incMsgAuthoritativeEngineBoots()

```
1  u8t incMsgAuthoritativeEngineBoots()
2  {
3          /*Increments the Value of MsgAuthoritativeEngineBoots when
               booting.*/
4          /*Checks if the maximum value of 2147483647 (RFC3414) is
              reached.*/
5          if((eeprom_read_dword(&MsgAuthoritativeEngineBoots))
              <2147483647)
6          {
7                  eeprom_update_dword(&MsgAuthoritativeEngineBoots,
                       (eeprom_read_dword(&MsgAuthoritativeEngineBoots
                       )+1));
8          }
9          else
10         {
11                 printf("Maximum Number of
                       MsgAuthoritativeEngineBoots reached, please
                       reconfigure all secret values and reinstall
                       SNMP Agent\n");
12         }
13         return 0;
14 }
```

msgAuthoritativeEngineBoots

Die Variable `msgAuthoritativeEngineBoots` muss laut RFC3414 [Blumenthal and Wijnen, 2002] innerhalb einer nicht flüchtigen Variable gespeichert werden und enthält die Anzahl der Bootvorgänge seit der Installation des SNMP Agenten. Sie wird innerhalb des USM `msgAuthoritativeEngineBoots` zur Generierung des Initialisierungsvektors sowie innerhalb des Timliness Moduls verwendet. In der ursprünglichen Implementierung von Contiki SNMP wird diese Variable einfach immer mit dem Wert Null verwendet. Dies stellt natürlich ein erhebliches Sicherheitsrisko dar.
Es wird deshalb eine 32Bit unsigned Integer Variable innerhalb des *EEPROM*, also im nicht flüchtigen Speicher, des *ATmega1281* angelegt. Um auf den EEPROM zuzugreifen, muss die Datei `<avr/eeprom.h>` miteinbezogen werden. Damit nun bei jedem Neustart des SNMP Agenten `msgAuthoritativeEngineBoots` inkrementiert wird, kommt die Funktion `incMsgAuthoritativeEngineBoots()`, siehe Listing 4.3, zum Einsatz. Sie wird beim Starten des `snmpd_process` in der Datei `snmpd.c` aufgerufen und erhöht die Variable um Eins.

Außerdem wurde die Ausgabe der Funktion `getSysUpTime()`, welche die Zeit für die Variable `snmpEngineTime` liefert, wie in RFC3414 festgelegt, auf einen maximalen Wert von 2147482647 begrenzt. Ist der Wert erreicht, so wird die Variable `seconds` (Zeit in Sekunden seit Systemstart) auf Null gesetzt und `msgAuthoritativeEngineBoots` über

Listing 4.4: usmStatsUnsupportedSecurityLevel

```
1 u8t usmStatsUnsupportedSecurityLevel_array[] = {0x2b, 0x06, 0x01,
    0x06, 0x03, 0x0f, 0x01, 0x01, 0x01};
2 ptr_t usmStatsUnsupportedSecurityLevel = {
    usmStatsUnsupportedSecurityLevel_array, 9};
3 u32t usmStatsUnsupportedSecurityLevelCounter;
```

Listing 4.5: Überprüfung des Sicherheitslevels

```
1 #if ENABLE_AUTH
2         if (!(request->msgFlags & FLAG_AUTH))
3         {
4                 TRY(report(request, &
                      usmStatsUnsupportedSecurityLevel, &
                      usmStatsUnsupportedSecurityLevelCounter));
5         return ERR_USM;
6         }
7 #endif
```

die Funktion `incMsgAuthoritativeEngineBoots()` um Eins erhöht[8].
Des Weiteren fordert RFC3414 bei Erreichen eines Wertes von 2147483647 für `msgAuthoritativeEngineBoots` eine Sperrung des Zugriffs auf den SNMP Agenten. Bei einem Zugriff muss als Fehler eine *notInTimeWindow* Fehlermeldung zurückgesendet werden. Dies wurde innerhalb der Datei `usm.c` realisiert.

Überprüfung des Benutzersicherheitslevels

Laut RFC3414 [Blumenthal and Wijnen, 2002] muss vor dem Verarbeiten einer eingehenden SNMP Nachricht überprüft werden ob das Sicherheitslevel der eingehenden Nachricht mit dem Sicherheitslevel des dazugehörigen Benutzernamens übereinstimmt. Ist dies nicht der Fall, so soll die Nachricht verworfen werden und eine Meldung mit dem Inhalt `unsupportedSecurityLevel` an den Sender zurückgesendet werden. Die original Contiki SNMP Implementierung erfüllt diese Bedingung nicht. Das USM Sicherheitsmodul antwortet immer mit dem Sicherheitslevel, das in der eingehenden Nachricht angegeben ist. Lediglich der Benutzername wird überprüft. Um dieses Problem zu lösen, wurden einige kleine Änderung innerhalb der Datei `usm.c` vorgenommen.
Zuerst musste ein Objekt für die Fehlermeldung `unsupportedSecurityLevel` angelegt werden (siehe Listing 4.4).

Nach diesem Schritt wird innerhalb der Funktion zur Bearbeitung eingehender Nachrichten[9] vor dem Aufruf des Privacy bzw. des Authentication Moduls eine Überprüfung des Sicherheitslevels durchgeführt, sofern dieses innerhalb des Agenten aktiviert wurde, siehe Listing 4.5.

[8] Dies ist ebenfalls festgelegt in RFC3414
[9] `processIncomingMsg_USM()`, siehe Abschnitt 3.5.4

Listing 4.6: Überprüfung der AuthoritativeEngineTime original Contiki SNMP

```
1  if (request->msgAuthoritativeEngineBoots !=
       getMsgAuthoritativeEngineBoots() ||
2        abs(request->msgAuthoritativeEngineTime - getSysUpTime
           ()) < TIME_WINDOW) {
3    TRY(report(request, &usmStatsNotInTimeWindows, &
       usmStatsNotInTimeWindowsCounter));
4    return ERR_USM;
5  }
```

Listing 4.7: Überprüfung der AuthoritativeEngineTime geändertes Contiki SNMP

```
1 abs(request->msgAuthoritativeEngineTime - (getSysUpTime()/100)) >
    TIME_WINDOW
```

Entspricht das Sicherheitslevel der Nachricht nicht dem des dazugehörigen Benutzernamens, so wird mithilfe der Funktion `report()` eine Antwortnachricht mit dem Fehlercode `unsupportedSecurityLevel` an den Absender geschickt. Für das Privacy Modul wurde diese Funktion nach dem gleichen Prinzip umgesetzt.

Korrektur zur Überprüfung der msgAuthoritativeEnigineTime des Timliness Moduls

Innerhalb des USM Moduls befindet sich das Timeliness Modul, dieses dient zur Überprüfung der Authentizität der Nachricht. Dazu muss laut RFC3414 [Blumenthal and Wijnen, 2002] die ankommende `msgAuthoritativeEngineTime` mit der lokalen `AuthoritativeEngineTime` des Agenten verglichen werden. Der Unterschied zwischen beiden Zeiten darf 150 Sekunden nicht überschreiten. Dies wurde im original Contiki SNMP mit der if-Abfrage in Listing 4.6 umgesetzt.

Es wird also eine Absolutwertbildung der Subtraktion der `msgAuthoritativeEngineTime` der ankommenden Nachricht und der Funktion `getSysUpTime()` durchgeführt. Der Wert der Variablen `msgAuthoritativeEngineTime` der eingehenden Nachricht ist in Sekunden angegeben. Und die Funktion `getSysUpTime()` liefert die Zeit seit dem letzten Systemstart in Millisekunden zurück. Dies führt natürlich grundsätzlich zu einem Problem, sodass dieser Wert scheinbar immer größer als 150 Sekunden ist. Dies hat zur Folge, dass die Überprüfung innerhalb der if-Abfrage also immer null ist. Es wird also immer angenommen dass der Wert von `msgAuthoritativeEngineTime` der eingehenden Nachricht innerhalb des Zeitfensters liegt.
Dies wurde durch die Änderung in Listing 4.7 korrigiert.

Alle Änderungen innerhalb des USM sind im Quellcode mit dem Kürzel `/*sz*/` gekennzeichnet.

4.2.4. Managed Objects

Da die 6LoWPAN SNMP Steckdose eine Last zu schalten hat, genügt es natürlich nicht den SNMP Agenten auf den Mikrocontroller zu laden. Die zu schaltende Last muss zuerst als MIB Objekt in die MIB des Agenten eingetragen werden. Die benötigten Funktionen um neue *managed Objects* in eine bestehende MIB zu integrieren, wurden im Abschnitt 3.5.6 vorgestellt. Außerdem ist im Anhang unter A.5 eine Anleitung für das Hinzufügen eines Beispiel MIB Objekts zu finden. In den folgenden Abschnitten werden deshalb nur die *Get-* bzw. *Set-* Funktionen der enthaltenen MIB Objekte beschrieben.

MIB Objekt zur Ansteuerung der Ausgangspins zur Lastschaltung und Schaltzustandsanzeige

Die auch in A.5 enthaltenen Get- und Set- Funktionen zur Ansteuerung der Port Pins, wurden um einen weiteren Pin erweitert, so dass nun zwei[10] Ausgangspins beim Aufrufen der Set Funktion gesetzt werden. Wie in Listing 4.8 zu sehen ist, werden die Pins 1 und 2 von Port G genutzt. Diese entsprechen GPIO5 und GPIO4 im Zigbit Datenblatt[11]. Über das Register `DDRG` werden diese zuerst als Ausgänge festgelegt (Zeile 9,10) und anschließend, wenn der Setztwert im ankommenden Request Eins ist, wird zuerst PIN 2 (Zeile 12) und dann PIN 1 (Zeile 13) von PortG gesetzt. Bei allen anderen Setzwerten innerhalb des Requests werden beide Pins ausgeschaltet (Zeile 15,16).
Die Get-Funktion ist noch einfacher aufgebaut. Nach dem Aufruf wird der aktuelle Wert von PIN2 in das Variable Binding des MIB Objekts geschrieben. Der grundsätzliche Aufbau der Get- und Set-Funktion basiert auf einem Beispiel aus [BHT, 2012]. Der weitere Ablauf vor und nach dem Aufruf der Get- bzw. Set- Funktion wurde in Kapitel 3.5 ausführlich beschrieben.

MIB Objekt zur Abfrage der Empfangsstärke

Um zu bestimmen, ob der Empfang zwischen SNMP Agent und 6LoWPAN Router ausreichend ist, wurde der aktuelle Wert der Funkempfangsstärke über SNMP abfragbar gemacht. Ermöglicht wird dies durch die Funktion `rf230_get_raw_rssi()`, welche sich ab Contiki 2.5 in den Radio Treibern für den *AT86RF230* in der Datei `rf230bb.c` im Contiki Unterverzeichnis *cpu\radio\rf230bb * befindet. Die Funktion `rf230_get_raw_rssi()` nutzt wiederum die Funktion `hal_register_read`, diese ermöglicht es einzelne Register, wie z.B. das für die aktuelle Empfangsstärke, des *AT86RF230* auszulesen. Es werden jedoch Werte von 0-84 dB in 1 dB Schritten zurückgeliefert. Diese Werte sind so zu interpretieren, dass 0 dem schlechtesten Empfangswert in dB entspricht und 84 dem besten Empfangswert. Um nun die Empfangsstärke in dBm zu berechnen, muss -91dBm als Startwert genommen werden und der Wert des Registers addiert werden [SICS, 2012].
In Listing 4.9 wurde dieser Vorgang in einer Get-Funktion umgesetzt.

[10] Da beim Hardwareaufbau noch nicht klar war, wofür die LED in der Vorderseite der 6LoWPAN Steckdose genutzt wird, wurde diese an einen separaten Ausgangspin angeschlossen. Es gab anfangs Überlegungen zur Signalisierung des Paketempfangs durch uIP, diese Idee wurde jedoch schlussendlich verworfen, da die Anzeige des Schaltzustands sinnvoller erschien. Aus rein elektrotechnischen Überlegungen wäre es auch problemlos möglich gewesen beide Ausgangssignale mit einem PIN zu schalten. (Strom LED = 10mA, Strom SharpS216S02 = 20mA, max. Ausgangsstrom pro Pin des ATmega1281 = 40mA, siehe zugehörige Datenblätter im Anhang)

[11] siehe Schaltplan Anhang A.12.1

Listing 4.8: Get- und Set- Funktion zur Laststeuerung

```
1  s8t getBeuthState(mib_object_t* object, u8t* oid, u8t len)
2  {
3      object->varbind.value.i_value = ((PORTG >> PIN2) & 1);
4      return 0;
5  }
6
7  s8t setBeuthState(mib_object_t* object, u8t* oid, u8t len,
       varbind_value_t value)
8  {
9          DDRG |= (1 << PIN2);
10         DDRG |= (1 << PIN1);
11     if (value.i_value == 1) {
12         PORTG |= (1 << PIN2);
13                 PORTG |= (1 << PIN1);
14     } else {
15         PORTG &= ~(1 << PIN2);
16                 PORTG &= ~(1 << PIN1);
17     }
18          return 0;
19 }
```

Listing 4.9: Get-Funktion zur Messung der Funkempfangsstärke

```
1  s8t getRssiValue(mib_object_t* object, u8t* oid, u8t len)
2  {
3          int rssi_temp;
4          rssi_temp=rf230_get_raw_rssi();
5      object->varbind.value.i_value = (-91)+(rssi_temp);
6      return 0;
7  }
```

MIB Objekt zur Temperaturmessung

Zur Überwachung der Temperatur des Gerätes, wird ein NTC Widerstand eingesetzt. Dieser wird mit einem zweiten Widerstand als Spannungsteiler über einer Referenzspannung betrieben[12]. Durch eine Veränderung der Temperatur ändert sich der Widerstand des NTCs. Durch die Widerstandsänderung des NTCs, verschiebt sich das Spannungsverhältnis des Spannungteilers. Diese Spannungsänderung kann mit dem Analog Digital Converter (ADC) des ATmega1281 gemessen werden.
Der ATmega 1281 des Zigbit Moduls besitzt einen 10 Bit Analog Digital Wandler, welcher wahlweise über die Pins *ADCINPUT0 - ADCINPUT3* gemultiplext werden kann. Diese Pins entsprechen dem Port *PF0-PF3* des 1281. Hier wird der Pin *PF3*, d.h. Pin *ADCINPUT3* genutzt. Der Pin *Aref*, welcher die Referenzspannung für den ADC bereitstellt, ist intern über einen Widerstand mit dem Pin *AVCC* verbunden. Die Referenzspannung beträgt somit 3,15 Volt.

Die Abfrage der Temperatur erfolgt auch hier mithilfe einer Get-Funktion. Bevor jedoch die eigentliche get-Funktion ausgeführt werden kann, sind folgende Schritte notwendig:

- Initialisieren des ADC
- Abfrage des ADC Werts
- Bestimmung des aktuellen PTC Widerstands
- Bestimmung der Temperatur

→ Innerhalb der ADC Funktionen werden viele vordefinierte Makros genutzt. Diese stehen durch Einbinden der Datei `avr/io.h` zur Verfügung. Die Bedeutung der einzelnen Register ist im Datenblatt des ATmega1281 zu finden, siehe Anhang A.12.3. Des Weiteren war im Laufe der ADC Programmierung das Tutorial von maxEmbedded [Prasad, 2011] sehr hilfreich.

Initialisieren des ADC: Im Anhang A.9 in Listing A.10 ist die ADC Initialisierungsfunktion zu sehen. Über das *ADMUX* Register wird die Referenzspannung gewählt. Durch das Setzen von *REFS0* auf 1 wird die externe Referenzspannung *AVCC* gewählt. Außerdem kann über das Register *ADCSRA* und das darin enthaltene Bit *ADEN* der ADC aktiviert und über *ADSP0*, *ADPS1* und *ADSP2* der Frequenzvorteiler von 128 gewählt werden (siehe Datenblatt A.12.3). Somit ergibt sich eine Abtastfrequenz von $\frac{4Mhz}{128} = 31,25kHz$, was für die Temperaturerfassung völlig ausreichend ist.

Abfrage des ADC Werts: Um nun einen Wert zu erhalten, muss das ADC Register ausgelesen werden. Hierzu wird die Funktion `adc_read()`, siehe Anhang A.9 in Listing A.12, verwendet. Vor dem Start der Messung muss zuerst der Kanal im Register *ADMUX* festgelegt werden. Anschließend kann durch Setzen des Bits *ADSC* im Register *ADCSRA* die Messung gestartet werden. Dieses Bit bleibt solange 1 bis die Messung abgeschlossen ist. Ist *ADSC* wieder 0 kann der ADC Wert übergeben werden.

[12] siehe Schaltplan der 6LoWPAN Steckdose A.12.1, R5 und R6

Bestimmung des aktuellen NTC Widerstands: Mit den beiden Funktionen `void adc_init()` und `u16t adc_read()` kann nun der aktuelle ADC Wert eingelesen werden. Der im ATmega1281 verwendete ADC besitzt eine Auflösung von 10 Bit, es sind also Werte von 0 bis 1024 möglich. Da die Referenzspannung 3,15 Volt beträgt, entspricht ein ADC Wert von 1:

$$\frac{3,15V}{1024} = 0,003076171875V = 1\frac{V}{ADC} = ADC_{1ADC}. \tag{4.1}$$

Somit ist es also möglich, die aktuelle Spannung am NTC zu bestimmen. Zur Bestimmung des aktuellen Widerstandswerts kann nun die Spannungsteilerformel angewandt werden.

$$\frac{U_{ges}}{R_{ges}} = \frac{U_{NTC}}{R_{NTC}} \tag{4.2}$$

Da $R_{ges} = R_1 + R_{NTC}$ und $U_{NTC} = ADC_{eingelesen} \cdot ADC_{1ADC}$, ergibt sich mit Gleichung 4.2 die Gleichung 4.3.

$$R_{NTC} = \frac{ADC_{eingelesen} \cdot ADC_{1ADC} \cdot R_1}{U_{ges} - (ADC_{eingelesen} \cdot ADC_{1ADC})} \tag{4.3}$$

Diese Berechnung kann im Anhang A.9 in Listing A.11, umgesetzt in die Funktion `adcToOhm()`, betrachtet werden.

Bestimmung der Temperatur: Leider besitzt ein NTC kein lineares Temperaturverhalten. Die Berechnung erweist sich aus diesem Grund als sehr kompliziert und würde zu viele Ressourcen in Anspruch nehmen. Es wurde deshalb ein Array mit zuvor berechneten Werten aus dem Datenblatt des NTC verwendet, siehe dazu Anhang A.12.7. Das Datenblatt enthält zuvor berechnete Werte im Format $\frac{R_{aktuelleTemp.}}{R_{Temp.bei25^\circ}}$ im Abstand von 5 Grad Celsius. Diese Werte wurden zur genaueren Darstellung interpoliert und in ein Array im Programmspeicher gespeichert, siehe Anhang A.9 unter Listing A.13.
Im nächsten Schritt kann nun der aktuelle Widerstandswert mit dem im Array verglichen werden und so die aktuelle Temperatur bestimmt werden. Dazu wird die Funktion `u8t find_temp_celsius(float r_ntc)`, aus Listing A.14 im Anhang unter A.9, verwendet. Die Funktion vergleicht den aktuellen Temperaturwert mit den im Array gespeicherten Temperaturwerten und zählt dabei mit einer Zählvariable die Position innerhalb des Arrays. Das Array enthält Werte im Bereich von 0 bis 100 Grad Celsius. Die Variable `i` liefert so den aktuellen Temperaturwert in Grad Celsius zurück.

Get Funktion: Somit können nun alle zuvor besprochenen Schritte durchgeführt werden. Um die einzelnen Funktionen zu starten und den ermittelten Temperaturwert in das MIB Objekt zu speichern wird die Get-Funktion in Listing 4.10 verwendet.

weitere MIB Objekte

Weitere MIB Objekte, welche in der verwendeten MIB enthalten sind, stammen aus dem orginal Contiki SNMP. Enthalten sind Teile der in RFC1213 [McCloghrie and Rose, 1991] definierten *System Group* sowie der *SNMP Group*. Eine Auflistung der implementierten managed Objekts mit OID ist im Anhang A.8 zu finden.

Listing 4.10: Get-Funktion zur Temperaturmessung

```
1  s8t getTempValue(mib_object_t* object, u8t* oid, u8t len)
2  {
3          adc_init();
4          u16t adc_value;
5          float r_ntc;
6          adc_value=adc_read();
7          r_ntc=adcToOhm(adc_value);
8      object->varbind.value.i_value = find_temp_celsius(r_ntc);
9      return 0;
10 }
```

4.3. Evaluierung / Tests

In diesem Abschnitt soll die entwickelte 6LoWPAN SNMPv3 Steckdose genau untersucht und geprüft werden. In Abschnitt 4.3.1 werden zuerst die Eckdaten, beginnend bei der Ermittlung des Speicherverbrauchs, der Entwicklung ermittelt. Im weiteren Verlauf dieses Abschnitts werden der Energieverbrauch sowie die Transfer- und Rechenzeiten bestimmt.
Im darauffolgende Abschnitt 4.3.2 werden die Funktionen des Agenten anhand eines Testplans überprüft, wobei der Schwerpunkt hierin auf die Sicherheit durch SNMPv3 gelegt wird.

4.3.1. Leistungstests

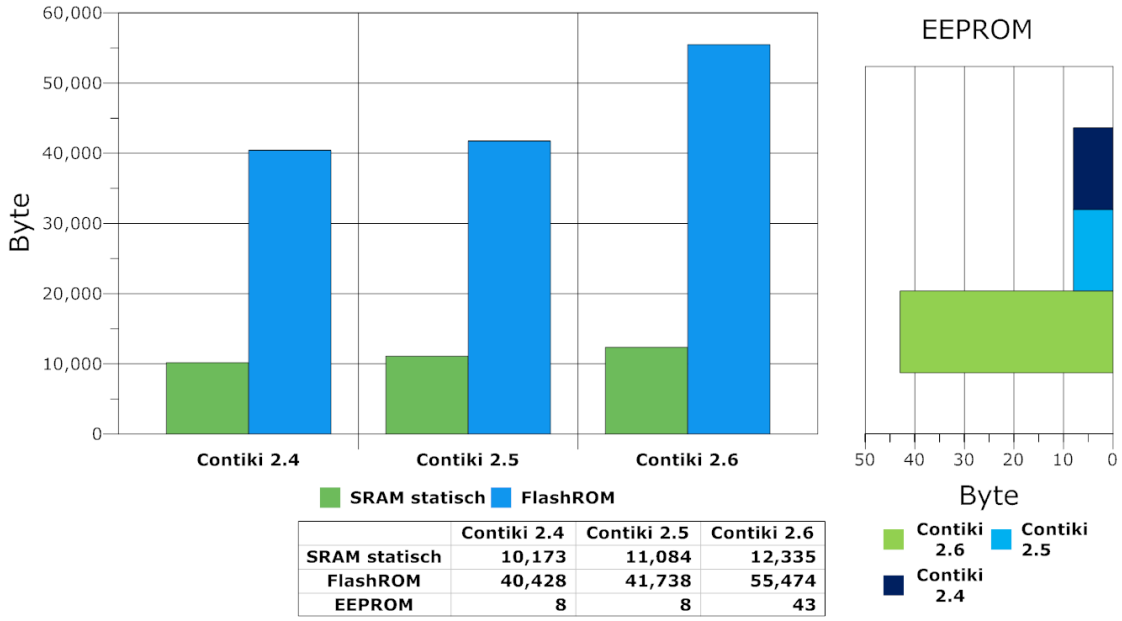
Für die in diesem Abschnitt folgenden Tests wurde teilweise die AVR Raven Plattform (A.12.2) zum größten Teil jedoch das AVR Zigbit Modul (A.12.3) eingesetzt, dies ist jedoch dementsprechend gekennzeichnet.

Speicherverbrauch

Um den Speicherverbrauch im Flash-ROM und innerhalb des SRAM zu bestimmen, wird das Werkzeug *avr-size* des Win AVR Pakets[13] genutzt. In Abbildung 4.5 ist der Speicherverbrauch der unterschiedlichen Contiki Versionen dargestellt. Dies wurde durch das Kompilieren des Contiki Quellcodes mit aktiviertem uIPv6 Support[14] für die AVR Raven Plattform und ohne weitere Programme mit *avr-size* ermittelt. Es ist deutlich zu Erkennen, dass mit Version 2.6 der Speicherbedarf stark angestiegen ist. Dies liegt daran, dass ab Version 2.6 das Flag `UIP_CONF_IPV6_RPL`[15] standardmäßig immer auf Eins gesetzt ist. Dies war in den bisherigen Versionen nicht der Fall. Kompiliert man Contiki 2.6 jedoch mit deaktiviertem `UIP_CONF_IPV6_RPL` Flag[16], so ist der Speicherverbrauch mit 42894 Byte ROM und 11247 Byte statischem SRAM kaum mehr als in Version 2.5. Des Weiteren ist auffällig, dass der EEPROM Verbrauch stark angestiegen ist. Dies liegt an der neu hinzugekommenen Datei `params.c` im Contiki2.6 Unterordner \platform\avr-raven\. In Version 2.6 werden dort

[13] `http://winavr.sourceforge.net/`
[14] Compiler Flag = `UIP_CONF_IPV6=1`
[15] Dieses Flag aktiviert das *Routing Protocol for Low power and Lossy Networks* kurz (RPL), ein Routing Protokoll für IPv6 Sensornetzwerke das auf dem Distanzvektorverfahren basiert.[siehe Tsiftes et al., 2010]
[16] dazu im Makefile `UIP_CONF_RPL=0` eintragen

Netzwerkeinstellungen, wie z.B. die MAC Adresse oder die PanID in den nicht flüchtigen Speicher geschrieben. Diese Funktion ist jedoch nicht für das AVR-Zigbit Modul verfügbar, dort bleibt der EEPROM Verbrauch bei den 8 Byte der Vorgängerversionen.



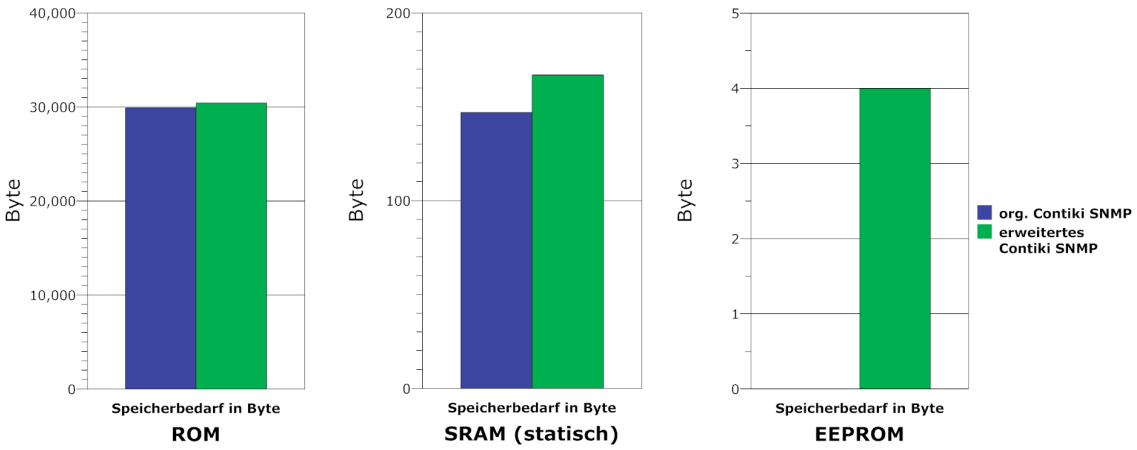
	Contiki 2.4	Contiki 2.5	Contiki 2.6
SRAM statisch	10,173	11,084	12,335
FlashROM	40,428	41,738	55,474
EEPROM	8	8	43

Abbildung 4.5.: Speicherverbrauch der einzelnen Contiki OS Versionen

Um nun den Speicherverbrauch der SNMP Implementierung zu überprüfen, wurde mit Contiki SNMP kompiliert und die Werte aus 4.5 subtrahiert. Dabei wurde nur das SNMP Programm an sich mit einer leeren MIB kompiliert. SNMPv3 im Modus AuthPriv und SNMPv1 wurden beide aktiviert. In Abbildung 4.6 sind die Ergebnisse dieser Subtraktion zu sehen. Die Berechnung wurde zuerst für das original Contiki SNMP von der *code.google* Seite durchgeführt und danach wurde der Speicherverbrauch des Contiki SNMP mit den zuvor in dieser Arbeit besprochenen Änderungen berechnet.

Es ist gut zu erkennen, dass die durchgeführten Änderungen keinen erheblich größeren Speicherverbrauch verursachen. Außerdem decken sich die ermittelten Werte für das orginal Contiki SNMP ungefähr mit denen aus [Kuryla and Schönwälder, 2011, Kuryla, 2010][17], wobei hier beachtet werden muss, dass in [Kuryla and Schönwälder, 2011, Kuryla, 2010] die Kompilierung mit MIB Objekten durchgeführt wurde, die Größe der MIB bei der Speichertestdurchführung ist jedoch nicht angegeben.
Ein weiterer interessanter Aspekt ist der benötigte Stack Speicher der Implementierung. Praktischerweise enthält das original Contiki SNMP schon eine Routine im Programmcode, mit dem der maximal genutzte Stackspeicher während eines Aufrufs der `udp_handler` Funktion, siehe Abschnitt 3.5.1, bestimmt werden kann. Die Funktionsweise basiert auf dem einfachen Prinzip, dass vor dem Funktionsaufruf der Stack mit einem bestimmten Bitmuster beschrieben wird[18] und bei Beendigung der Funktion wird der Stackspeicher solange

[17] ROM = 31220 Byte, RAM = 235 Byte
[18] in diesem Fall mit `0xAAAAAAAA`



	Speicherbedarf in Byte
org. Contiki SNMP	29,914
erweitertes Contiki SNMP	30,420

	Speicherbedarf in Byte
org. Contiki SNMP	147
erweitertes Contiki SNMP	167

	Speicherbedarf in Byte
org. Contiki SNMP	0
erweitertes Contiki SNMP	4

Abbildung 4.6.: Anstieg des Speicherverbrauchs durch die Änderungen im USM

durchsucht bis das Bitmuster wieder auftaucht. Mit diesem Messverfahren wurde für das erweiterte Contiki SNMP mit der Option *AuthPriv* ein maximaler Stackverbrauch von 1432 Byte gemessen. Für das original Contiki SNMP wurde mit dem gleichen Messverfahren und den gleichen SNMPv3 Optionen ein maximaler Stackverbrauch von 1412 Byte gemessen.
Durch die Änderungen, die im Laufe dieser Arbeit vorgenommen wurden, hat sich der Speicherverbrauch im ROM um 506 Byte erhöht, dies entspricht einer Erhöhung um 1,69 Prozent. Der Verbrauch des Datenspeichers, also des Speicherplatzes welcher statisch im SRAM alloziert wird, hat sich um 20 Byte erhöht, also um 13,6 Prozent. Dieser Wert klingt erstmal viel, werden jedoch MIB Objekte in den Agenten integriert, so liegen die 20 Byte Mehrverbrauch im Prozentbereich der Erhöhung des Programmspeichers. Zusätzlich benötigt die erweiterte Version durch die nicht flüchtige Variable *snmpEngineBoots*, siehe Abschnitt 4.2.3, 4 Byte im EEPROM.
Bei der Berechnung des Stack Speichers wurde ein SNMP Objekt in die MIB integriert, um den Speicherverbrauch bei der Bearbeitung eines Get Requests zu messen. Das original Contiki SNMP nutzte, wie zuvor erwähnt, maximal 1412 Byte des Stack Speichers, das erweiterte 1432 Byte. Somit benötigt das erweiterte Contiki SNMP 20 Byte mehr vom Stackspeicher. Dies entspricht 1,42 Prozent Mehrverbrauch.
Nach dem gleichen Prinzip wurde nun der Speicherverbrauch des SNMP Agenten, welcher auf der 6LoWPAN Steckdose installiert ist mit allen implementierten MIB Objekten, siehe A.8, und den unterschiedlichen SNMPv3 Sicherheitsoptionen getestet. Die Ergebnisse sind in Tabelle 4.1 und in Abbildung 4.7 zu betrachten.

Natürlich benötigt die Sicherheitsoption *AuthPriv* die meisten Ressourcen des Mikrocontrollers, jedoch ist nur in dieser Option die maximale Sicherheit gewährleistet. Insgesamt verbraucht die höchste Sicherheitsstufe des SNMP Agenten mit der neusten Contiki Version 2.6 auf der Zielplattform AVR Zigbit 65,6 Prozent des verfügbaren Programmspeichers und 56,8 Prozent des Datenspeichers.

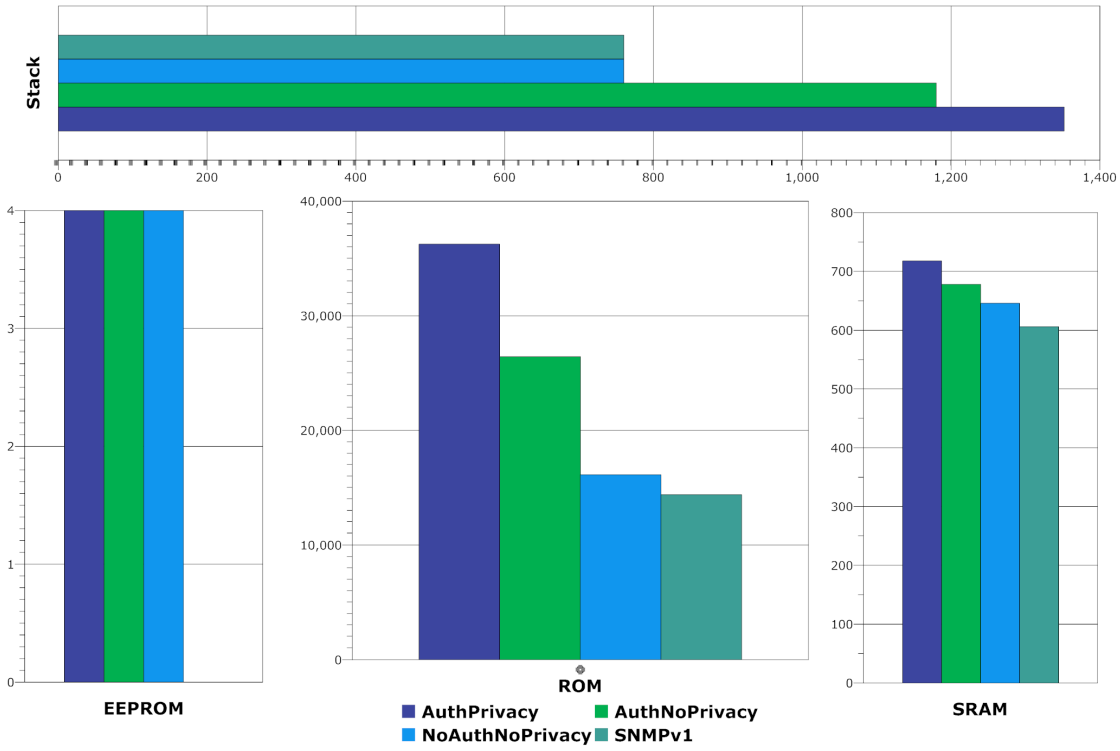


Abbildung 4.7.: Speicherverbrauch von Contiki SNMP bei unterschiedlichen Sicherheitsoptionen (alle Angaben in Byte)

Wird ohne RPL kompiliert, so sind noch 55,2 Prozent des Programmspeichers und 44,1 Prozent des Datenspeichers belegt. Da in diesem Versuchsszenario nur ein Knotenpunkt verwendet wird, ist dies die weiterhin verwendete Konfiguration.
Wird Abbildung 4.7 genauer untersucht, so ist hier deutlich zu sehen, dass der meiste Stackverbrauch durch das Authentication Modul verursacht wird. Während bei SNMPv1 und SNMPv3 mit *NoAuthNoPriv* genau gleich viel Stackspeicher benötigt wird, steigt beim Zuschalten des Authentication Moduls der maximale Speicherbedarf durch den Stack von 760 Byte auf 1180 Byte. Dies entspricht einer Erhöhung von 420 Byte bzw. 55 Prozent. Das Privacy Modul hingegen, benötigt bei Zuschaltung maximal 172 Byte mehr Stack Speicher. Dies entspricht einem Anstieg von 22 Prozent. Werden die Werte auf den Gesamtverbrauch im Modus *AuthPriv* des Stackspeichers verteilt, so benötigen die Grundfunktionen des SNMP Agenten ohne Sicherheit 56 Prozent des maximal genutzten Stackspeichers, während das Authentication Modul 31 Prozent beansprucht. Das Privacy Modul benötigt im Gesamtvergleich lediglich 13 Prozent des gesamten maximal genutzten Stacks.
Beim Speicherverbrauch des ROMs bzw. des Programmspeichers, ist diese Ausprägung nicht so stark zu sehen. Das SNMPv3 Grundprogramm benötigt hier 45 Prozent des durch Contiki SNMP genutzten Programmspeichers. Das Authentication Modul beansprucht von dem Gesamtverbrauch 28 Prozent und durch das Privacy Modul werden die übrigen 27 Prozent genutzt. Hier ist also die Aufteilung des Speicherverbrauchs zwischen Authentication Modul und Privacy Modul ausgeglichen[19]. Wird SNMPv1 verwendet, so sinkt der Programmspeicherverbrauch natürlich erheblich. So werden mit SNMPv1 nur 14388 Byte benötigt, dies entspricht 40 Prozent des durch SNMPv3 im Modus *AuthPriv* benötigten Programmspeichers.
Wird der Verbrauch des SRAM-Speichers[20] betrachtet, so steigt die Nutzung des durch globale Daten verursachten Verbrauchs, pro hinzukommende Sicherheitsoption und im Vergleich zu SNMPv1, um fünf bis sechs Prozent.

Tabelle 4.1.: Speicherverbrauch von Contiki SNMP bei unterschiedlichen Sicherheitsoptionen (alle Angaben in Byte)

Sicherheitslevel	**Program Memory**	**Data Memory**	**EEPROM**	**max. Stack**
AuthPriv	36256	718	4	1352
AuthNoPriv	26444	678	4	1180
NoAuthNoPriv	16144	646	4	760
SNMPv1	14388	606	0	760

Energieverbrauch

Zur Bestimmung des Energieverbrauchs der gesamten 6LoWPAN SNMPv3 Steckdose wurde ein Leistungsmessgerät der Firma Voltcraft des Typs SBC-500 verwendet. Dieses besitzt eine Auflösung von 0,01W und eine Genauigkeit von ±5 Prozent. Es wurden pro Messtyp zehn

[19] Dieser Ausgleich wird jedoch nur dadurch hervorgerufen, dass in Abschnitt 4.2.2 die Konstanten des AES Algorithmus in den Programmspeicher verschoben wurden.
[20] ohne Stack und Heap Nutzung

Messungen durchgeführt und anschließend der Mittelwert dieser Messungen gebildet. Die Ergebnisse sind in Tabelle 4.2 aufgelistet.

Tabelle 4.2.: Energieverbrauch 6LoWPAN SNMPv3 Socket

Schaltzustand	**Wirkleistung**	**Scheinleistung**	**Verbrauch im Jahr**
Ein	0,53 W	2,02 VA	4,642 kWh
Aus	0,39 W	1,84 VA	3,516 kWh

Es ist zu erkennen, dass im eingeschalteten Zustand die Leistung ansteigt. Die Ursache für diesen Leistungsanstieg ist schnell gefunden wenn Tabelle 4.3 betrachtet wird. Hier wurde die Leistung der Zigbit Platine gemessen.

Tabelle 4.3.: Energieverbrauch Zigbit Platine

Schaltzustand	**Strom**	**Spannung**	**Wirkleistung**
Ein	68,7 mA	5,01 V	0,344 W
Aus	48,5 mA	5,01 V	0,243 W

Durch das Einschalten der Ausgangspins für den Optokoppler und die Leuchtdiode zur Schaltzustandsanzeige benötigt der Microcontroller mehr Strom. Dies bringt natürlich auch eine Leistungszunahme mit sich. Insgesamt ist der Leistungsverbrauch eher gering, so beträgt die Arbeit in kWh für ein Jahr, wenn davon ausgegangen wird, dass sich die Steckdose zu 80 Prozent im ausgeschalteten und 20 Prozent im eingeschalteten Zustand befindet, etwa 3,7412 kWh. Geht man von einem durchschnittlichen Strompreis von etwa 25,74 $\frac{Cent}{kWh}$ [BDEW, 2012] aus, so liegen die jährlichen Kosten für die Nutzung der 6LoWPAN SNMPv3 Steckdose unter einem Euro.

Rechenzeit und Transferzeit

Zur Bestimmung der Zeit, welche ein SNMP Request benötigt, wurde mit *Wireshark*[21] die Zeit zwischen dem Absenden eines Requests und dem Empfang der dazugehörigen Response Nachricht gemessen. Als SNMP Managementstation wurde die Software *MIB Browser Enterprise Edition 8.1*[22] der Firma *Ireasoning* verwendet. Um nun zwischen der SNMP Verarbeitungszeit auf dem Mikrocontroller und der uIPv6 Verarbeitungszeit sowie der Übertragungszeit auf der Funkstrecke unterscheiden zu können, wurde im gleichen räumlichen Abstand die Round Trip Time (RTT)[23] mithilfe des Tools `ping6`[24] gemessen. Da 6LoWPAN Netzwerke

[21] `https://www.wireshark.org/`
[22] `http://ireasoning.com/mibbrowser.shtml`
[23] In diesem Versuch beinhaltet die RTT die Übertragungszeit auf der Funkstrecke, sowie die Verarbeitungszeit von uIPv6
[24] Test Rechner: *VMWare Player 4.02* Image mit *Windows XP SP2*, *ping6* wurde innerhalb *Cygwin* ausgeführt

aufgrund der Fragmentierung der IPv6 Pakete starke Unterschiede bei unterschiedlichen Paketgrößen in der Round Trip Time haben, wurden die Pakete des Tools `ping6` exakt auf die Paketgröße des jeweiligen SNMP Pakets abgestimmt. Es wurden jeweils pro Sicherheitslevel zehn Messungen der SNMP Paket Übertragungszeit und zehn Messungen der RTT durchgeführt. Durch die Berechnung des linearen Mittelwerts und die anschließende Subtraktion der RTT von der Gesamtübertragungszeit wurden die Ergebnisse in Tabelle 4.4 berechnet. In Abbildung 4.8 ist dies zusätzlich noch grafisch dargestellt.

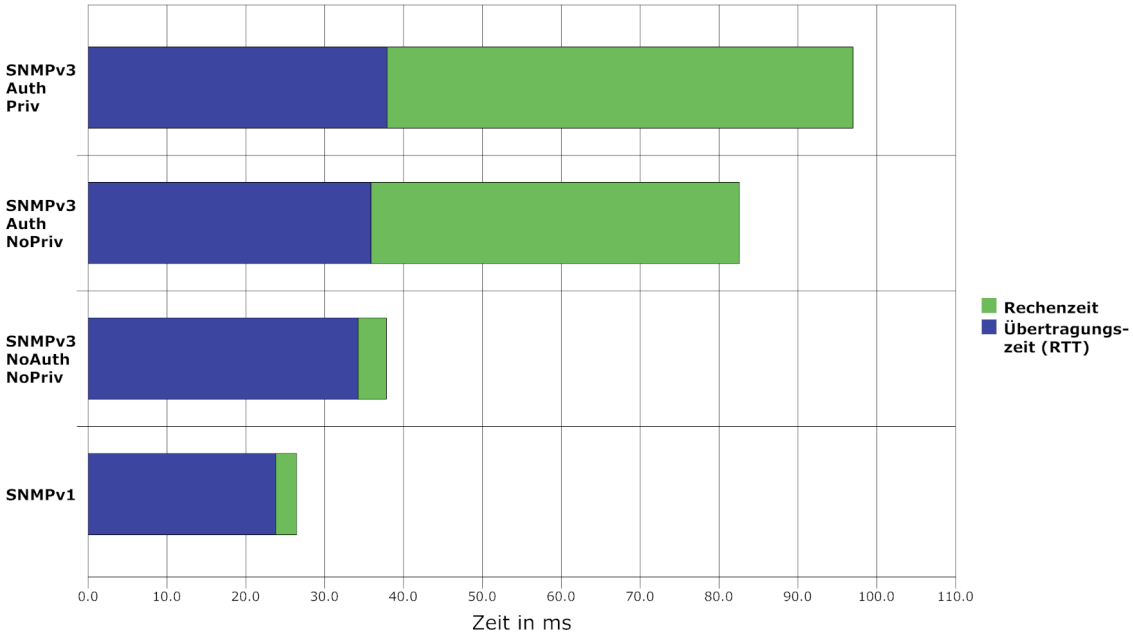


Abbildung 4.8.: Contiki SNMP Rechenzeit und Übertagungszeit der unterschiedlichen SNMP Sicherheitslevel (räumlicher Abstand 1,5m)

Tabelle 4.4.: Contiki SNMP Rechenzeit und Übertragungszeit der unterschiedlichen SNMP Sicherheitslevel (räumlicher Abstand 1,5m)

	SNMPv3 AuthPriv	**SNMPv3 AuthNoPriv**	**SNMPv3 NoAuthNo-Priv**	**SNMPv1**
Paketgröße Get-Request	194 Byte	183 Byte	171 Byte	107 Byte
Gesamtzeit	96,994 ms	82,581 ms	41,809 ms	26,43 ms
Übertragungszeit Funkstrecke	37,916 ms	35,855 ms	34,238 ms	23,8 ms
Rechenzeit	59,078 ms	46,726 ms	3,571 ms	2,63 ms

Der räumliche Abstand bei dieser Messung betrug 1,5m, als SNMP Request Nachricht wurde immer ein Get-Request mit der OID der Temperaturüberwachung gewählt.
Wie erwartet, benötigt die SNMPv3 Nachricht im Sicherheitsmodus *AuthPriv*, also bei Verschlüsselung und Authentifizierung die längste Berechnungszeit auf dem Mikrocontroller. Da auch das Gesamtpaket mit 194 Byte[25] das größte Datenvolumen darstellt, ist auch die gemessene RTT am höchsten. Wird das Sicherheitslevel *AuthNoPrivacy* verwendet, so ist schon deutlich zu sehen, dass die Arbeitszeit des Mikrocontrollers geringer wird. Wirklich interessant ist jedoch die Betrachtung bei *NoAuthNoPriv*. Hier ist deutlich zu sehen, dass nicht die PDU Verschlüsselung des Privacy Moduls die meisten Ressourcen benötigt, sondern dass die meiste Rechenzeit das Authentication Modul beansprucht. SNMPv1 unterscheidet sich in der Rechenzeit kaum von SNMPv3 im Sicherheitsmodus *NoAuthNoPriv*, lediglich die RTT ist aufgrund des höheren Datenvolumens innerhalb SNMPv3 etwas erhöht.

Belastungstest

Um herauszufinden, wie performant die Implementierung unter Belastung ist, wird in diesem Abschnitt die maximal mögliche Paketrate[26], für die jeweilige Sicherheitsoption, pro Sekunde, ermittelt. Hierzu werden Get Request Nachrichten im pcap Format[27] mit Wireshark aufgezeichnet und anschließend mit dem Tool Scapy[28] im minimal möglichen Zeitabstand zwischen den Requests über einen Zeitraum von mindestens 60 Sekunden versendet. Die Schwierigkeit hierbei ist es, den richtigen Zeitabstand zwischen den Paketen zu finden, um den maximalen Datendurchsatz zu erreichen. Es wurde deshalb der Paketabstand für die jeweilige Sicherheitsoption verwendet, durch die der maximale Paketverlust zwischen versendetem Request und empfangener Response fünf Prozent nicht übersteigt. Abbildung 4.9 und Tabelle 4.5 zeigen die ermittelten Paketdatenraten, sowie die daraus berechnete Zeit die effektiv zwischen dem Versenden der Request Nachricht und dem Erhalt der Response Nachricht benötigt wird.

[25] gemessen wurde hier immer der gesamte Ethernet II Frame
[26] vom Senden des Get Requests, bis zum Empfangen der Response
[27] `https://www.winpcap.org/`
[28] `http://www.secdev.org/projects/scapy/`

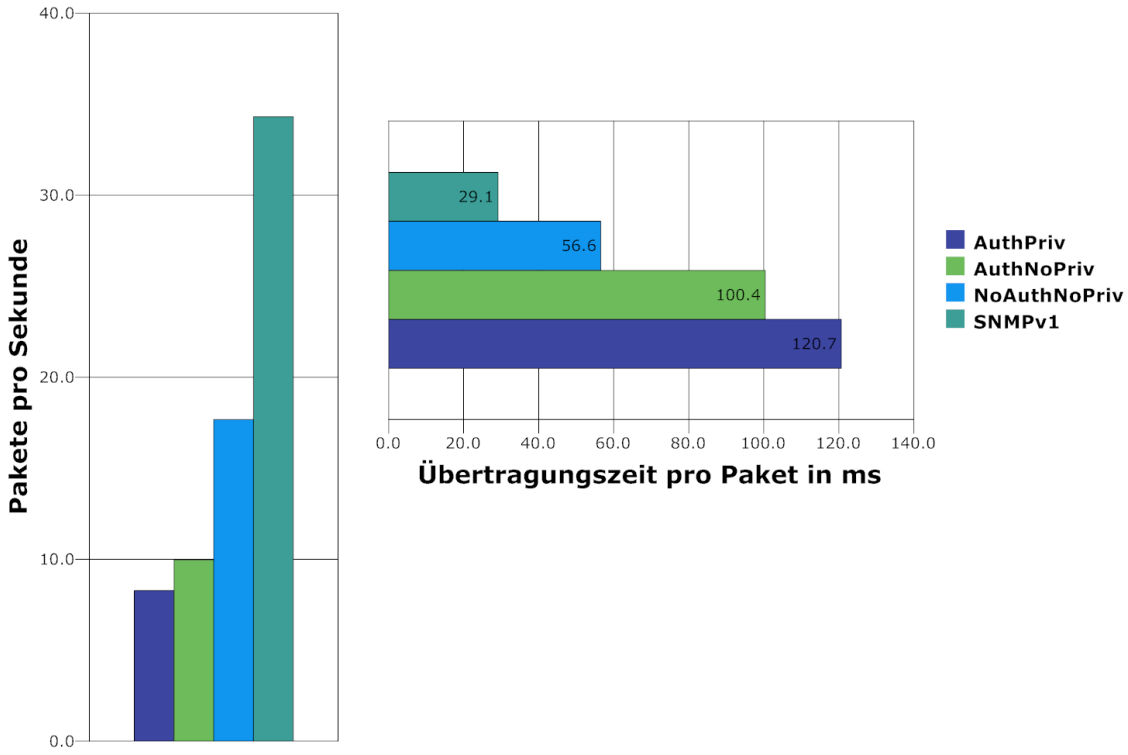


Abbildung 4.9.: Contiki SNMP unter Belastung, maximale Paketrate pro Sekunde und effektive Übertragungsdauer im Stresstest (räumlicher Abstand 1,5m)

Tabelle 4.5.: Contiki SNMP unter Belastung, maximale Paketrate pro Sekunde und effektive Übertragungsdauer im Stresstest (räumlicher Abstand 1,5m)

	SNMPv3 AuthPriv	**SNMPv3 AuthNoPriv**	**SNMPv3 NoAuthNo-Priv**	**SNMPv1**
Anzahl gesendeter Get-Requests	749	705	1305	2136
Anzahl empfangener Responses	722	670	1242	2118
gemessener Zeitraum in s	87,13747	67,2522	70,2574	61,7147
Paketrate in Pkt/s	8,28576	9,9625	17,6779	34,3192
effektive Zeit zwischen Request und Response in ms	120,689	100,376	56,568	29,138

Wie zu erwarten war, ist SNMPv1 die beste Möglichkeit um viele Daten auf dem Agenten in kürzester Zeit abzufragen. Interessant ist jedoch, dass im Vergleich zu SNMPv3 mit der Sicherheitsoption *NoAuthNoPriv* die maximal mögliche Paketdatenrate fast doppelt so hoch ist. Bei der vorherigen Messung in Abbildung 4.8 konnte dies ebenfalls bestätigt werden, dort beträgt die Gesamtübertragungszeit von SNMPv1 etwa 56 Prozent von SNMPv3 bei *NoAuthNoPriv*. Die Erklärung hierfür ist schnell gefunden, wenn die Wireshark Paketmitschnitte im Anhang unter A.10 betrachtet werden. Wird Abbildung A.8 mit Abbildung A.9 verglichen, so sieht man sofort, dass für das SNMPv1 Paket nur ein 802.15.4 Frame benötigt wird, während für das SNMPv3 NoAuthNoPriv zwei 802.15.4 Frames gebraucht werden. Das dies zu einer längeren Übertragungszeit führt, ist selbstverständlich. Für die anderen beiden SNMPv3 Sicherheitsoptionen gelangt man zu den gleichen Erkenntnissen wie in der vorherigen Messung. Die größte Zeiteinbuße verursacht nicht die PDU Verschlüsselung sondern das Authentifizierungsmodul.
Allgemein ist auch gut zu erkennen, dass die ermittelten Werte aus der vorherigen Messung unter Belastung nicht eingehalten werden können, so werden im Schnitt unter Belastung bei SNMPv1 ca. zehn Prozent zusätzliche Übertragungszeit benötigt. Bei SNMPv3 liegt dieser Wert sogar zwischen 22 und 35 Prozent. Dies ist zurückzuführen auf die zuvor besprochenen zusätzlichen 802.15.4 Frames, da durch die 6LoWPAN Fragmentierung zusätzlicher Overhead und zusätzliche Fehlerquellen hinzukommen (welche sich unter Belastung stärker auswirken) wird insgesamt mehr Übertragungszeit im Belastungsfall benötigt.

4.3.2. Funktionstests

In diesem Abschnitt werden die Funktionen der 6LoWPAN SNMPv3 Steckdose überprüft, wobei der Schwerpunkt des Funktionstests auf die Sicherheit ausgelegt ist.
Zuerst wurden die allgemeinen Funktionen der Steckdose überprüft. Alle MIB Objekte aus A.8 wurden je nach Schreib- bzw. Lesezugriff mit Get- bzw. Set Befehlen auf ihre jeweilige Funktion überprüft. Hierzu kam, wie im vorherigen Abschnitt, das Programm *MIB Browser*

Enterprise Edition[29] der Firma Ireasoning in der Version 8.1 zum Einsatz. Im weiteren Verlauf wurde der Schwerpunkt auf die Überprüfung der Sicherheit durch SNMPv3 gelegt. In Abbildung 4.10 ist ein Paketmitschnitt einer SNMPv3 Get Response zu sehen.

Abbildung 4.10.: Wireshark Paketmitschnitt eines SNMPv3 Pakets mit Sicherheitslevel AuthPriv

Wie gut zu erkennen ist, wurde durch das Privacy Modul der gesamte PDU verschlüsselt, sodass es nicht möglich ist Informationen über die transportierten Variable Bindings zu erhalten. Zur Entschlüsselung wurde dem Empfänger die sogenannte "Salt Value" im Feld `msgPrivacyParameters` hinterlegt. Im Nachrichtenfeld `msgFlags` ist zu sehen, dass die Nachricht durch das Authentication Modul und das Privacy Modul bearbeitet wurde, es besitzt also das Sicherheitslevel *AuthPriv*. Im Feld `msgSecurityModel` ist das verwendete Sicherheitsmodell eingetragen, in diesem Beispiel steht dort "3", also User Based Security Model. Weitere Sicherheitsmerkmale, die im Klartext übertragen werden, sind der `msgUserName` sowie die Felder `msgAuthoritativeEngineTime` und `msgAuthoritativeEngineBoots`. Diese beiden letzten Felder, sowie das Feld `msgAuthoritativeEngineID` werden dem Manager durch den SNMP Discovery Vorgang übermittelt. Für das Authentication Modul sind im Feld `msgAuthenticationParameters` die Parameter zur Überprüfung der Nachrichtenintegrität gespeichert. All diese Felder und ihre Funktionen wurden in den Abschnitten 2.3.8 und 3.5.4 detailliert erläutert. Im Anhang unter A.11 wurden alle diese SNMPv3 sicherheitsrelevanten Felder und ihre dazugehörigen Funktionen mittels eines festgelegten Testplans überprüft. Die Ergebnisse sind ebenfalls im Anhang unter A.11 zu finden.
Durch die Durchführung des Testplans konnte festgestellt werden, dass alle Sicherheitsvorkehrungen, welche in RFC 3414 [Blumenthal and Wijnen, 2002] definiert sind eingehalten wurden. Jedoch sind nicht alle Funktionen komplett RFC konform umgesetzt. Aus Gründen der beschränkten Ressourcen wurde teilweise auf die Versendung eines Reports und auf das

[29] `http://ireasoning.com/mibbrowser.shtml`

Speichern der Anzahl des Auftretens in dazugehörigen *Countern* verzichtet. Dies ist der Fall bei Test Nr.3 - *Get Request mit falschem USM Privacy Passwort*, die OID sowie der Counter für *usmStatsDecryptionErrors* sind auf dem Agenten nicht vorhanden. Pakete die normalerweise diesen Report auslösen würden, werden einfach verworfen, ohne dass ein Report an den Absender gesendet wird. Außerdem konnte in Test Nr 5 *Get Request mit veränderten Authentication Parameter* festgestellt werden, dass obwohl die OID und der Zähler für *usmStatsWrongDigests* vorhanden ist, kein Report versendet wird und auch der Counter nicht inkrementiert wird. Die Nachricht wird jedoch verworfen und somit kann dieses Ergebnis aus sicherheitstechnischer Sicht als bestanden angesehen werden. Insgesamt wurden alle Tests sicherheitstechnisch bestanden.

4.4. Fazit

Dieses Kapitel begann mit der Vorstellung der *6LoWPAN SNMPv3 Steckdose*, welche den praktischen Teil dieser Arbeit darstellt. Es wurde der Aufbau und die verwendete Hardware besprochen. Im darauf folgenden Abschnitt wurden die notwendigen Änderungen des originalen *Contiki SNMP* Quellcodes beschrieben. Es wurden die Codeänderungen zur Portierung auf die neuste Contiki Version und das AVR Zigbit Modul erklärt. Des Weiteren wurden die notwendigen Anpassungen des User Based Security Models detailliert dargelegt. Im weiteren Verlauf wurden die implementierten Managed Objects, die zur Steuerung der Steckdose sowie zur Temperatur- und Funkempfangsstärkeüberwachung notwendig waren, anhand des Quellcodes erklärt.
Im letzten Teil dieses Kapitels wurde die Entwicklung ausführlich untersucht und getestet. Es wurden zuerst Leistungstests, wie die Ermittlung des Speicherverbrauchs, Energieverbrauchs und die Messung der Transfer- bzw. Rechenzeiten, durchgeführt und besprochen. Anschließend wurden Funktionstests, anhand des Testplans in A.11, zur Überprüfung der Sicherheit der Entwicklung, durchgeführt und im Anschluss bewertet.

5. Zusammenfassung

Diese Arbeit beschäftigte sich mit dem *Simple Network Management Protokoll* der dritten Generation und deren Nutzung auf ressourcenbeschränkten Geräten, sogenannten *Constrained Devices*. Um den Leser langsam an das Thema heranzuführen, wurde in Kapitel 2 mit den Grundlagen zu dieser Thematik begonnen. Dort wurden zuerst die unterliegenden Schichten besprochen. Abschnitt 2.1 begann mit den beiden untersten Schichten, dem Physical Layer und dem Media Acces Control Layer. Hierfür wurde eine Übersicht über das Protokoll *802.15.4* geboten. Es wurde gezeigt, dass wenn als überliegende Schicht (Network Layer) das *Internet Protokoll Version 6* zum Einsatz kommen soll, aufgrund der sehr geringen Nutzdatenmenge der *802.15.4* Rahmen, eine Adaptionsschicht notwendig ist. Diese Adaptionsschicht wurde in Abschnitt 2.2 mit *IPv6 over Low power Wireless Personal Area Network (6LoWPAN)* vorgestellt. Es wurden die Eckdaten dieser Technologie kurz zusammengefasst und die Möglichkeiten, die durch Fragmentierung und Header Komprimierung zur Verfügung stehen, erläutert. Im letzten Abschnitt des Grundlagen Kapitels wurde dann das *Simple Network Management Protokoll* vorgestellt, es befindet sich im Application Layer und wird durch das *IPv6 Protokoll* und das darüberliegende *User Datagram Protokoll* mit der *6LoWPAN Adaptionsschicht* verbunden. Es wurde der baumartige Aufbau der *Management Information Base*, sowie die Adressierung über *Objekt Identifier* besprochen. Als Schwerpunkt dieses Kapitels wurde dann die dritte Version dieses Protokolls untersucht. Es konnte gezeigt werden, dass die Simplizität der ersten beiden Versionen in der dritten Version einem etwas komplexeren Aufbau weichen musste. Es wurde jedoch auch gezeigt, dass die enormen Sicherheitslücken eben dieser ersten beiden Versionen in der dritten Version nun kein Thema mehr sind. Durch das *User Based Access Model* erfüllt *SNMPv3* alle Anforderungen, die zum derzeitigen Zeitpunkt an ein Protokoll aus sicherheitstechnischer Sicht gestellt werden.
Es folgte das Kapitel 3 worin das Betriebssystem *Contiki OS* vorgestellt wurde. Es konnte gezeigt werden, dass sich dieses Betriebssystem für *8 Bit Mikrocontroller* durch seine *IPv6* und *6LoWPAN* Implementierung, welche in dem *mikro IP Stack (uIP)* enthalten ist, hervorragend für die angestrebte Entwicklung einer über SNMPv3 steuerbaren *6LoWPAN Funksteckdose* eignet. Die folgenden Abschnitte dieses Kapitels besprachen den Aufbau von *Contiki OS* detailliert und beschrieben die ressourcenschonenden threadähnlichen Contiki *Protothreads* sowie den Aufbau des leichtgewichtigen *uIP*, des ersten IP Stacks für 8 Bit Mikroprozessoren. Der letzte Abschnitt des Kapitels 3 beschrieb eines der wichtigsten Bauteile dieser Arbeit. Das von der Jacobs Universität entwickelte *Contiki SNMP*. Dieses wurde detailgenau anhand des Quellcodes und durch das Wissen aus dem Grundlagenkapitel untersucht und beschrieben, wodurch gezeigt werden konnte, dass die Implementierung für das geplante Vorhaben sehr gut geeignet ist.
Das letzte Kapitel, das Kapitel 4, zeigte das Resultat dieser Arbeit, die mit dem Wissen aus den vorherigen Kapiteln entwickelte *6LoWPAN SNMPv3 Funksteckdose*. Es wurden zuerst die einzelnen Hardware und Software Komponenten vorgestellt und besprochen. Im weiteren Verlauf wurden die Portierungs- sowie sicherheitstechnisch notwendigen Modifikationen des original *Contiki SNMP* im Detail erklärt. Schließlich endete das Kapitel mit den aus-

führlichen Tests der angefertigten Entwicklung. Dort wurden Speicherverbrauch, Energieverbrauch sowie Transfer- und Rechenzeiten bestimmt. Außerdem wurden mithilfe des SNMPv3 Sicherheitstestplans in A.11 alle sicherheitsrelevanten Funktionen der *6LoWPAN SNMPv3 Steckdose* überprüft und ausgewertet.

Resümierend lässt sich sagen, dass *Contiki SNMP* sich prinzipiell hervorragend zur Steuerung von *Smart Objects* nutzen lässt. Zu beachten ist jedoch, dass trotz der durchaus speichereffizienten Programmierung große Ressourcen des Mikrocontrollers beansprucht werden, wenn das sichere *SNMPv3* mit der Sicherheitsoption *AuthPriv* eingesetzt wird, siehe Kapitel 4.3.1. Sehr viel speichereffizienter ist natürlich die erste Version des *Simple Network Management Protokolls*. Aufgrund der mangelhaften bzw. nicht vorhandenen Sicherheit sollte diese Version jedoch nicht eingesetzt werden. Wird *Contiki SNMP*, wie in dieser Arbeit als Hauptprogramm zur Steuerung genutzt, so ist die Implementierung hervorragend geeignet. Ist das Einsatzgebiet jedoch die Verwaltung eines Knotens, worauf ein anderes Hauptprogramm mit ebenfalls speicherintensiven Eigenschaften eingesetzt wird, so ist der Einsatz von *Contiki SNMP*, zumindest wenn *SNMPv3* eingesetzt wird, nicht effektiv.

Zukünftige Arbeiten sollten sich mit der Implementierung von *SNMPv3 Traps* in das *Contiki SNMP* beschäftigen, dies würde die Möglichkeit schaffen, durch Nutzung der bereits implementierten Temperaturüberwachung, ab einem bestimmten Schwellwert Warnmeldungen an den *SNMP Manager* zu senden. Auch eine Warnmeldung beim Erreichen eines niedrigen Funkempfangswerts wäre denkbar. Des Weiteren wäre die Implementierung von *Over-The-Air Updates* eine äußerst sinnvolle Funktion. Durch die in [Dunkels et al., 2006a] beschriebenen Verfahren wäre es möglich während der Laufzeit *MIB Objekte* hinzuzufügen bzw. zu löschen oder gar ein komplettes *Contiki SNMP* Versionsupdate durchzuführen. Denkbar wäre die Implementierung des in [Dunkels et al., 2004] vorgestellten *Over-the-air programming*, bei dem hunderte Sensoren, steuerbar durch *Contiki SNMP*, über einen *Konzentrator Knoten* ihr Software Update via Broadcasts erhalten. Grundsätzlich sind die Einsatzmöglichkeiten unbegrenzt, theoretisch könnten durch die Nutzung der *Contiki SNMP* Implementierung alle denkbaren Geräte gesteuert bzw. verwaltet werden. Und dies praktisch von jedem Ort der Welt, welcher eine Anbindung zum *globalen Internet* besitzt.

Abkürzungsverzeichnis

6LoWPAN	IPv6 over Low power Wireless Personal Area Network
ADC	Analog Digital Converter
AE	Authoritative Engine (SNMP Timeliness Modul)
AES	Advanced Encryption Standard
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
CBC	Cipher Block Chaining
CBC-DES	Cipher Block Chaining Data Encryption Standard
CFB	Cipher Feedback Mode
CSMA/CA	Carrier Sense Multiple Access/Collision Avoidance
GNU	General Public License
HEMS	High-Level Entity Management Systems
HMAC	Hased-Based Message Authentication Code
IAB	Internet Architecture Board
IANA	Internet Assigned Numbers Authority
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protokoll
IPv6	Internet Protokoll Version 6
IPX	Internetwork Packet Exchange
IV	Initialisierungsvektor
MAC	Medium Access Control Layer
MD5	Message Digest 5
MIB	Management Information Base
NAE	Nonauthoritative Engine (SNMP Timeliness Modul)
NIST	National Institute of Standards and Technology
NTC	Negative Temperature Coefficient Thermistor
OID	Object Identifier
PAN	Personal Area Network
PDU	Protocol Data Unit
PHY	Physical Layer
RFC	Request for Comment
RTT	Round Trip Time
SHA	Secure Hash Algorithm 1
SMI	Structure of Management Information
SNMP	Simple Network Management Protokoll
UDP	User Datagramm Protocol
USM	User Based Security Model
VACM	View-Based Access Control Model

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

H. Ashtawy, T. Brown, X. Wang, and Y. Zhang. Take a hike - a trek through the contiki operating system. Technical report, Department of Computer Science and Engineering Michigan State University, 2012.

BDEW. Bundesverband der energie- und wasserwirtschaft e.v. -strompreisanalyse mai 2012, haushalte und industrie. Online, pdf, May 2012. URL http://bdew.de/internet.nsf/id/0E5D39E2E798737FC1257A09002D8C9C/$file/120525%20BDEW-Strompreisanalyse%202012%20Chartsatz%20gesamt.pdf. Online Accessed 27.07.2012.

B. H. B. BHT. Installation vom instant-contiki und kompilieren eines ersten programms. Online, May 2012. URL https://wiki.ipv6lab.beuth-hochschule.de/contiki/howto_compile. Online Accessed 08.08.2012.

R. Bless, S. Mink, E.-O. Blaß, M. Conrad, H.-J. Hof, K. Kutzner, and M. Schöller. *Sichere Netzwerkkommunikation: Grundlagen, Protokolle und Architekturen (X.systems.press) (German Edition)*. Springer, 2005. ISBN 3540218459.

U. Blumenthal and B. Wijnen. User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3). RFC 3414 (Standard), Dec. 2002. URL http://www.ietf.org/rfc/rfc3414.txt. Updated by RFC 5590.

U. Blumenthal, F. Maino, and K. McCloghrie. The Advanced Encryption Standard (AES) Cipher Algorithm in the SNMP User-based Security Model. RFC 3826 (Proposed Standard), June 2004. URL http://www.ietf.org/rfc/rfc3826.txt.

S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119 (Best Current Practice), Mar. 1997. URL http://www.ietf.org/rfc/rfc2119.txt.

D. Bruey. Snmpsnmp: Simple? network management protocol. Technical report, Rane Corporation, 2005.

J. Case, M. Fedor, M. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990. URL http://www.ietf.org/rfc/rfc1157.txt.

J. Case, D. Harrington, R. Presuhn, and B. Wijnen. Message Processing and Dispatching for the Simple Network Management Protocol (SNMP). RFC 3412 (Standard), Dec. 2002a. URL http://www.ietf.org/rfc/rfc3412.txt. Updated by RFC 5590.

J. Case, R. Mundy, D. Partain, and B. Stewart. Introduction and Applicability Statements for Internet-Standard Management Framework. RFC 3410 (Informational), Dec. 2002b. URL http://www.ietf.org/rfc/rfc3410.txt.

J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. RFC 1067: Simple network management protocol, Aug. 1988. URL `ftp://ftp.internic.net/rfc/rfc1067.txt`, `ftp://ftp.internic.net/rfc/rfc1098.txt`, `ftp://ftp.math.utah.edu/pub/rfc/rfc1067.txt`, `ftp://ftp.math.utah.edu/pub/rfc/rfc1098.txt`. Obsoleted by RFC1098 Status: UNKNOWN.

V. G. Cerf. RFC 1052: IAB recommendations for the development of Internet network management standards, Apr. 1988. URL `ftp://ftp.internic.net/rfc/rfc1052.txt`, `ftp://ftp.math.utah.edu/pub/rfc/rfc1052.txt`. Status: UNKNOWN.

M. Crawford. Transmission of IPv6 Packets over Ethernet Networks. RFC 2464 (Proposed Standard), Dec. 1998. URL `http://www.ietf.org/rfc/rfc2464.txt`. Updated by RFC 6085.

S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), Dec. 1998. URL `http://www.ietf.org/rfc/rfc2460.txt`. Updated by RFCs 5095, 5722, 5871, 6437, 6564.

O. Dubuisson. *ASN.1 Communication Between Heterogeneous Systems*. Morgan Kaufmann, 2000. ISBN 0126333610.

T. Duff. Duffs device. `http://doc.cat-v.org/bell_labs/duffs_device`, 1988. Online; accessed 28-May-2012.

A. Dunkels. Full tcp/ip for 8 bit architectures. In *In Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003))*, San Francisco, USA, May 2003.

A. Dunkels. Towards tcp/ip for wirelesssensor networks. Master's thesis, Swedish Institute of Computer Science Stockholm, Sweden, 2005.

A. Dunkels. Contiki wiki - multithreading library. `https://www.sics.se/contiki/wiki/index.php/Multithreading/`, 2011a. Online; accessed 07-June-2012;.

A. Dunkels. Contiki wiki - processes. `https://www.sics.se/contiki/wiki/index.php/Processes/`, 2011b. Online; accessed 07-June-2012;.

A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. Technical report, In Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I), Tampa, Florida, USA, November 2004, 2004.

A. Dunkels, O. Schmidt, and T. Voigt. Using protothreads for sensor node programming. In *In Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, Nov. 2006a. URL `http://www.sics.se/~adam/dunkels06runtime.pdf`.

A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *In Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006b.

M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels. Making sensor networks ipv6 ready. In *Proceedings of the Sixth ACM Conference on Networked Embedded Sensor Systems (ACM SenSys 2008)*, North Carolina, USA, November 2008a.

M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels. Making sensor networks ipv6 ready. In *In Proceedings of the Sixth ACM Conference on Networked Embedded Sensor Systems (ACM SenSys 2008)*, Raleigh, North Carolina, USA, November 2008b.

E. F. F. EFF. *Cracking Des: Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly Media, 1998. ISBN 1565925203.

W. Ertel. *Angewandte Kryptographie.* Hanser Fachbuchverlag, 2007. ISBN 344641195X.

J. A. Gutierrez, L. Winkel, E. H. Callaway-Jr., and R. L. Barrett-Jr. *Low-Rate Wireless Personal Area Networks: Enabling Wireless Sensors With IEEE 802.15.4 (Ieee Standards Wireless Networks).* John Wiley and Sons; Auflage: 3, 2011. ISBN 073816285X.

D. Harrington and W. Hardaker. Transport Security Model for the Simple Network Management Protocol (SNMP). RFC 5591 (Draft Standard), June 2009. URL `http://www.ietf.org/rfc/rfc5591.txt`.

D. Harrington, R. Presuhn, and B. Wijnen. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. RFC 3411 (Standard), Dec. 2002. URL `http://www.ietf.org/rfc/rfc3411.txt`. Updated by RFCs 5343, 5590.

R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), Feb. 2006. URL `http://www.ietf.org/rfc/rfc4291.txt`. Updated by RFCs 5952, 6052.

J. Hui. Internet-Draft: Compression format for ipv6 datagrams in 6lowpan networks draft-hui-6lowpan-hc-00, march 2008a. URL `http://tools.ietf.org/html/draft-hui-6lowpan-hc-00`. Expires: September 11, 2008.

J. Hui. Internet-Draft: Compression format for ipv6 datagrams in 6lowpan networks draft-hui-6lowpan-hc-01, july 2008b. URL `http://tools.ietf.org/html/draft-hui-6lowpan-hc-01`. Expires: January 29, 2009.

J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282 (Proposed Standard), Sept. 2011. URL `http://www.ietf.org/rfc/rfc6282.txt`.

IANA. Iana ipv4 address space registry. Online, April 2012. URL `https://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml`. Online Accessed 31.07.2012.

IEEE. *IEEE Std 802.15.4^{TM}-2003 - Wireless Medium Access Control (Mac) and Physical Layer (Phy) Specifications for Low-Rate Wireless Personal Area Networks (Lr-Wpans) (Ieee Standard for Information Technology 802.15.4)*. Institute of Electrical and Electronics Engineers, 2003. ISBN 0738136867.

IEEE. *IEEE Std 802.15.4^{TM}-2006 (Revision of IEEE Std 802.15.4-2003) IEEE standard for information technology telecommunications and information exchange between systems–local and metropolitan area networks–specific requirements*. Institute of Electrical and Electronics Engineers, New York, NY, 2006. ISBN 0738149969.

IEEE. *IEEE Std 802.15.4a^{TM}-2007 (Amendment to IEEE Std 802.15.4^{TM}-2006) IEEE standard for information technology telecommunications and information exchange between systems–local and metropolitan area networks–specific requirements*. Institute of Electrical and Electronics Engineers, New York, NY, 2007. ISBN 0738155845.

IEEE. *Standard for information technology portable operating system interface (POSIX) : base specifications*. Institute of Electrical and Electronics Engineers, New York, 2008. ISBN 9780738157993.

ITU. *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) (ITU-T Recommendation X.690)*. International Telecommunications Union, 2002.

D. Knuth. *The art of computer programming*. Addison-Wesley, 1997. ISBN 0201896834.

H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), Feb. 1997. URL `http://www.ietf.org/rfc/rfc2104.txt`. Updated by RFC 6151.

S. Kumar, C. Paar, J. Pelzl1, G. Pfeiffer, A. Rupp, and M. Schimmler. How to break des for bc 8,980 euro. Technical report, Ruhr University Bochum and Christian-Albrechts-University of Kiel, 2006.

S. Kuryla. Implementation and evaluation of the simple network management protocol over ieee 802.15.4 radios under the contiki operating system. Master's thesis, Jacobs University Bremen, 2010.

S. Kuryla and J. Schönwälder. Evaluation of the resource requirements of snmp agents on constrained devices. In I. Chrisment, A. Couch, R. Badonnel, and M. Waldburger, editors, *Managing the Dynamics of Networks and Services*, Lecture Notes in Computer Science, pages 100–111. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-21483-7.

N. Kushalnagar, G. Montenegro, and C. Schumacher. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. RFC 4919 (Informational), Aug. 2007. URL `http://www.ietf.org/rfc/rfc4919.txt`.

J. Larmouth. *ASN.1 Complete*. Morgan Kaufmann, 1999. ISBN 0122334353.

D. Levi, P. Meyer, and B. Stewart. Simple Network Management Protocol (SNMP) Applications. RFC 3413 (Standard), Dec. 2002. URL `http://www.ietf.org/rfc/rfc3413.txt`.

J. Loughney. IPv6 Node Requirements. RFC 4294 (Informational), April 2006. URL http://www.ietf.org/rfc/rfc4294.txt. Updated by RFC 5095.

P. Mandl. *Grundkurs Betriebssysteme (German Edition)*. Vieweg+Teubner Verlag, 2008. ISBN 3834803928.

K. McCloghrie and M. Rose. Management Information Base for Network Management of TCP/IP-based internets:MIB-II. RFC 1213 (Standard), Mar. 1991. URL http://www.ietf.org/rfc/rfc1213.txt. Updated by RFCs 2011, 2012, 2013.

K. McCloghrie, D. Perkins, and J. Schoenwaelder. Structure of Management Information Version 2 (SMIv2). RFC 2578 (Standard), Apr. 1999a. URL http://www.ietf.org/rfc/rfc2578.txt.

K. McCloghrie, D. Perkins, and J. Schoenwaelder. Textual Conventions for SMIv2. RFC 2579 (Standard), Apr. 1999b. URL http://www.ietf.org/rfc/rfc2579.txt.

D. McNett. Formal press release - us government's encryption standard broken in less than a day. Technical report, distributed.net, 1999.

MMG. Miniwatts marketing group - world internet usage and population statistics. Online, July 2012. URL http://www.internetworldstats.com/stats.htm. Online Accessed 31.07.2012.

G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard), September 2007. URL http://www.ietf.org/rfc/rfc4944.txt.

S. of Cable Telecommunications Engineers. ANSI/SCTE 52 2008 - Data Encryption Standard – Cipher Block Chaining Packet Encryption Specification. Technical report, AMERICAN NATIONAL STANDARD, 2008. URL http://www.scte.org/documents/pdf/Standards/ANSI_SCTE%2052%202008.pdf.

N. I. of Standards and Technology. Federal information processing standards publication (FIPS 197). Advanced Encryption Standard (AES). Technical report, DEPARTMENT OF COMMERCE, 2001a. URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

N. I. of Standards and Technology. Recommendation for block cipher modes of operation methods and techniques nist special publication 800-38a. Technical report, DEPARTMENT OF COMMERCE, 2001b. URL http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

N. I. of Standards and Technology. Federal information processing standards publication (FIPS 198). The Keyed-Hash Message Authentication Code (HMAC). Technical report, DEPARTMENT OF COMMERCE, 2002a. URL http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf.

N. I. of Standards and Technology. FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2. Technical report, DEPARTMENT OF COMMERCE, Aug. 2002b. URL http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

M. Prasad. The adc of the avr. http://maxembedded.wordpress.com/2011/06/20/the-adc-of-the-avr/, 2011. Online; accessed 18-July-2012.

R. Presuhn. Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP). RFC 3416 (Standard), Dec. 2002a. URL http://www.ietf.org/rfc/rfc3416.txt.

R. Presuhn. Transport Mappings for the Simple Network Management Protocol (SNMP). RFC 3417 (Standard), Dec. 2002b. URL http://www.ietf.org/rfc/rfc3417.txt. Updated by RFCs 4789, 5590.

R. Presuhn. Management Information Base (MIB) for the Simple Network Management Protocol (SNMP). RFC 3418 (Standard), Dec. 2002c. URL http://www.ietf.org/rfc/rfc3418.txt.

C. Priplata and C. Stahlke. Wie sicher ist kryptografie? knackige maschinen. *iX*, 2006.

R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), Apr. 1992. URL http://www.ietf.org/rfc/rfc1321.txt. Updated by RFC 6151.

M. Rose and K. McCloghrie. Structure and identification of management information for TCP/IP-based internets. RFC 1155 (Standard), May 1990. URL http://www.ietf.org/rfc/rfc1155.txt.

T. Schwenkler. *Sicheres Netzwerkmanagement: Konzepte, Protokolle, Tools (X.systems.press) (German Edition)*. Springer, 2005. ISBN 3540236120.

SciEngines-GmbH. Break des in less than a single day. Online, august 2012. URL http://www.sciengines.com/company/news-a-events/74-des-in-1-day.html. Online Accessed 01.08.2012.

Z. Shelby and C. Bormann. *6LoWPAN: The Wireless Embedded Internet (Wiley Series on Communications Networking & Distributed Systems)*. Wiley, 2009. ISBN 0470747994.

SICS. Contiki 2.5 doxygen documentation. http://dak664.github.com/contiki-doxygen/, 2012. Online; accessed 04-June-2012; Kann auch aus Contiki Quellcode generiert werden.

S. Spitz, M. Pramateftakis, and J. Swoboda. *Kryptographie und IT-Sicherheit: Grundlagen und Anwendungen (German Edition)*. Vieweg+Teubner Verlag, 2011. ISBN 3834814873.

W. Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2 (3rd Edition)*. Addison-Wesley Professional, 1999. ISBN 0201485346.

W. Stallings. *Cryptography and Network Security: Principles and Practice (5th Edition)*. Prentice Hall, 2010. ISBN 0136097049.

S. Tatham. Coroutines in c. 2000. URL http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html.

N. Tsiftes, J. Eriksson, and A. Dunkels. Poster abstract: Low-power wireless ipv6 routing with contikirpl. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2010)*, Stockholm, Sweden, Apr. 2010. URL `http://www.sics.se/~adam/tsiftes10rpl.pdf`.

J.-P. Vasseur and A. Dunkels. *Interconnecting Smart Objects with IP: The Next Internet*. Morgan Kaufmann, 2010. ISBN 0123751659.

B. Wijnen, R. Presuhn, and K. McCloghrie. View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP). RFC 3415 (Standard), Dec. 2002. URL `http://www.ietf.org/rfc/rfc3415.txt`.

A. Anhang

A.1. Aufbau der usmUser Group laut RFC3414

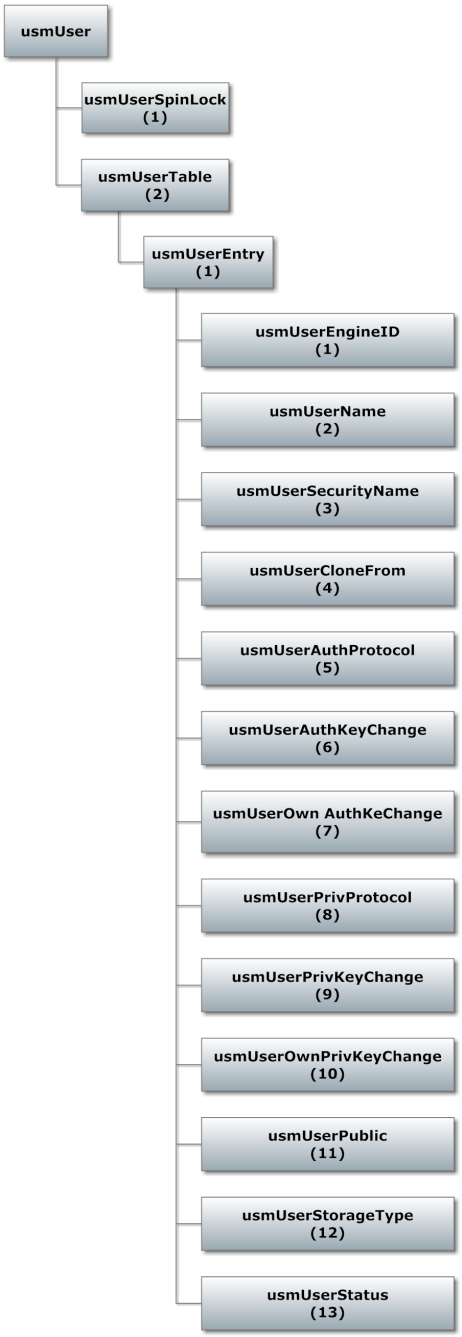


Abbildung A.1.: Aufbau der USM User Group

A.2. Konfigurationsparameter Contiki SNMP

A.2.1. Allgemeine Parameter

Tabelle A.1.: Allgemeine Parameter des Contiki SNMP Agents

Funktion	Parameter	Definition	Möglichkeiten
SNMPv1 Unterstützung aktivieren	#define ENABLE_SNMPv1	snmpd-conf.h	Wert 1 -> SNMPv1 Unterstützung aktiviert, Wert 0 SNMPv1 Unterstützung deaktiviert.
SNMPv3 Unterstützung aktivieren	#define ENABLE_SNMPv3	snmpd-conf.h	Wert 1 -> SNMPv3 Unterstützung aktiviert, Wert 0 SNMPv3 Unterstützung deaktiviert.
Festlegung des SNMP Nachrichtenbuffers	#define MAX_BUF_SIZE	snmpd-conf.h	dezimal in Byte, Standard 484
Festlegung des Zeitfensters für das Timeliness Modul	#define TIME_WINDOW	snmpd-conf.h	Standard laut RFC 150, sollte nicht verändert werden.

A.2.2. Logging und Debugging Parameter

Tabelle A.2.: Logging und Debugging Parameter des Contiki SNMP Agents

Funktion	Parameter	Definition	Möglichkeiten
Festlegung des UDP Ports für Logging Output	#define LOGGING_PORT	logging.c	kann frei gewählt werden
Festlegung des UDP Buffers für Info bzw. Debugging Nachrichten	#define BUF_LEN	logging.c	kann frei gewählt werden
Ein und Auschalten des Debug Modus (Nur für Minimal Net)	#define DEBUG	logging.h	Wert 1 -> Debug Modus aktiv, Wert 0 Debug Modus deaktiviert.
Ein- und Auschalten des Info Modus (Debug Modus für Zielgeräte die nicht MinimalNet entsprechen)	#define INFO	logging.h	Wert 1 -> Info Modus aktiv, Wert 0 Info Modus deaktiviert.

A.2.3. Sicherheits Parameter

Tabelle A.3.: Sicherheits Parameter des Contiki SNMP Agents

Funktion	**Parameter**	**Definition**	**Möglichkeiten**
Festlegung der Authoritative Engine ID	u8t msgAuthoritativeEngineID_array[]	snmpd-conf.c	Standard 13 Byte, hexadezimal
Festlegung des Benutznamens	u8t* usmUserName	snmpd-conf.c	Mit Typecast String in Anführungszeichen zuweisen, also z.B. (u8t*)"Benutzername"
Privacy Modul aktivieren	#define ENABLE_PRIVACY	snmpd-conf.h	Wert 1 -> Privacy Modul aktiviert, Wert 0 Privacy Modul deaktiviert.
Authentication Modul aktivieren	#define ENABLE_AUTH	snmpd-conf.h	Wert 1 -> Authentication Modul aktiviert, Wert 0 Authentication Modul deaktiviert.
Festlegung des Authorisierungs Schlüssels	u8t authKul[16]	keytools.c	16 Byte Länge, muss zuvor mithilfe des Keygenerators erzeugt werden
Festlegung des Privacy Schlüssels	u8t privKul[16]	keytools.c	16 Byte Länge, muss zuvor mithilfe des Keygenerators erzeugt werden
Festlegung des Community Strings	#define COMMUNITY_STRING	snmpd-conf.h	Standard ist public", kann frei gewählt werden

A.2.4. MIB Parameter

Tabelle A.4.: Parameter zur Management Information Base des Contiki SNMP Agents

Funktion	Parameter	Definition	Möglichkeiten
Festlegung der Speicherallozierung für MIB Objekte	#define MIB_SIZE	mib.h	Wert 1 -> statische Speicherallozierung, Wert 0 -> dynamische Speicherallozierung.
Aktivierung/-Deaktivierung von tabularen MIB Objekten	#define ENABLE_MIB-_TABLE	mib.h	Wert 1 -> tabulare Objekte möglich, Wert 0 -> keine tabularen Objekte möglich
Ein und Auschalten des Debug Modus (Nur für Minimal Net)	#define ENABLE-_PROGMEM	mib.h	Wert 1 -> Speicherung der MIB Objekte ins Programm Memory aktiv, Wert 0 -> Speicherung der MIB Objekte ins Programm Memory deaktiviert.

A.3. Einrichtung der Contiki Entwicklungsumgebung

Benötigte Hardware:

- AVR Raven Board (ATmega1284P und ATmega3290P sowie AT86RF230 802.15.4 Funkchip)
- Raven RZ USB Stick
- PC mit Windows/Linux oder einen Mac
- JTAGICE mkII

Benötigte Software:

- VMware Player
- Windows XP SP3 VMware Image mit folgender Software:
  - Cygwin + make
  - AVR Studio 4
  - WinAVR
- Instant Contiki VMware Image

VMWare Player Installation

Je nachdem, welches Betriebssystem genutzt wird, müssen die passenden VMware Player Installationsdateien heruntergeladen werden. Dies ist unter `http://www.vmware.com/de/products/desktop_virtualization/player/overview` möglich. Es ist jedoch notwendig sich zuvor bei VMware zu registrieren.
Nach der Installation des VMware Players wird nun zunächst das Windows XP SP3 Gastsystem installiert, hierzu wird eine Windows XP CD oder Image benötigt.
Das zweite Gastbetriebssystem wird durch das Instant Contiki VMware Image repräsentiert, dieses kann unter `http://www.contiki-os.org/p/instant-contiki.html` heruntergeladen werden.

Anpassung des Windows Gastsystems

Um auf dem Windows Gastsystem Contiki Software kompilieren zu können wird *Cygwin + make Utility* sowie *WinAVR* benötigt. Cygwin kann unter `http://cygwin.com/install.html` heruntergeladen werden. Während der Installation ist es möglich das Make Utility mit zu installieren. WinAVR kann unter `http://sourceforge.net/projects/winavr/files/WinAVR/` heruntergeladen werden.

Zum Programmupload auf den Mikrocontroller kann das *AVR Studio* genutzt werden. Es wird empfohlen AVR Studio 4 zu verwenden, da dieses die Möglichkeit bietet `.elf` Files zu nutzen, diese Funktion ist in AVR Studio 5 nicht mehr vorhanden. Unter `http://www.mikrocontroller.net/articles/AVR-Studio` ist es möglich AVR Studio 4 ohne Anmeldung herunterzuladen.

Nach Abschluss der Installation kann nun der *JTAGICE mkII* mit dem PC und anschließend mit dem Gastsystem verbunden werden. Dazu im laufenden Windows Gastsystem im Menü auf *Virtual Machine/Removeable Devices/* den Menüpunkt *"atmel jtagice mkII"* auswählen und Windows die Treiber selbst suchen und installieren lassen.

A.4. Einrichtung des RZRaven USB Sticks

Contiki Installation ab Contiki 2.5

Sind diese Schritte aus Kapitel A.3 erfolgt, so muss zuerst in das Windows XP Gastsystem gewechselt werden und anschließend von `http://sourceforge.net/projects/contiki/files/Contiki/Contiki2.x` heruntergeladen werden. Es ist nun sinnvoll den Contiki Sourcecode in das Cygwin Verzeichnis zu entpacken, also beispielsweise unter *C:\cygwin\home\Administrator\Contiki-2.x* um anschließend einfacher mit Cygwin arbeiten zu können. Nun wird Cygwin gestartet und in das Contiki Source Code Verzeichnis gewechselt. Anschließend in das Verzeichnis *\examples\ravenusbstick*. Durch den Befehl `make` beginnt die Kompilierung. Ist diese abgeschlossen, so sollte nun im derzeitigen Verzeichnis die Datei `ravenusbstick.elf` zu sehen sein. Diese Datei wird nun mithilfe des AVR Studios auf den USB Stick geladen. Zuerst muss jedoch der USB Stick mit dem Programmieradapter verbunden werden. Es wird empfohlen die Verbindung spannungslos aufzubauen, d.h. den Programmieradapter sowie den USB Stick spannungslos zu verbinden und danach zuerst den Programmieradapter einschalten und dann den USB Stick mit dem USB-Port verbinden. Die korrekte Position des JTAG Anschlusses ist in Abbildung A.2 zu sehen.

Abbildung A.2.: Anschluß des Raven USB Sticks

Im nächsten Schritt das AVR Studio starten und sämtliche PopUps schließen. Dann im Menü auf *Tools/Program AVR/ Auto Connect/* klicken. Im anschließend erscheinenden Fenster den Reiter *Main* auswählen und als Device *AT90USB1287* markieren. Mit einem Klick auf *Read Signature* wird die korrekte Verbindung überprüft und die Mikrocontroller Signatur eingelesen.
→ Wenn kein Sockel auf dem USB Stick vorhanden ist, so kann es vorkommen, dass an dieser Stelle eine Fehlermeldung auftritt. Dies kann durch justieren der Steckverbindung behoben werden.
Wurde die Signatur korrekt eingelesen, so kann die `.elf` Datei auf den Mikrocontroller geladen werden. Dazu den Reiter *Program* anklicken und im Feld *ELF Production File Format* die Datei `ravensubstick.elf` auswählen und anschließend auf *Program* klicken.

Einrichtung unter Instant Contiki

Da der USB-Stick nun als Router fungieren soll, muss dies unter Ubuntu zuerst eingestellt werden. Zuerst muss der USB Stick mit Instant Contiki verbunden werden, dazu wie bei der Installation des Programmieradapters über *Virtual Machine/Removable Devices/* den Menüpunkt *atmel jackdaw 6lowpan adaptor* auswählen. Ubuntu erkennt den USB Stick automatisch, eine weitere Treiber Installation ist normalerweise nicht nötig. Zur Überprüfung in einer Konsole `ifconfig` eingeben und überprüfen ob `USB0` als Netzwerkinterface vorhanden ist und ob die Adressen korrekt gesetzt wurden. Falls die link locale IPv6 Adresse nicht mit der in der Abbildung A.3 übereinstimmt so kann diese mit `sudo ip -6 address-add fe80::0012:13ff:fe14:1516/64 scope link dev usb0` gesetzt werden. Das SuperUser Passwort ist wie der Benutzername `user`. Nun muss noch die globale IPv6 Adresse zugewiesen werden, dies geschieht durch den Befehl `sudo ip -6 address add aaaa::1/64 dev usb0`.

Damit der USB Stick nun Router Advertisments versendet, gibt es die Software *radvd*, diese muss jedoch zuerst installiert werden, dazu: `sudo apt-get install radvd` in einer Console eingeben.

```
usb0      Link encap:Ethernet  HWaddr 02:12:13:14:15:16
          inet6 addr: fe80::12:13ff:fe14:1516/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1298  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:19 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:6354 (6.3 KB)
```

Abbildung A.3.: ifconfig USB0

Listing A.1: Parameter radvd.conf

```
1  interface usb0
2  {
3      AdvSendAdvert on;
4      AdvLinkMTU 1280;
5      AdvCurHopLimit 128;
6      AdvReachableTime 360000;
7      MinRtrAdvInterval 100;
8      MaxRtrAdvInterval 150;
9      AdvDefaultLifetime 200;
10     prefix AAAA::/64
11     {
12         AdvOnLink on;
13         AdvAutonomous on;
14         AdvPreferredLifetime 4294967295;
15         AdvValidLifetime 4294967295;
16     };
17 };
```

Zur Konfiguration muss die Datei radvd.conf unter /etc/radvd.conf erstellt werden. Dazu genügt es mit nano und superuser Rechten eine neue Datei zu erstellen: `sudo nano /etc-/radvd.conf`. Innerhalb der Datei müssen danach die Konfigurationsparameter aus Listing A.1 übernommen werden.

Ist dies geschehen, so muss nur noch das IPv6 Forwarding im SuperUser Konto aktiviert werden. Dazu zuerst `sudo su` und im Anschluss `echo 1 > /proc/sys/net/ipv6/-conf/all/forwarding`. Danach kann der SuperUser mit `exit` wieder verlassen werden. Nun kann radvd gestartet werden. Dazu den Befehl `/etc/init.d/radvd restart` in einer Konsole eingeben.

A.5. Erstellung eines MIB Objekts für Contiki SNMP

Um ein MIB Objekt auf dem Agent zu erstellen, kann als Vorlage die Datei `mib-init.c` verwendet werden, diese befindet sich, insofern die Installationsanleitung aus Abschnitt A.6 durchgeführt wurde unter *\cygwin\home\Administrator\C2X-SNMP\contiki-2.x\apps\snmpd*. Die Datei `mib-init.c` enthält bereits einige Standard MIB Objekte wie System Descrip-

Listing A.2: OID

```
1 static u8t ber_oid_beuth_int[] PROGMEM      =
2         {0x2b, 0x06, 0x01, 0x04, 0x01, 0x81,
3         0xac, 0x5d, 0x64, 0x01};
```

Listing A.3: OID in ptr_t Struktur

```
1 static ptr_t oid_beuth_int PROGMEM=
2         {ber_oid_beuth_int, 10};
```

tion, System Time usw... Diese können entweder gelöscht werden oder auch weiterhin verwendet werden.

OID erstellen

Um nun ein neues Objekt hinzuzufügen muss zuerst eine gültige OID erstellt werden. Zur Erzeugung einer eigenen OID kann der Quelltext unter \cygwin \home \Administrator \C2X-SNMP \Source \ber-encoder verwendet werden. In diesem Beispiel soll ein Pin des Raven Boards angesteuert werden und die OID der Beuth Hochschule *1.3.6.1.4.1.22109* verwendet werden. (`http://www.iana.org/assignments/enterprise-numbers/` unter TFH Berlin) Hierzu wird nun die OID mit mithilfe des BER Encoders codiert, verwendet wird als Unterstruktur 100.1, also 1.3.6.1.4.1.22109.100.1. Als Ergebnis lierfert der BER Encoder `0x2b, 0x06, 0x01, 0x04, 0x01, 0x81, 0xac, 0x5d, 0x64, 0x01;` zurück.

OID in die MIB einfügen

Diese OID kann nun in einem Array vom Datentyp `u8t` abgespeichert werden. Um Datenspeicher zu sparen kann beim Raven Board anschließend mit `PROGMEM` festgelegt werden, dass diese im Programmspeicher abgelegt wird, siehe Listing A.2.

Anschließend muss mithilfe des Datentyps `ptr_t`, welcher eine Struktur aus Pointer sowie einem unsigned Short darstellt,(siehe Abb.3.9) die Adresse sowie die Array Länge abgespeichert werden, siehe ListingA.3.

Definieren der Get und Set Funktionen

Damit der SNMP Agent weiß, welche *GET*- bzw. *SET*-Funktion beim Erhalt eines solchen Befehls aufgerufen werden soll, müssen diese festgelegt werden. Bsp. für das Setzen eines Pins am Raven Board (Quelle der get und set Funktionen Frank Schwarze - frank.schwarze@gmx.net). Siehe Listing A.4 und A.5.

Um nun das neue MIB-Objekt zu initialisieren, muss dieses noch in der Funktion `mib_init()` aufgerufen werden. Hierzu wird bei skalaren MIB Objekten die Funktion `add_scalar()` aus Listing A.6 genutzt.

Wobei als erstes die Adresse der zuvor erstellten `ptr_t`-Struktur übergeben wird. Danach können Flags definiert werden. Das dritte Argument ist der Datentyp des Objects, hier *Integer*. Als viertes Argument folgt der Initialisierungszustand und als fünftes und sechstes

Listing A.4: Beispiel GET-Funktion

```
1  s8t getBeuthState(mib_object_t* object, u8t* oid, u8t len)
2  {
3      object->varbind.value.i_value =
4      ((PORTD >> PIN7) & 1);
5      return 0;
6  }
```

Listing A.5: Beispiel SET-Funktion

```
1  s8t setBeuthState(mib_object_t* object, u8t* oid, u8t len,
       varbind_value_t value)
2  {
3      DDRD |= (1 << PIN7);
4      if (value.i_value == 1) {
5          PORTD |= (1 << PIN7);
6      } else {
7          PORTD &= ~(1 << PIN7);
8      }
9      object->varbind.value.i_value = (PORTD & (1 << PIN7));
10         return 0;
11 }
```

Listing A.6: add_scalar()

```
1  s8t add_scalar(ptr_t* oid, u8t flags, u8t value_type, const void*
       const value, get_value_t gfp, set_value_t svfp);
```

Listing A.7: Beispiel add_scalar()

```
1 if (add_scalar(&oid_beuth_int, 0, BER_TYPE_INTEGER, 0, &
    getBeuthState, &setBeuthState) == -1)
2         {
3         return -1;
4     }
```

Listing A.8: Editierung des Makefiles

```
1 raven-beuth:
2         make TARGET=avr-raven MIB_INIT=mib-init-beuth.c $(PROJECT)
              .elf
3         avr-objcopy -O ihex -R .eeprom -R .fuse -R .signature $(
              PROJECT).elf $(PROJECT).hex
4         avr-size -C --mcu=atmega1284p $(PROJECT).elf
5         rm -rf obj_native
```

müssen Pointer auf die get() und set() Funktion übergeben werden. Für das hier behandelte Beispiel ist der Aufruf in Listing A.7 zu betrachten. War der Funktionsaufruf nicht erfolgreich, so liefert `add_scalar()` -1 als Rückgabewert. Damit wurde ein neues Objekt in die MIB eingefügt. Nun wird die neue MIB unter neuem Namen gespeichert, z.B. `mib-init-beuth.c`.

Kompilierung mit neuer MIB

Damit bei der Kompilierung die neue MIB genutzt wird, muss das Makefile angepasst werden. Dazu das Makefile unter \cygwin\home\Administrator\C2X-SNMP\contiki-2.x\examples\-snmp die Datei *Makefile* editieren. Durch hinzufügen der Textzeilen in Listing A.8 und die anschließende Kompilierung mit `make raven-beuth` starten. Der anschließende Upload auf den Mikrocontroller funktioniert äquivalent zum Upload auf den RZRaven USB Stick, jedoch muss hier als Zielplattform innerhalb des AVR Studios der ATmega1284p ausgewählt werden (siehe hierzu Anhang A.4).

A.6. Installationschritte von Contiki SNMP auf Contiki OS ab Version 2.5

Eine Implementierung von SNMP auf Contiki ist bereits vorhanden, sie wurde im Rahmen einer Masterarbeit an der Jacobs University in Bremen entwickelt. Jedoch wurde diese Version für Contiki 2.4 entwickelt und ist mit dem derzeitigen Quellcode, welcher unter `http://code.google.com/p/contiki-snmp/` veröffentlicht wurde, auf Contiki 2.5 bzw. 2.6 nicht lauffähig. Es sind deshalb einige kleine Änderungen zur Anpassung auf Contiki ab Version 2.5 im Programcode nötig.

Download des SNMP-Quellcodes

Der Quellcode kann via SVN von der Google Code Seite heruntergeladen werden.

Diese Anleitung setzt die Konfigurationsschritte von *"Installation der Contiki Entwicklungsumgebung"* aus Abschnitt A.3 voraus.

Dazu in das Windows XP Gastsystem wechseln und Tortoise SVN downloaden (`http://tortoisesvn.net/`) und installieren. Anschließend in das Cygwin Verzeichnis wechseln und unter *\home Administrator* einen neuen Ordner erstellen (Bsp. *C25-SNMP*). Dann mit der rechten Maustaste in eine freie Fläche klicken und *SVN Checkout...* auswählen.(siehe Abbildung A.4). Im folgenden Fenster nun die Angaben aus Abbildung A.5 einfügen und

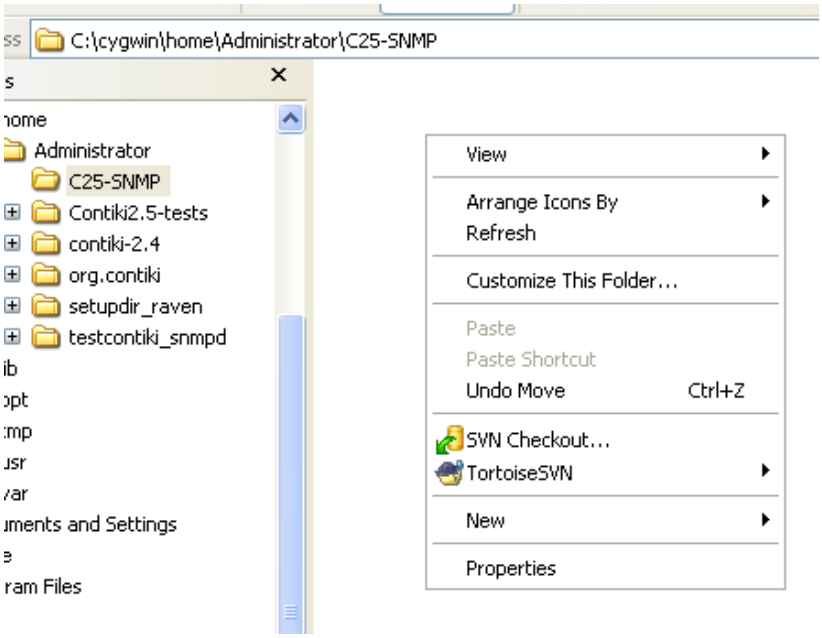


Abbildung A.4.: SVN Checkout

anschließend mit Ok bestätigen. Der Quellcode wird nun heruntergeladen und im Ordner *Source* gespeichert.

Download von Contiki 2.x

Als nächstes wird der Contiki 2.5 bzw. 2.6 Quellcode benötigt, dieser kann von `http://sourceforge.net/projects/contiki/files/Contiki/Contiki2.x/contiki-2.x.zip/download` heruntergeladen werden. Anschließend sollte er im *C2X-SNMP* Verzeichnis unter Contiki-2.x entpackt werden.

Anpassung der Ordnerstruktur

Um Anwendungen auf Contiki zu installieren sind einige Regeln sowie Festlegungen zu beachten. Der Quellcode der zu installierenden Software wird in das Contiki *apps* Verzeichnis kopiert. Da der Sourcecode des Contiki SNMPs im Ordner *cygwin\home\Administrator\C2X-SNMP\Source\src* befindet muss dieser gesamte Ordner in den Contiki 2.x *apps* Ordner (*\cygwin\home\Administrator\C2X-SNMP\contiki-2.x\apps*) kopiert werden. Anschlie-

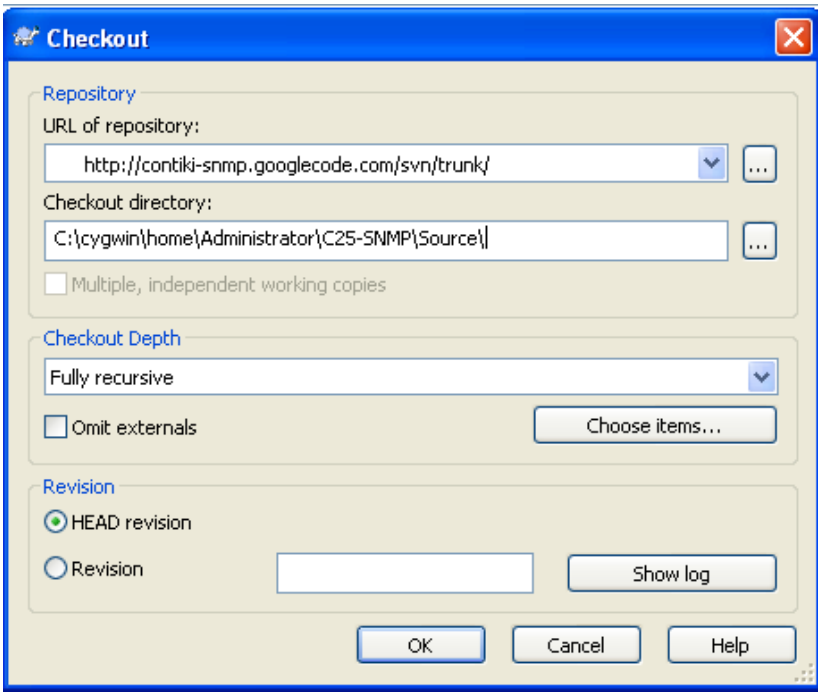


Abbildung A.5.: SVN Checkout

Tabelle A.5.: notwendige Codeänderungen in Contiki SNMP

Dateiname	**Zeile**
`logging.c`	Zeile 46 und Zeile 48
`snmpd.c`	Zeile 159 und Zeile 16

ßend wird dieser Ordner von *src* in *snmpd* umbenannt. Da die Kompilierung von Contiki auf Makefiles basiert, wird dieses noch benötigt. Es befindet sich unter *\cygwin\home\Administrator\C2X-SNMP\Source\app*. Zusätzlich befindet sich in diesem Ordner die Datei `snmp-server.c`, diese Datei enthält den Autostarteintrag für den SNMP Daemon. Nun wird dieser gesamte Ordner *app* in das Contikiverzeichnis *\cygwin\home\Administrator\C2X-SNMP\contiki-2.x\examples* kopiert und anschließend in *snmp* umbenannt.

Änderungen im Contiki Quellcode

Ab Contiki 2.5 hat sich im Vergleich zu Contiki 2.4 die Funktion `HTONS()` geändert, diese heisst unter Contiki 2.5 `UIP_HTONS()`. Es muss also in allen Dateien die Funktion von `HTONS()` nach `UIP_HTONS()` umbenannt werden. Änderungen sind im Verzeichnis *\cygwin\home\Administrator\C2X-SNMP\contiki-2.x\apps\snmpd* in den in Tabelle A.5 notwendig.

Des Weiteren existiert ab Contiki 2.6 der Datentyp `u_8t` nicht mehr, in Zeile 100 der Datei `snmpd.c` muss deshalb der Datentyp auf `u8t` geändert werden. Der Datentyp `u8t` ist in der Datei `snmpd-conf.h` definiert.

Änderungen im Makefile

Anschließend muss noch das Makefile angepasst werden, dazu in den Ordner *\cygwin\home\Administrator\C2X-SNMP\contiki-2.x\examples\snmp* wechseln und dort die Datei `Makefile` editieren. In Zeile 7 ist `CONTIKI=/data/masters/dev/contiki-2.x` durch `CONTIKI = ../..` zu ersetzten.

Kompilierung

Damit sind alle benötigten Änderungen abgeschlossen und der SNMP Daemon kann kompiliert werden. Dazu muss Cygwin gestartet werden und in das Verzeichnis *\C2X-SNMP\contiki-2.x\examples\snmp* gewechselt werden. Nun wird die Kompilierung mit `make raven` gestartet, siehe Listing A.9.

Listing A.9: make start

```
1 cd /C25-SNMP/contiki-2.5/examples/snmp
2 make raven
```

Erscheint nach der Kompilierung der Text in Abbildung A.6 so war die Kompilierung erfolgreich und es kann mit dem Upload auf den ATmega1284P begonnen werden.

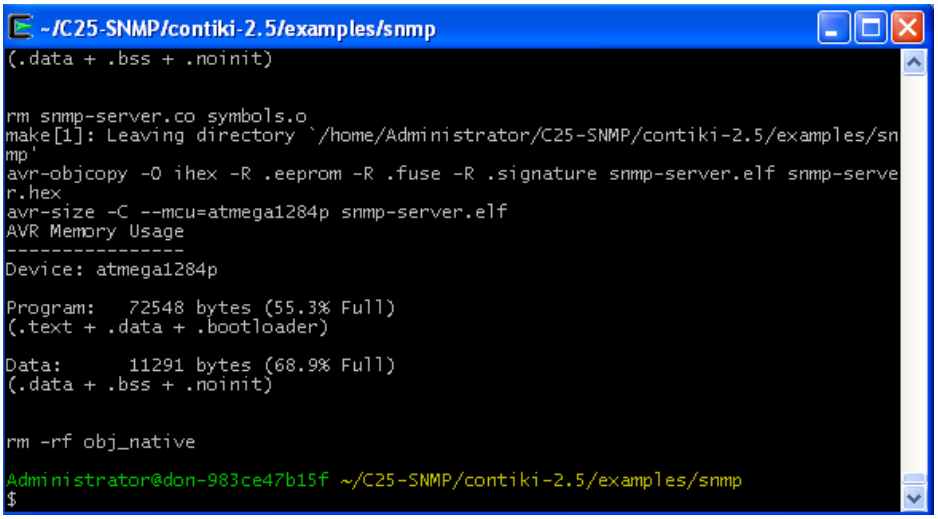


Abbildung A.6.: Erfolgreiche Kompilierung

Upload auf den AVR Raven (ATmega1284p)

Hierzu wird äquivalent zu AVR RavenRZ USB-Stick Installation aus Abschnitt A.4 vorgegangen. Der korrekte Anschluß des JTAG Kabels ist in Abbildung A.7 zu sehen.

Tabelle A.6.: Standard Einstellungen Contiki SNMP

Einstellung	**Wert**
IP Adresse	aaaa::11:22ff:fe33:4455
Read CS	public
Write CS	public
USM User	sk
Auth Password	password1
Privacy Password	password2

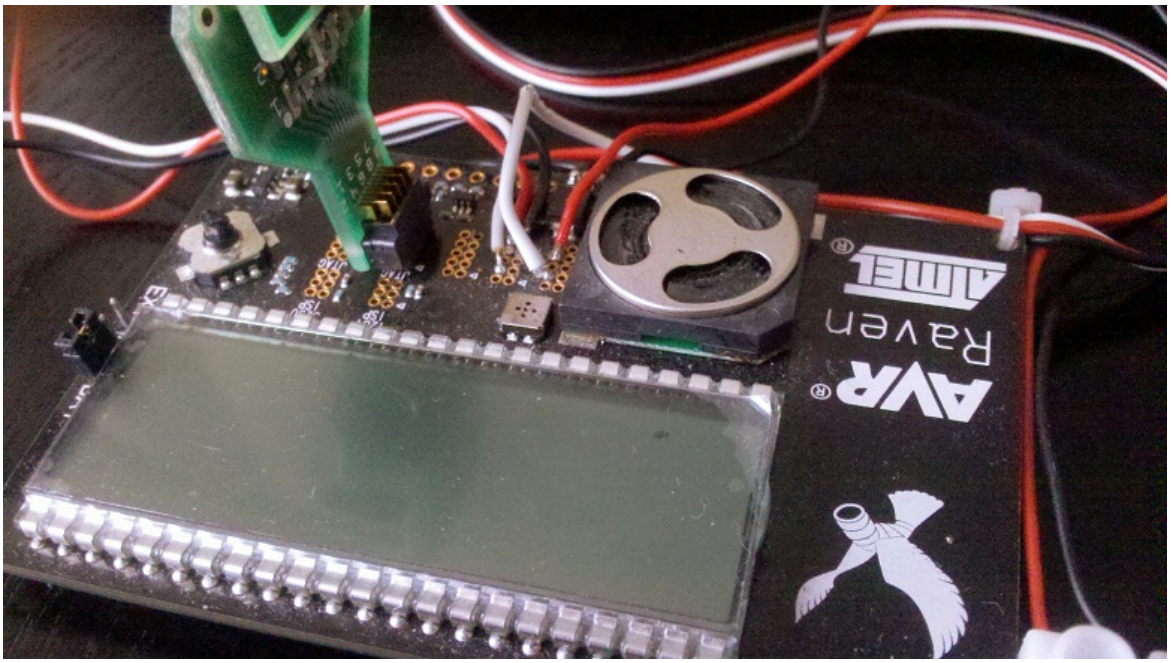


Abbildung A.7.: Anschluß des 1284p des AVR Raven Boards

In AVR Studio muss diesmal unter „Device and Signature Bytes“ als Microcontroller der ATmega1284p gewählt werden. Als `.elf-File` wird die Datei `snmp-server.elf` im Verzeichnis *\cygwin\home\Administrator\C2X-SNMP\contiki-2.x\examples\snmp*gewählt. Durch den anschließend Klick auf Program wird der Upload gestartet.

Test der Implementierung

Um das Contiki 2.5 bzw. 2.6 SNMP zu testen bietet sich das Programm MIB Browser (`http://ireasoning.com/mibbrowser.html`) an. Dazu in das Instant Contiki Gastsystem wechseln. (Die Schritte aus AVR RavenRZ USB-Stick Installation aus Abschnitt A.4 müssen abgeschlossen sein.) MIB Browser starten und die Einstellungen in der folgenden Tabelle A.6 durch Klick auf Advanced vornehmen.

A.7. Installationsschritte des erweiterten Contiki SNMP Agenten auf der Zigbit Plattform

Der Quellcode, des in dieser Arbeit erweiterten Contiki SNMP Agenten, ist auf der beigefügten CD im Verzeichnis *\Quellcode\contiki-2.6-6LoWPAN Socket* zu finden. In diesem Ordner befindet sich außerdem die neuste Contiki Version 2.6 sowie die angepasste Ordnerstruktur, welche zur Installation notwendig ist. Es wird empfohlen den kompletten Ordner auf die Festplatte zu kopieren um anschließend eine Neukompilierung durchführen zu können. Danach sind die folgenden Schritte notwendig:

1. Cygwin starten und in den Ordner *\contiki-2.6-6LoWPAN Socket\examples\snmp* wechseln.
2. Kompilierung mit `make zigbit` starten.
3. Nach Fertigstellung des Kompilierungsvorgangs AVR Studio 4 starten und wie in den vorherigen Abschnitten beschrieben einen Upload durchführen. Diesmal jedoch als Zielplattform *ATmega1281* auswählen. Der korrekte Anschluss des JTAG Adapters ist in Abbildung A.16 und A.17 zu sehen. Die benötigte `.elf` Datei ist im Kompilierungsverzeichnis unter dem Namen `snmp-server.elf` zu finden.

Die für den Zugriff benötigten Daten sind Tabelle A.6 zu entnehmen.

A.8. Liste der implementierten managed Objekts

Tabelle A.7.: Auflistung der MIB Objekte

Name	**OID**	**Typ**	**Beschreibung**
sysDescr	.1.3.6.1.2.1.1.1.0	Octet String	System Beschreibung
sysUpTime	.1.3.6.1.2.1.1.3.0	Time Ticks	vergangene Zeit seit dem Systemstart
sysContact	.1.3.6.1.2.1.1.4.0	Octet String	Kontaktperson des Systems
sysName	.1.3.6.1.2.1.1.5.0	Octet String	Name des Systems
sysLocation	.1.3.6.1.2.1.1.6.0	Octet String	System Standort
snmpInPkts	.1.3.6.1.2.1.11.1.0	Counter	Anzahl der eingegangenen SNMP Pakete
snmpInBadVersions	.1.3.6.1.2.1.11.3.0	Counter	Anzahl der eingegangenen SNMP Pakete mit nicht unterstützter Version
snmpInASNParseErrs	.1.3.6.1.2.1.11.6.0	Counter	Anzahl der Fehler bei der Decodierung mithilfe der BER
beuthSocketControl	.1.3.6.1.4.1.22109.100.600.1	Integer	Ausgangspinsteuerung der 6LoWPAN Steckdose
beuthRadioStrength	.1.3.6.1.4.1.22109.100.600.2	Integer	Abfrage der Funkempfangsstärke in dBm
beuthTemperature	.1.3.6.1.4.1.22109.100.600.3	Integer	Abfrage der Gerätetemperatur in Grad Celsius

A.9. Funktionen und Variablen der Temperaturmessung

Listing A.10: Initialisierung des ADC

```
1 void adc_init()
2 {
3     // AREF = AVcc
4     ADMUX = (1<<REFS0);
5     // ADC Enable and prescaler of 128
6     ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
7 }
```

Listing A.11: Berechnung des Widerstandswertes von R_{NTC} mit der Funktion adcToOhm()

```
1 float adcToOhm(int adc_value)
2 {
3         float one_adc=0.003076171875;
4         float r_div=10000;
5         float u_ges=3.15;
6         float r_ntc;
7         r_ntc=(((float)adc_value)*one_adc*r_div)/(u_ges-(((float)
              adc_value)*one_adc));
8         return r_ntc;
9 }
```

Listing A.12: Abfrage des ADC Registers

```
1  u16t adc_read()
2  {
3      ADMUX = (ADMUX & 0xF8)|3;       // clears the bottom 3 bits and
           sets channel
4      // start single convertion
5      // write '1' to ADSC
6      ADCSRA |= (1<<ADSC);
7      // wait for conversion to complete
8      while(ADCSRA & (1<<ADSC));
9
10     return (ADC);
11 }
```

Listing A.13: Temperaturwertearray im Programmspeicher

```
1
2  u16t temp_array[] PROGMEM ={
3                   325, 311, 296, 282, 267,
4                   253, 242, 231, 220, 209,
5                   198, 189, 182, 173, 165,
6                   157, 151, 144, 138, 131,
7                   125, 120, 115, 110, 105,
8                   100, 96, 92, 88, 84,
9                   80, 77, 74, 71, 68,
10                  65, 63, 60, 58, 55,
11                  53, 51, 49, 47, 45,
12                  43, 42, 40, 39, 37,
13                  36, 35, 33, 32, 30,
14                  29, 28, 27, 27, 26,
15                  25, 24, 23, 23, 22,
16                  21, 20, 19, 19, 18,
17                  17, 17, 16, 16, 15,
18                  15, 14, 14, 13, 13,
19                  12, 12, 12, 11, 11,
20                  11, 11, 10, 10, 9,
21                  9, 9, 8, 8, 7,
22                  7, 7, 7, 6, 6,
23                  6};
```

Listing A.14: Temperaturvergleichsfunktion find_temp_celsius(float r_ntc)

```
1
2  u8t find_temp_celsius(float r_ntc)
3  {
4
5          float rt_r25;
6          float r_div=10000;
7          rt_r25=(r_ntc/r_div)*100;
8          u8t i=0;
9          while((u16t)rt_r25 <= getTempArray(i))
10         {
11                 i++;
12         }
13         return (i-1);
14
15 }
```

A.10. Screenshots der Wireshark Messungen

→ alle auch verfügbar auf der beigefügten CD im Verzeichnis Wireshark.

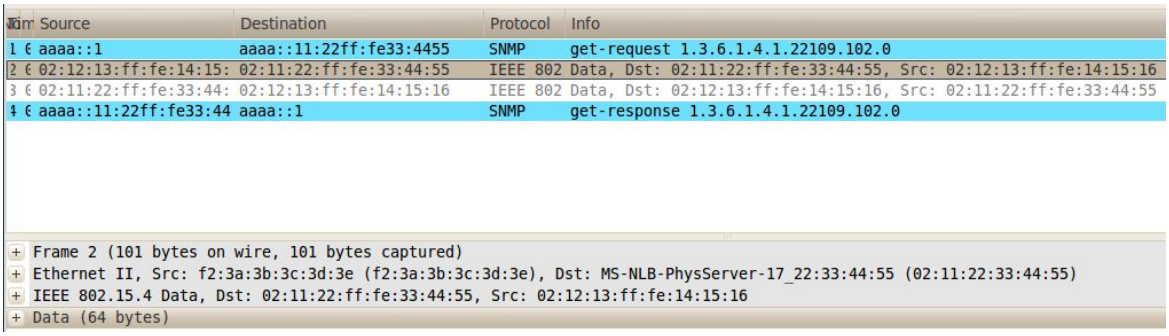


Abbildung A.8.: SNMPv1 Paket, Fragmentierung in 802.15.4 Frames

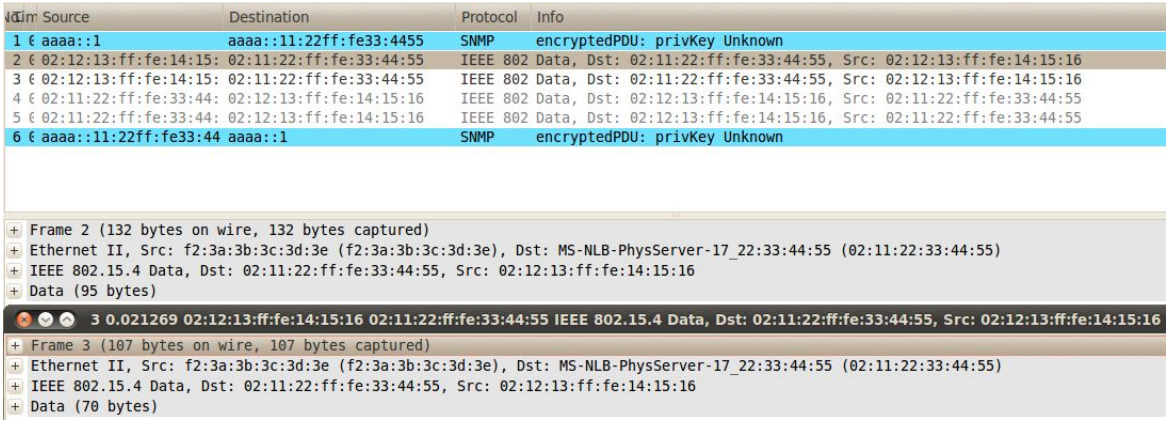


Abbildung A.9.: SNMPv3 AuthPriv Paket, Fragmentierung in 802.15.4 Frames

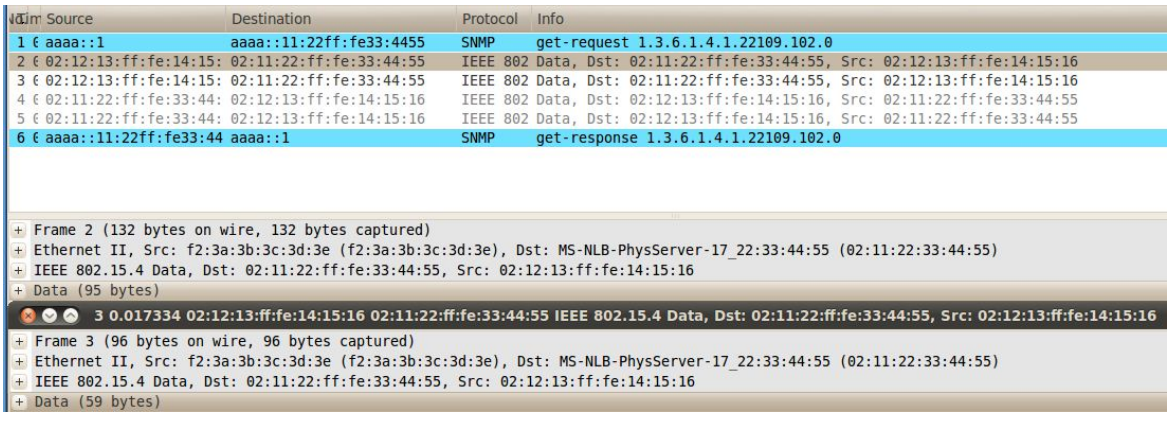


Abbildung A.10.: SNMPv3 AuthNoPriv Paket, Fragmentierung in 802.15.4 Frames

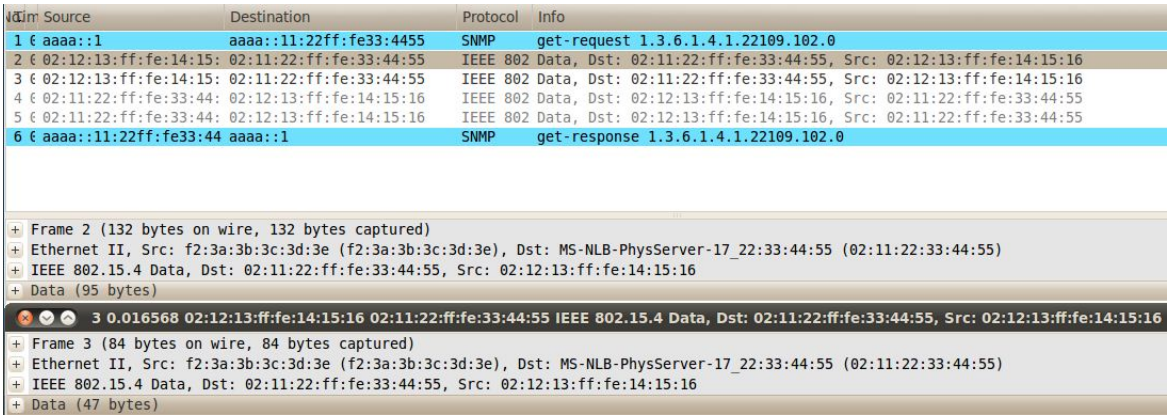


Abbildung A.11.: SNMPv3 NoAuthNoPriv Paket, Fragmentierung in 802.15.4 Frames

A.11. Testplan zur Überprüfung der SNMPv3 Sicherheit

Dieser Abschnitt enthält den Testplan sowie die Ergebnisse der Durchführung zu den Funktionstests der Sicherheit der 6LoWPAN SNMPv3 Steckdose.
Alle Tests wurden unter einer Ubuntu x86 Plattform innerhalb einer VMWare Player Umgebung durchgeführt. Verwendet wurden außerdem die Programme *Wireshark*[1] in der Version 1.2.7, das Programm *gHex*[2], das Programm *Tcprewrite* beinhaltet in dem Programm *Tcpreplay*[3], verwendete Version 3.4.4, sowie das Programm *Scapy*[4] in der Version 2.2.0. Außerdem kommt das Programm *MIB Browser Enterprise Edition*[5] der Firma Ireasoning in der Version 8.1 mehrfach zum Einsatz.

Test **Nr 1**	*Get Request mit falschem USM Benutzername*
Beschreibung	Versenden eines Get Requests, wobei das SNMPv3 Feld *msgUserName* einen Benutzernamen beinhaltet, der nicht auf dem System existiert.
Vorgehen	Mit der Software Ireasoning MIB Browser einen Get Request zur OID .1.3.6.1.2.1.1.3.0 absenden, wobei alle SNMPv3 Sicherheitsfelder korrekte Werte enthalten, bis auf den das Feld *msgUserName*, anschließend die Antwort des Agenten mit Wireshark beobachten. Einstellungen des SNMP Agenten: SNMPv3 enabled, Authentication enabled, Privacy Enabled, SNMPv1 disabled.
Erwartetes Ergebnis	Laut RFC3414 [Blumenthal and Wijnen, 2002] muss eine *Report* Nachricht mit einem Variable Bindings welches den Wert der OID .1.3.6.1.6.3.15.1.1.3.0 zurückgesendet werden. .1.3.6.1.6.3.15.1.1.3.0 steht für "usmStatsUnknownUserNames" und enthält die Anzahl von Nachrichten die der SNMPv3 Agent erhalten hat, worin ein im System nicht vorhandener Benutzername eingetragen war.
Ergebnis der Messung	report mit 1.3.6.1.6.3.15.11.3.0 "usmStatsUnknownUserNames.0"
Status der Messung	erfolgreich, Test bestanden

[1] https://www.wireshark.org/
[2] https://live.gnome.org/Ghex/
[3] http://tcpreplay.synfin.net/
[4] http://www.secdev.org/projects/scapy/
[5] http://ireasoning.com/mibbrowser.shtml

Test **Nr 2**	*Get Request mit falschem USM Authentication Passwort*
Beschreibung	Versenden eines Get Requests, wobei zur Durchführung der HMAC96 Durchführung ein falsches Authentication Passwort benutzt wird.
Vorgehen	Mit der Software Ireasoning MIB Browser einen Get Request zur OID .1.3.6.1.2.1.1.3.0 absenden, wobei alle SNMPv3 Sicherheitsfelder korrekte Werte enthalten, jedoch wurde zur Berechnung der HMAC96 Parameter ein falsches Authentication Passwort verwendet, anschließend die Antwort des Agenten mit Wireshark beobachten. Einstellungen des SNMP Agenten: SNMPv3 enabled, Authentication enabled, Privacy Enabled, SNMPv1 disabled.
Erwartetes Ergebnis	Laut RFC3414 [Blumenthal and Wijnen, 2002] muss eine *Report* Nachricht mit einem Variable Bindings welches den Wert der OID .1.3.6.1.6.3.15.1.1.5.0 zurückgesendet werden. .1.3.6.1.6.3.15.1.1.5.0 steht für "usmStatsWrongDigests" und enthält die Anzahl von Nachrichten die der SNMPv3 Agent erhalten hat, worin ein Fehler durch das Authentication Modul festgestellt wurde.
Ergebnis der Messung	report mit 1.3.6.1.6.3.15.11.1.1.5.0 "usmStatsWrongDigests"
Status der Messung	erfolgreich, Test bestanden

Test **Nr 3**	*Get Request mit falschem USM Privacy Passwort*
Beschreibung	Versenden eines Get Requests, wobei zur Durchführung der AES Verschlüsselung ein falsches Privacy Passwort benutzt wird.
Vorgehen	Mit der Software Ireasoning MIB Browser einen Get Request zur OID .1.3.6.1.2.1.1.3.0 absenden, wobei alle SNMPv3 Sicherheitsfelder korrekte Werte enthalten, jedoch wurde zur Verschlüsselung ein falsches Privacy Passwort verwendet, anschließend die Antwort des Agenten mit Wireshark beobachten. Einstellungen des SNMP Agenten: SNMPv3 enabled, Authentication enabled, Privacy Enabled, SNMPv1 disabled.
Erwartetes Ergebnis	Laut RFC3414 [Blumenthal and Wijnen, 2002] muss eine *Report* Nachricht mit einem Variable Bindings welches den Wert der OID .1.3.6.1.6.3.15.1.1.6 zurückgesendet werden. .1.3.6.1.6.3.15.1.1.6 steht für "usmStatsDecryptionErrors" und enthält die Anzahl von Nachrichten die der SNMPv3 Agent erhalten hat, worin ein Fehler durch das Privacy Modul festgestellt wurde.
Ergebnis der Messung	Keine Antwort, Paket wird verworfen, SNMP Agent generiert keinen report.
Status der Messung	nicht erfolgreich, jedoch keine Gefährdung der Sicherheit

Test **Nr 4**	*Get Request mit falscher Protokollversion (Version 1 statt Version 3)*
Beschreibung	Versenden eines Get Requests, wobei das SNMPv3 Feld *msgVersion* anstatt 3 den Wert 0 enthält.
Vorgehen	Mit der Software Ireasoning MIB Browser einen Get Request zur OID .1.3.6.1.2.1.1.3.0 absenden, wobei alle SNMPv3 Sicherheitsfelder korrekte Werte enthalten. Gleichzeitig das abgesendete Paket mit Wireshark abfangen und abspeichern. Im Anschluß gespeichertes Paket mit Hex Editor öffnen und den Wert des Feldes *msgVersion* von 03 auf 00 ändern. Im nächsten Schritt mithilfe des Programms tcprewrite die Integrität des Pakets wiederherstellen. Zum Schluß das Paket mit Scapy absenden. Anschließend die Antwort des Agenten mit Wireshark beobachten. Einstellungen des SNMP Agenten: SNMPv3 enabled, Authentication enabled, Privacy Enabled, SNMPv1 disabled. Es sollte außerdem die OID .1.3.6.1.2.1.11.3.0 überprüft werden, der Counter muss nach dem Absenden des Pakets inkrementiert worden sein.
Erwartetes Ergebnis	Laut RFC3413 [Levi et al., 2002] muss der Counter "snmpInBadVersions" inkrementiert werden. Das Paket sollte verworfen werden.
Ergebnis der Messung	Keine Antwort, Paket wird verworfen, SNMP Agent generiert keinen report. OID .1.3.6.1.2.1.11.3.0 wurde inkrementiert.
Status der Messung	erfolgreich, Test bestanden.

Test **Nr 5**	*Get Request mit veränderten Authentication Parameter*
Beschreibung	Versenden eines Get Requests, wobei das SNMPv3 Feld *msgAuthenticationParameter* mit 0xFF überschrieben wurde.
Vorgehen	Mit der Software Ireasoning MIB Browser einen Get Request zur OID .1.3.6.1.2.1.1.3.0 absenden, wobei alle SNMPv3 Sicherheitsfelder korrekte Werte enthalten. Gleichzeitig das abgesendete Paket mit Wireshark abfangen und abspeichern. Im Anschluß gespeichertes Paket mit Hex Editor öffnen und den Wert des Feldes *msgAuthenticationParametern* mit 0xFF überschreiben. Im nächsten Schritt mithilfe des Programms tcprewrite die Integrität des Pakets wiederherstellen. Zum Schluß das Paket mit Scapy absenden. Anschließend die Antwort des Agenten mit Wireshark beobachten. Einstellungen des SNMP Agenten: SNMPv3 enabled, Authentication enabled, Privacy Enabled, SNMPv1 disabled.
Erwartetes Ergebnis	Laut RFC3414 [Blumenthal and Wijnen, 2002] muss eine *Report* Nachricht mit einem Variable Bindings welches den Wert der OID .1.3.6.1.6.3.15.1.1.5.0 zurückgesendet werden. .1.3.6.1.6.3.15.1.1.5.0 steht für "usmStatsWrongDigests" und enthält die Anzahl von Nachrichten die der SNMPv3 Agent erhalten hat, worin ein Fehler durch das Authentication Modul festgestellt wurde.
Ergebnis der Messung	Keine Antwort, Paket wird verworfen, SNMP Agent generiert keinen report.
Status der Messung	nicht erfolgreich, jedoch keine Gefährdung der Sicherheit

Test **Nr 6**	*Get Request mit Engine Time die den Wert der lokalen Engine Time um mindestens 150s unter- bzw. überschreitet*
Beschreibung	Versenden eines Get Requests, wobei das SNMPv3 Feld *msgAutoritativeEngineTime* mit 0x01 bzw. 0xFF überschrieben wurde.
Vorgehen	Mit der Software Ireasoning MIB Browser einen Get Request zur OID .1.3.6.1.2.1.1.3.0 absenden, wobei alle SNMPv3 Sicherheitsfelder korrekte Werte enthalten. Gleichzeitig das abgesendete Paket mit Wireshark abfangen und abspeichern. Im Anschluß gespeichertes Paket mit Hex Editor öffnen und den Wert des Feldes *msgAutoritativeEngineTime* mit 0x01 überschreiben. Im nächsten Schritt mithilfe des Programms tcprewrite die Integrität des Pakets wiederherstellen. Zum Schluß das Paket mit Scapy absenden. Anschließend die Antwort des Agenten mit Wireshark beobachten. Einstellungen des SNMP Agenten: SNMPv3 enabled, Authentication enabled, Privacy Enabled, SNMPv1 disabled. Um die Werte für das HMAC Authentication Modul nicht neuberechnen zu müssen, muss das HMAC Authentication Modul vorübergehend abgeschaltet werden. Dazu in der Datei `usm.c` das Flag `#define DISABLE_HMAC` von Null auf Eins setzen und eine neue Kompilierung durchführen. Gleiche Vorgehensweise für Zeitüberschreitung wiederholen, *msgAutoritativeEngineTime* mit 0xFF überschreiben.
Erwartetes Ergebnis	Laut RFC3414 [Blumenthal and Wijnen, 2002] muss eine *Report* Nachricht mit einem Variable Bindings welches den Wert der OID .1.3.6.1.6.3.15.1.1.2.0 zurückgesendet werden. .1.3.6.1.6.3.15.1.1.2.0 steht für "usmStatsNotInTimeWindows" und enthält die Anzahl von Nachrichten die der SNMPv3 Agent erhalten hat, wobei die SNMP Engine Time um 150s unter oder überschritten wurde, bzw. die SNMP Engine Boots Anzahl nicht übereinstimmt
Ergebnis der Messung	report mit 1.3.6.1.6.3.15.11.1.1.2.0 "usmStatsNotInTimeWindows'"
Status der Messung	erfolgreich, Test bestanden.

Test **Nr 7**	*Get Request mit flascher Engine Boots Anzahl*
Beschreibung	Versenden eines Get Requests, wobei das SNMPv3 Feld *msgAutoritativeEngineBoots* mit 0xFF überschrieben wurde.
Vorgehen	Mit der Software Ireasoning MIB Browser einen Get Request zur OID .1.3.6.1.2.1.1.3.0 absenden, wobei alle SNMPv3 Sicherheitsfelder korrekte Werte enthalten. Gleichzeitig das abgesendete Paket mit Wireshark abfangen und abspeichern. Im Anschluß gespeichertes Paket mit Hex Editor öffnen und den Wert des Feldes *msgAutoritativeEngineBoots* mit 0xFF überschreiben. Im nächsten Schritt mithilfe des Programms tcprewrite die Integrität des Pakets wiederherstellen. Zum Schluß das Paket mit Scapy absenden. Anschließend die Antwort des Agenten mit Wireshark beobachten. Einstellungen des SNMP Agenten: SNMPv3 enabled, Authentication enabled, Privacy Enabled, SNMPv1 disabled.Um die Werte für das HMAC Authentication Modul nicht neuberechnen zu müssen, muss das HMAC Authentication Modul vorübergehend abgeschaltet werden. Dazu in der Datei `usm.c` das Flag `#define DISABLE_HMAC` von Null auf Eins setzen und eine neue Kompilierung durchführen. Eventuell kann auch die AuthoritativeEngineTime Überprüfung deaktiviert werden, dazu in der gleichen Datei das Flag `#define DISABLE_MAET` auf Eins setzen.
Erwartetes Ergebnis	Laut RFC3414 [Blumenthal and Wijnen, 2002] muss eine *Report* Nachricht mit einem Variable Bindings welches den Wert der OID .1.3.6.1.6.3.15.1.1.2.0 zurückgesendet werden. .1.3.6.1.6.3.15.1.1.2.0 steht für “usmStatsNotInTimeWindows“ und enthält die Anzahl von Nachrichten die der SNMPv3 Agent erhalten hat, wobei die SNMP Engine Time um 150s unter oder überschritten wurde bzw. die SNMP Engine Boots Anzahl nicht übereinstimmt.
Ergebnis der Messung	report mit 1.3.6.1.6.3.15.11.1.1.2.0 “usmStatsNotInTimeWindows“
Status der Messung	erfolgreich, Test bestanden.

Test **Nr 8**	*Get Request mit verbotener msgFlags Kombination*
Beschreibung	Versenden eines Get Requests, wobei das SNMPv3 Feld *msgFlags* die Kombination Binär 0110 bzw. 0x06 (NoAuthPriv + Report Request) enthält.
Vorgehen	Mit der Software Ireasoning MIB Browser einen Get Request zur OID .1.3.6.1.2.1.1.3.0 absenden, wobei alle SNMPv3 Sicherheitsfelder korrekte Werte enthalten. Gleichzeitig das abgesendete Paket mit Wireshark abfangen und abspeichern. Im Anschluß gespeichertes Paket mit Hex Editor öffnen und den Wert des Feldes *msgFlags* mit 0x06 überschreiben. Im nächsten Schritt mithilfe des Programms tcprewrite die Integrität des Pakets wiederherstellen. Zum Schluß das Paket mit Scapy absenden. Anschließend die Antwort des Agenten mit Wireshark beobachten. Einstellungen des SNMP Agenten: SNMPv3 enabled, Authentication enabled, Privacy Enabled, SNMPv1 disabled.
Erwartetes Ergebnis	Laut RFC3414 [Blumenthal and Wijnen, 2002] muss im Falle der verbotenen *msgFlags* Kombination die Nachricht verworfen werden.
Ergebnis der Messung	keine Antwort, Nachricht wird verworfen.
Status der Messung	erfolgreich, Test bestanden.

Test **Nr 9**	*Get Request mit ungültiger Security Model Nummer*
Beschreibung	Versenden eines Get Requests, wobei das SNMPv3 Feld *msgSecurityModel* den Wert 0x09 enthält.
Vorgehen	Mit der Software Ireasoning MIB Browser einen Get Request zur OID .1.3.6.1.2.1.1.3.0 absenden, wobei alle SNMPv3 Sicherheitsfelder korrekte Werte enthalten. Gleichzeitig das abgesendete Paket mit Wireshark abfangen und abspeichern. Im Anschluß gespeichertes Paket mit Hex Editor öffnen und den Wert des Feldes *msgSecurityModel* mit 0x09 überschreiben. Im nächsten Schritt mithilfe des Programms tcprewrite die Integrität des Pakets wiederherstellen. Zum Schluß das Paket mit Scapy absenden. Anschließend die Antwort des Agenten mit Wireshark beobachten. Einstellungen des SNMP Agenten: SNMPv3 enabled, Authentication enabled, Privacy Enabled, SNMPv1 disabled.
Erwartetes Ergebnis	Laut RFC2572 [Case et al., 2002a] muss im Falle eines nicht vorhanden Sicherheitsmodells die Nachricht verworfen werden.
Ergebnis der Messung	keine Antwort, Nachricht wird verworfen.
Status der Messung	erfolgreich, Test bestanden.

Test **Nr 10**	*Get Request mit ungültiger Engine ID*
Beschreibung	Versenden eines Get Requests, wobei das SNMPv3 Feld *msgAuthoritativeEngineID* mit 0xFF überschrieben wurde.
Vorgehen	Mit der Software Ireasoning MIB Browser einen Get Request zur OID .1.3.6.1.2.1.1.3.0 absenden, wobei alle SNMPv3 Sicherheitsfelder korrekte Werte enthalten. Gleichzeitig das abgesendete Paket mit Wireshark abfangen und abspeichern. Im Anschluß gespeichertes Paket mit Hex Editor öffnen und den Wert des Feldes *msgAuthoritativeEngineID* mit 0xff überschreiben. Im nächsten Schritt mithilfe des Programms tcprewrite die Integrität des Pakets wiederherstellen. Zum Schluß das Paket mit Scapy absenden. Anschließend die Antwort des Agenten mit Wireshark beobachten. Einstellungen des SNMP Agenten: SNMPv3 enabled, Authentication enabled, Privacy Enabled, SNMPv1 disabled.
Erwartetes Ergebnis	Laut RFC3414 [Blumenthal and Wijnen, 2002] muss eine *Report* Nachricht mit einem Variable Bindings welches den Wert der OID .1.3.6.1.6.3.15.1.1.4.0 zurückgesendet werden. .1.3.6.1.6.3.15.1.1.4.0 steht für "usmStatsUnknownEngineID" und enthält die Anzahl von Nachrichten die der SNMPv3 Agent erhalten hat, wobei die SNMP Engine ID nicht mit der des SNMP Agenten übereinstimmt.
Ergebnis der Messung	report mit 1.3.6.1.6.3.15.11.1.1.4.0 "usmStatsusmStatsUnknownEngineID"
Status der Messung	erfolgreich, Test bestanden.

A.12. Hardware

A.12.1. Schaltplan 6LoWPAN Socket

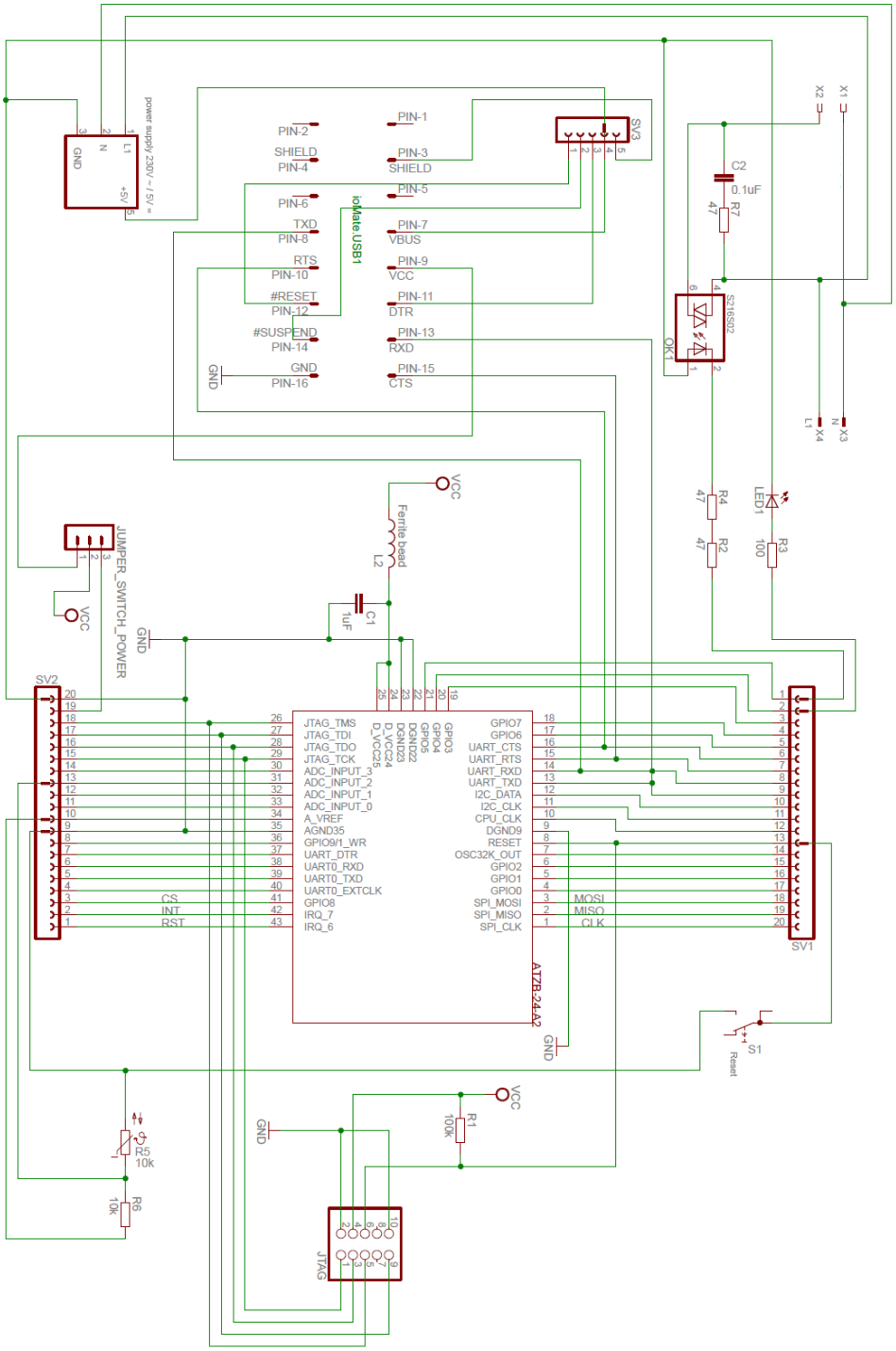


Abbildung A.12.: Schaltplan 6LoWPAN Socket

A.12.2. AVR Raven und RZRavenUSBStick

In diesem Abschnitt ist ein Auszug des Datenblatts des AVR Raven Boards zu finden, das vollständige Datenblatt ist entweder von der Atmel Homepage[6] oder auf der dieser Arbeit beigefügten CD im Verzeichnis *Datenblätter\Raven * in der Datei `RZRAVENGUIDE.pdf` zu finden. Die Datenblätter für die auf dem Raven Board verbauten Microcontroller ATmega1284p und ATmega3290 sind im gleichen Verzeichnis unter `Atmega1284p.pdf` und `Atmega3290.pdf` gespeichert. Sie sind ebenso über die Homepage von Atmel[7] zu beziehen. Der verwendete Funkchip ist ebenfalls von Atmel und trägt die Typbezeichnung AT86RF230 das Datenblatt ist im Verzeichnis *Datenblätter\AT86RF230 * in der Datei `AT86RF230.pdf` oder auf der Atmel Homepage[8] zu finden.

Außerdem beinhaltet das Raven Set von Atmel den AVR RZUSBStick, welcher als Router zwischen 6LoWPAN, IEEE 802.15.4 und IPv6 fungiert. Durch den USB Stick erhält ein gewöhnlicher PC ein 802.15.4 Interface und ermöglicht so die Kommunikation zwischen PC und Sensor. Der verwendete Prozessor trägt die Typenbezeichnung AT90USB1287, sein Datenblatt ist Verzeichnis *Datenblätter\RZUSBStick* in der Datei `AT90USB1287.pdf` und auf der Atmel Homepage verfügbar[9]. Der verwendete Funkchip ist wie bei den Raven Boards auch der AT86RF230.

Auf den folgenden Seiten ist ein kurzer Auschnitt des Datenblatts mit den wichtigsten Spezifikationen zu finden.

[6] `http://www.atmel.com/Images/doc8117.pdf`
[7] Atmega1284p: `http://www.atmel.com/Images/doc8272.pdf` ATmega3290: `http://www.atmel.com/Images/doc2552.pdf`
[8] `http://www.atmel.com/Images/doc5131.pdf`
[9] `http://www.atmel.com/Images/doc7593.pdf`

AVR2016: RZRAVEN Hardware User's Guide

8-bit AVR® Microcontrollers

Application Note

Features

- **Development kit for the AT86RF230 radio transceiver and AVR® microcontroller.**
- **CE, ETSI and FCC approved.**
- **LCD module (AVRRAVEN):**
  - **AT86RF230 radio transceiver with high gain PCB antenna.**
  - **Dual AVR microcontrollers.**
  - **Dynamic Speaker and microphone.**
  - **Atmel Serial Dataflash®.**
  - **User IO section:**
    - **USART**
    - **GPIO**
    - **Relay Driver**
  - **Powered by battery or external supply:**
    - **5V to 12V external supply.**
- **USB module (RZUSBSTICK):**
  - **AT86RF230 radio transceiver with miniature PCB antenna.**
  - **AVR microcontroller with integrated Full Speed USB interface.**
  - **External memory interface.**

1 Introduction

The RZRAVEN is a development kit for the AT86RF230 radio transceiver and the AVR microcontroller. It serves as a versatile and professional platform for developing and debugging a wide range of RF applications; spanning from: simple point-to-point communication through full blown sensor networks with numerous nodes running complex communication stacks. On top of this, the kit provides a nice human interface, which spans from PC connectivity, through LCD and audio input and output.

Figure 1-1. The RZRAVEN Kit Modules

Rev. 8117D-AVR-04/08

2 General

The RZRAVEN kit is built from one RZUSBSTICK module and two AVRRAVEN modules. See Figure 2-1 to Figure 2-4 for further details.

The complete schematics and Gerber files are available from the compressed archive accompanying this application note.

Figure 2-1 Assembly drawing AVRRAVEN - front view.

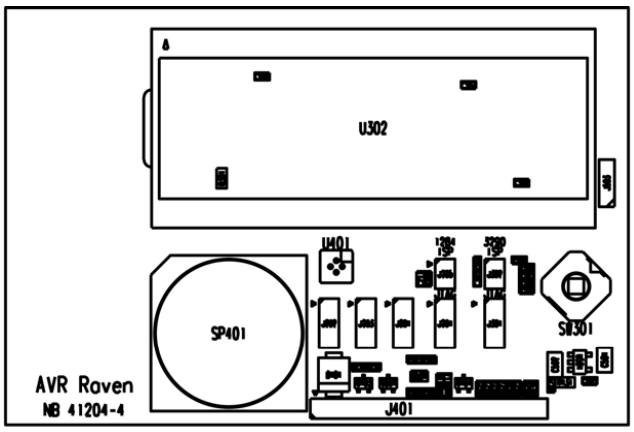


Figure 2-2 Assembly drawing AVRRAVEN - back view.

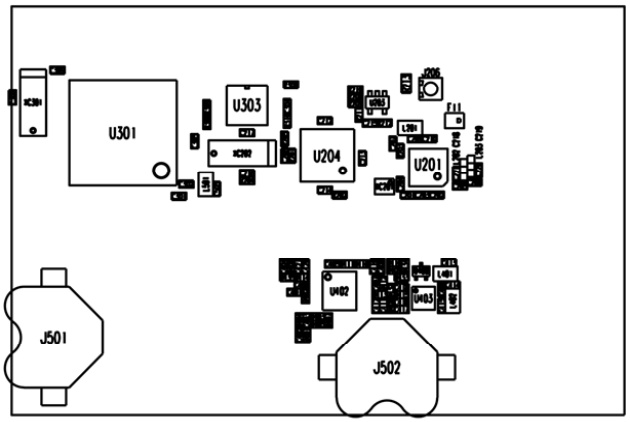


Figure 2-3 Assembly drawing RZUSBSTICK - front view.

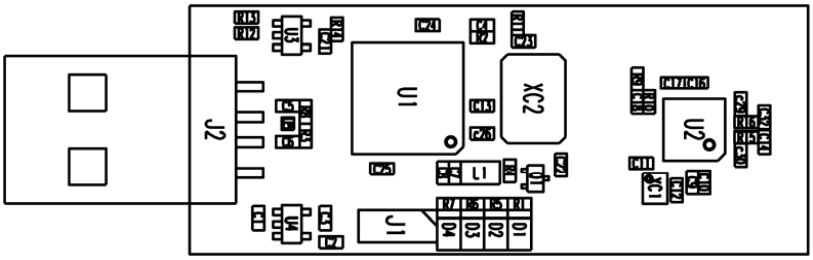
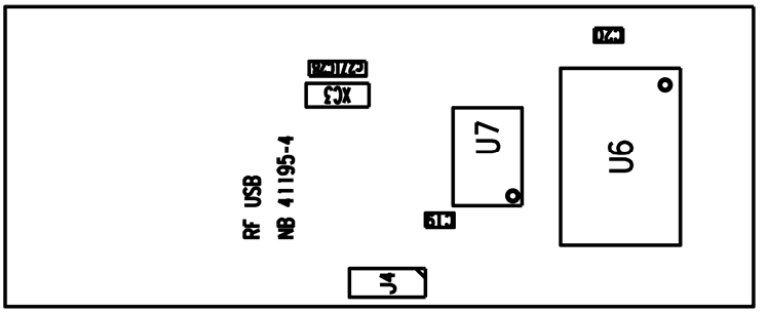
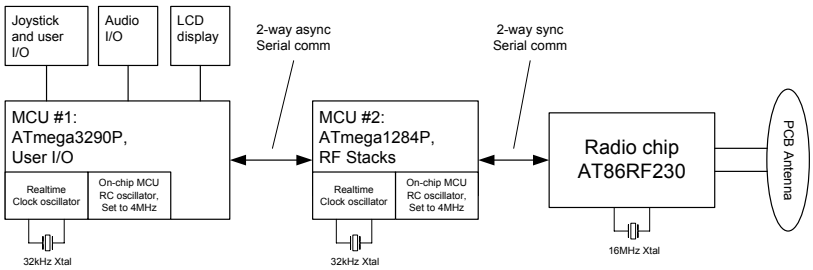


Figure 2-4 Assembly drawing RZUSBSTICK - back view



3 The AVRRAVEN Module

Figure 3-1 AVRRAVEN overview

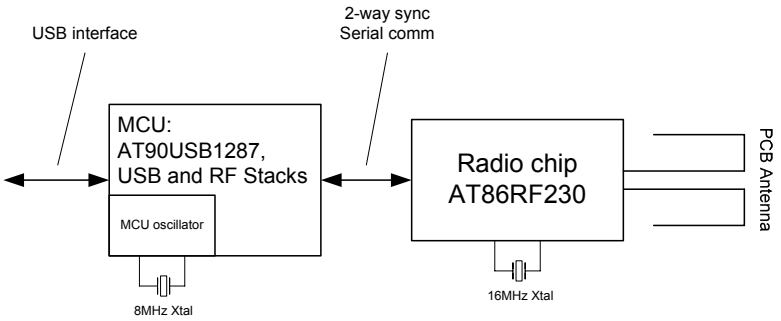


The AVRRAVEN hardware is based on 2 microcontroller and one radio transceiver chip. The ATmega3290P handles the sensors and the user interface and the ATmega1284P handles the AT86RF230 radio transceiver and the RF protocol stacks. The MCUs and the radio communicate via serial interfaces.

For hardware details please refer to Appendix A for the complete AVRRAVEN schematics.

4 The AVR RZUSBSTICK Module

Figure 4-1 RZUSBSTICK overview



The AVR RZUSBSTICK hardware is based a USB microcontroller and a radio transceiver chip. The AT90USB1287 microcontroller handles the USB interface, the AT86RF230 radio transceiver and the RF protocol stacks.

For hardware details please refer to Appendix D for the complete AVR RZUSBSTICK schematics.

4.1 AVR Microcontroller

The AT90USB1287 is a device in the family of AVRs with a low and full speed USB macro with device, host and On-the-go (OTG) capabilities.

4.2 Atmel Radio Transceiver

The AT86RF230 is a 2.4GHz radio transceiver that is tailored for a wide range of wireless applications. Low power consumption and market leading RF performance makes it an excellent choice for virtually any type of networking device. Support for IEEE 802.15.4 (Automatic acknowledge of packets, address filtering and automatic channel access) type of applications is available through an enhanced layer of functionality on top of the basic radio transceiver.

4.3 Antenna description

The antenna on the RZUSBSTICK is a folded dipole antenna with a net peak gain of 0dB

A.12.3. AVR Zigbit

Das Zigbit Modul von Atmel bietet auf kleinsten Abmessungen von nur 24 x 13,5 x 2 mm einen ATmega 1281 Microcontroller und einen AT86RF230 Funkchip sowie eine integrierte Antenne. Der Funkchip AT86RF230 entspricht dem Funkchip des Raven Boards.
Das Datenblatt des AT86RF230 ist im Verzeichnis *Datenblätter\AT86RF230 * in der Datei `AT86RF230.pdf` oder auf der Atmel Homepage[10] zu finden.
Der verwendete Prozessor ATmega 1281 bietet 128kByte Flash Speicher, 4kByte EEPROM und 8kByte RAM, sein Datenblatt ist Verzeichnis *Datenblätter\Zigbit* in der Datei `ATmega1281.pdf` und auf der Atmel Homepage verfügbar[11].
Das Datenblatt des kompletten Moduls ist ebenfalls im Ordner *Datenblätter\Zigbit* jedoch unter Datei `Zigbit.pdf` und im Internet zu finden[12].

Auf den folgenden Seiten ist ein kurzer Auschnitt des Datenblatts mit den wichtigsten Spezifikationen zu finden.

[10] `http://www.atmel.com/Images/doc5131.pdf`
[11] `http://www.atmel.com/Images/doc2549.pdf`
[12] `http://www.atmel.com/Images/doc8226.pdf`

Section 1
Introduction

1.1 Summary

ZigBit™ is an ultra-compact, low-power, high-sensitivity 2.4 GHz IEEE 802.15.4/ZigBee® OEM module based on the innovative Atmel's mixed-signal hardware platform. It is designed for wireless sensing, control and data acquisition applications. ZigBit modules eliminate the need for costly and time-consuming RF development, and shortens time to market for a wide range of wireless applications.

Two different versions of 2.4 GHz ZigBit modules are available: ATZB-24-B0 module with balanced RF port for applications where the benefits of PCB or external antenna can be utilized and ATZB-24-A2 module with dual chip antenna satisfying the needs of applications requiring integrated, small-footprint antenna design.

1.2 Applications

ZigBit module is compatible with robust IEEE 802.15.4/ZigBee stack that supports a self-healing, self-organizing mesh network, while optimizing network traffic and minimizing power consumption. Atmel offers two stack configurations: BitCloud and SerialNet. BitCloud is a ZigBee PRO certified software development platform supporting reliable, scalable, and secure wireless applications running on Atmel's ZigBit modules. SerialNet allows programming of the module via serial AT-command interface.

The applications include, but are not limited to:

- **Building automation & monitoring**
  - **Lighting controls**
  - **Wireless smoke and CO detectors**
  - **Structural integrity monitoring**
- **HVAC monitoring & control**
- **Inventory management**
- **Environmental monitoring**
- **Security**
- **Water metering**
- **Industrial monitoring**
  - **Machinery condition and performance monitoring**
  - **Monitoring of plant system parameters such as temperature, pressure, flow, tank level, humidity, vibration, etc.**
- **Automated meter reading (AMR)**

1.3 Key Features

- **Ultra compact size (24 x 13.5 x 2.0 mm for ATZB-24-A2 module and 18.8 x 13.5 x 2.0 mm for ATZB-24-B0 module)**
- **Innovative (patent-pending) balanced dual chip antenna design with antenna gain of approximately 0 dBi (for ATZB-24-A2 version)**
- **High RX sensitivity (-101 dBm)**
- **Outperforming link budget (104 dB)**
- **Up to 3 dBm output power**
- **Very low power consumption:**
  - **< 6 µA in Sleep mode,**
  - **19 mA in RX mode,**
  - **18 mA in TX mode**
- **Ample memory resources (128K bytes of flash memory, 8K bytes RAM, 4K bytes EEPROM)**
- **Wide range of interfaces (both analog and digital):**
  - **9 spare GPIO, 2 spare IRQ lines**
  - **4 ADC lines + 1 line for supply voltage control (up to 9 lines with JTAG disabled)**
  - **UART with CTS/RTS control**
  - **USART**
  - **I^2C**
  - **SPI**
  - **1-Wire**
  - **Up to 30 lines configurable as GPIO**
  - **Capability to write own MAC address into the EEPROM**
  - **Optional antenna reference designs**
  - **IEEE 802.15.4 compliant transceiver**
  - **2.4 GHz ISM band**
  - **BitCloud embedded software, including serial bootloader and AT command set**

1.4 Benefits

- **Small physical footprint and low profile for optimum fit in even the smallest of devices**
- **Best-in-class RF link range**
- **Extended battery life**
- **Easy prototyping with 2-layer PCB**
- **Ample memory for user software application**
- **Mesh networking capability**
- **Easy-to-use low cost Evaluation Kit**
- **Single source of support for HW and SW**
- **Worldwide license-free operation**

1.5 Abbreviations and Acronyms

ADC	Analog-to -Digital Converter
API	Application Programming Interface
DC	Direct Current

Section 2

Zigbit™ Module Overview

2.1 Overview

ZigBit is a low-power, high-sensitivity IEEE 802.15.4/ ZigBee-compliant OEM module. This multi-functional device occupies less than a square inch of space, which is comparable to a typical size of a single chip. Based on a solid combination of Atmel's latest MCU Wireless hardware platform [1], the ZigBit offers superior radio performance, ultra-low power consumption, and exceptional ease of integration.

Figure 2-1. ATZB-24-B0 Block Diagram

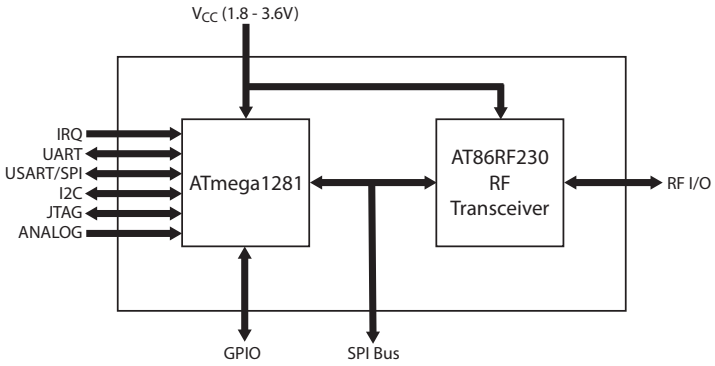
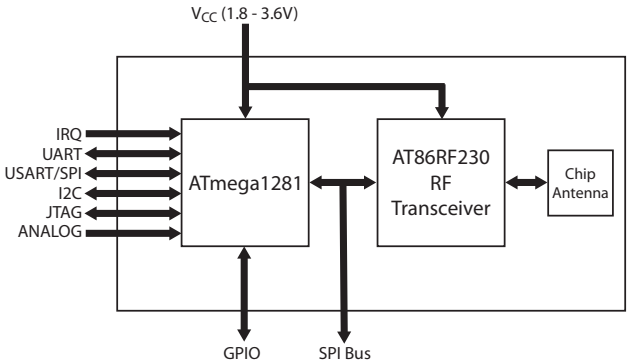


Figure 2-2. ATZB-24-A2 Block Diagram



A.12.4. Zigbit BreakOut Board

Um die Pins des Zigbit Boards zugänglich zu machen, wurde in Zusammenarbeit mit dem IPv6 Labor und dem Elektronik Labor der Beuth Hochschule eine Platine entwickelt, welche in das Gehäuse der 6LoWPAn Steckdose passt. In den folgenden Abschnitten ist zuerst das Layout der Platine zu finden, im zweiten folgenden Abschnitt kann der Schaltplan betrachtet werden.
Das Layout wurde mit der Software *Eagle* erstellt, die Dateien sind auf der CD im Verzeichnis *\Zigbit_Break_Out* abgelegt.

Platinen Layout

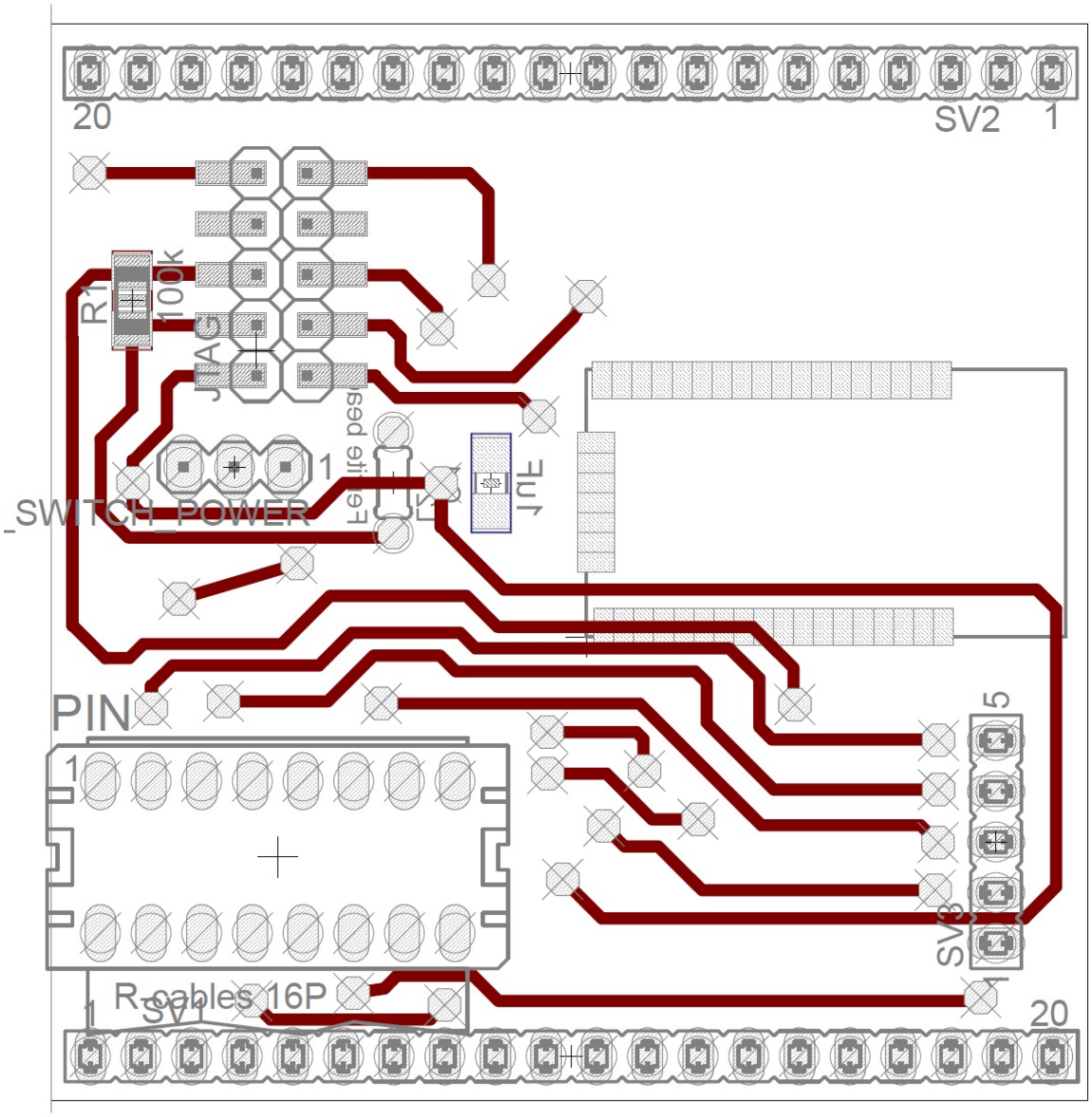


Abbildung A.13.: Zigbit Platinenlayout Vorderseite

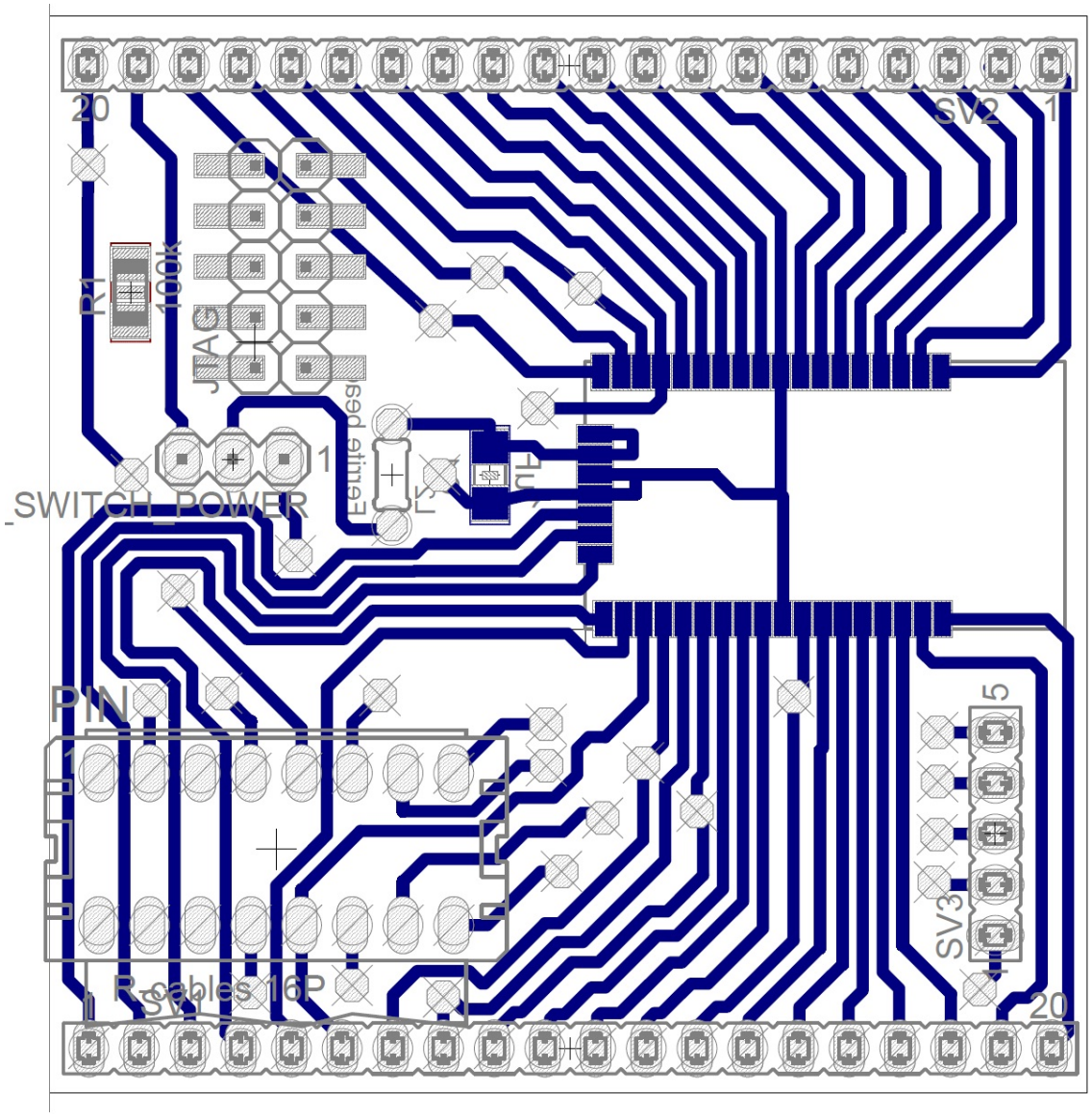


Abbildung A.14.: Zigbit Platinenlayout Rückseite

Schaltplan

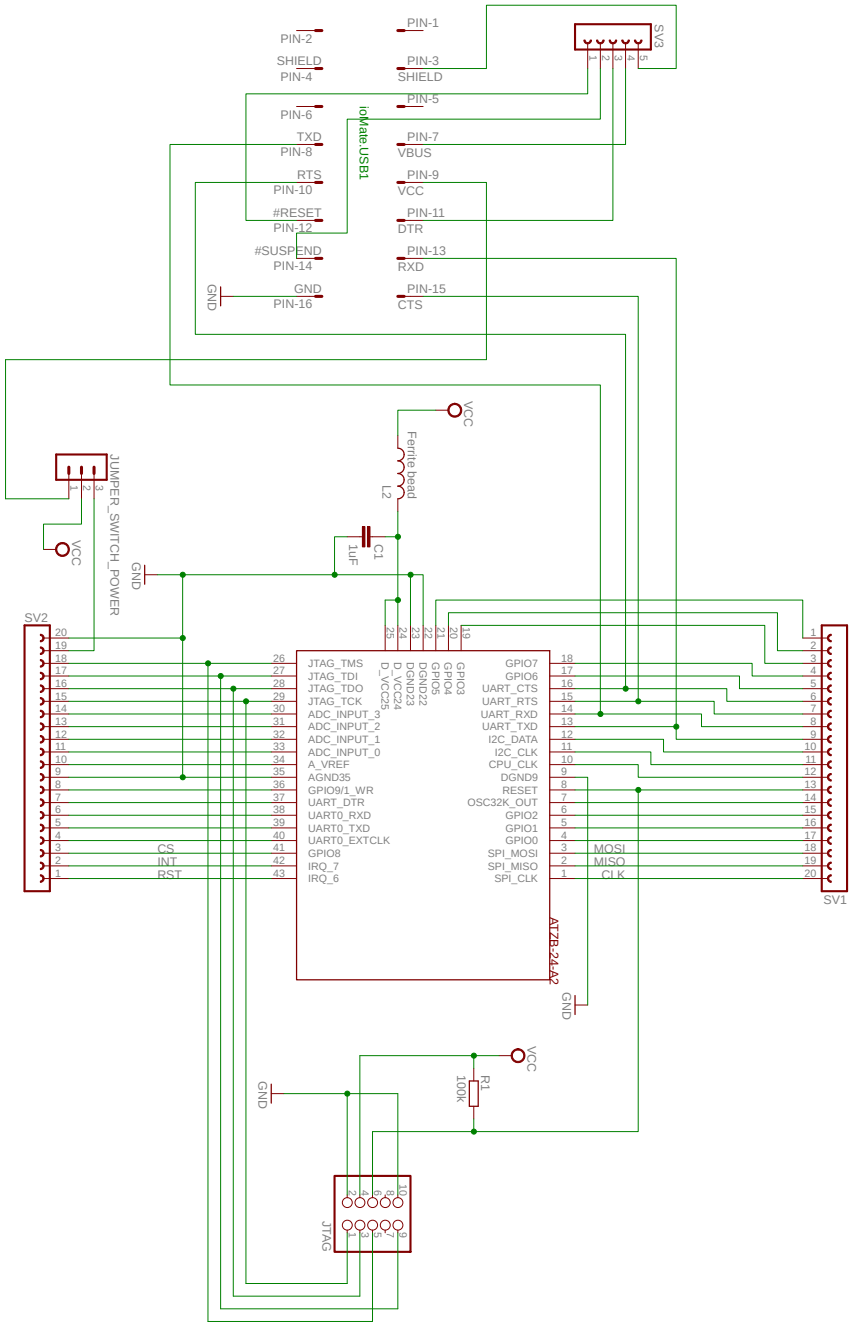


Abbildung A.15.: Zigbit Platine Schaltplan

Anschluß des JTAG Programmieradapters

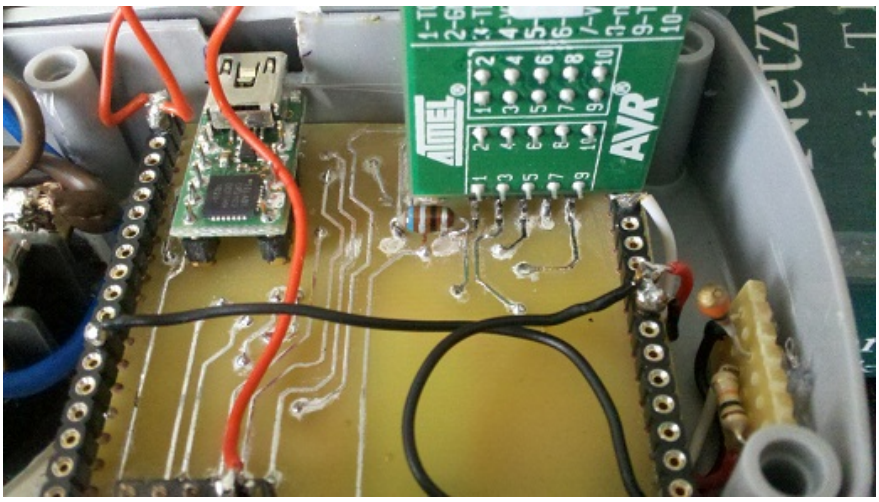


Abbildung A.16.: Anschluss des JTAG Programmieradapters

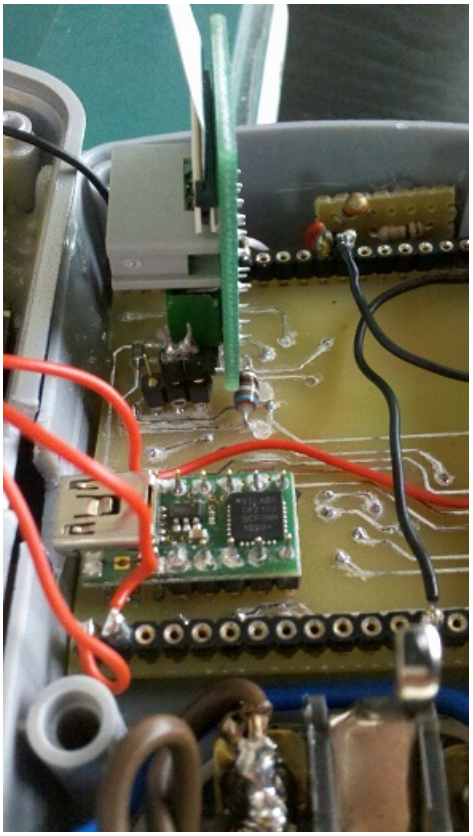


Abbildung A.17.: Anschluss des JTAG Programmieradapters

A.12.5. CP2102

Der verwendete CP2102 Chip der Firma Silabs, wurde als komplettes Modul mit Mini USB Anschluß, passend auf einen DIL IC Sockel, verwendet. Dieses Modul wird unter der Typenbezeichnung IoMate USB1 von dem Hersteller chip45 vertrieben. Es tranformiert außerdem die USB Eingangsspannung von 5V auf 3,3V. Diese steht an Pin VCC zur Verfügung.
Die folgende Seite beinhaltet das Datenblatt des IoMate USB1 Moduls, es ist außerdem auf der CD im Verzeichnis *\Datenblätter\ioMate-USB1* in Datei `iomateusb1.pdf` gespeichert. Des Weiteren ist im gleichen Verzeichnis das vollständige Datenblatt des Silabs Chips CP2102 in der Datei `cp2102.pdf` vorhanden. Beide Datenblätter sind außerdem im Internet downloadbar[13].

[13] Iomate USB1 : `http://download.chip45.com/ioMate-USB1_V2.0_infosheet.pdf` CP2102: `http://www.silabs.com/SupportDocuments/TechnicalDocs/cp2102.pdf`

ioMate-USB1 V2.0 Infosheet

ioMate-USB1 V2.0

Tiny OEM interface module with CP2102 USB to USART converter.

ioMate-USB1 is a universal interface adapter module for connecting microcontrollers, FPGAs, etc. to the USB bus.

It is based on Silicon Laboratories' CP2102 USB-to-UART interface chip, which converts data traffic between USB and UART formats. ioMate.USB1 includes a complete USB 2.0 full-speed function controller, bridge control logic and a UART interface with transmit/receive buffers and handshake signals. A royality-free device driver for PC (Windows 98SE/2000/XP), Macintosh and Linux platforms is available for download at www.chip45.com.

CP2102 Customzing – Furthermore the CP2102 offers an integrated EEPROM for storing USB parameters like VID, PID, product string and serial number, hence providing the possibility to individualize a product based on ioMate.USB1, by customizing these parameters. Silabs offers a free Windows tool, as well as another tool for generating a customized Windows USB driver. By providing such a driver to your customers, your product will be recognized immediately when connected to a PC and will be registered in Windows control panel with it's unique product name. Later your product will be assigned the same COM port number each time it is conneced to the PC, which simplifies access to your product from PC applications.

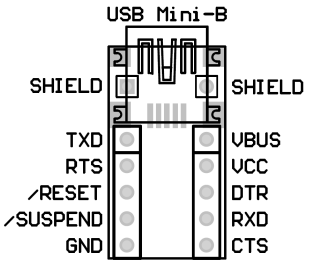
The tools and the corresponding application notes (AN144 and AN220) by Silabs are also available on http://www.chip45.com.

Due to the low price of ioMate-USB1, it is ideally suited as an assembling option for new system designs. Simply add an appropriate IC socket to your application and add USB functionality as required at any time.

Signals – ioMate.USB1 provides an open-drain /RESET signal, which can be used to either force a reset on ioMate.USB1 or drive the reset line of the application. If not used, /RESET can be left open.

The /SUSPEND output can trigger an interrupt and signal a USB bus suspend state. If not used, /SUSPEND can be left open.

New with version 2.0: The VBUS pin still provides the USB bus supply voltage. This pin is now internally connected to the CP2102 supply. Pin VCC now provides the 3.3V output voltage of the CP2102 internal regulator and can source up to 70mA output current for user applications. See datasheet for details current ratings and limiting values!



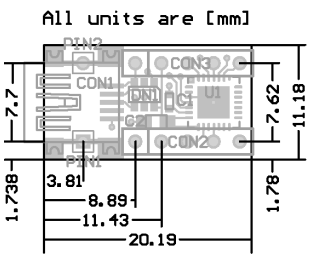
The two SHIELD pins are connected to the USB mini-B connector's shielding. It may used in the application for shielding or grounding purposes.

The remaining five signals provide the UART's receive (RXD) and transmit (TXD) signal, as well as the handshake/modem control sgnals request to send (RTS), clear to send (CTS) and data terminal ready (DTR).

ESD Protection – ioMate.USB1 provides an array of transient voltage suppressor diodes, which provide ESD protection according to EN61000-4 (ESD: Air – 15kV, Contact – 8kV, EFT: 40A – 5/50ns, Surge: 12A, 8/20µs – Level 1 Line-Gnd & Level 2 Line-Line).

***Note:** The UART signals are 3.3V CMOS/TTL compatible signals (5V tolerant), which can be connected to a microcontroller's UART, an FPGA etc. It is not possible to connect these signals directly to RS232 level signals! This might damage the ioMate-USB1 module!*

Formfactor – Dimensions of ioMate-USB1 have been chosen such, that a standard DIL16 pin header fits into the connector pads. This allows to simply plug ioMate-USB1 into a standard DIL14/16 IC socket in the application. The pins are arranged in the standard 2.54mm grid, hence most common pin headers or receptacles will fit into ioMate.USB1. Two (DIL14) or four (DIL16) pins have to be cut off underneath the USB mini-B connector, before assembly. The picture on the right shows the exact dimensions of ioMate.USB1.



A.12.6. Sharp S216S02

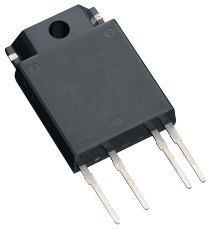
Um die Last zu schalten, wird ein Optokoppler der Firma Sharp mit der Typenbezeichnung S216S02 verwendet. Angesteuert wird dieser mit 1,2V und 20mA Gleichspannung, der daraufhin schaltende Triac erlaubt Wechselspannungen bis zu 600V und Wechselströme bis zu 16A. Auf der folgenden Seite ist das Titelblatt des zugehörigen Datenblatts zu sehen. Es gibt einen Überblick über die Spezifikationen. Das vollständige Datenblatt ist auf der CD im Verzeichnis *Datenblätter\Sharp_S216S02* in der Datei `sharp.pdf` zu finden.

SHARP

S116S02 Series
S216S02 Series

*Non-zero cross type is also available. (**S116S01 Series/ S216S01 Series**)

I_T(rms)≤16A, Zero Cross type
SIP 4pin
Triac output SSR



■ Description

S116S02 Series and **S216S02 Series** Solid State Relays (SSR) are an integration of an infrared emitting diode (IRED), a Phototriac Detector and a main output Triac. These devices are ideally suited for controlling high voltage AC loads with solid state reliability while providing 4.0kV isolation (V_{iso}(rms)) from input to output.

■ Features

1. Output current, I_T(rms)≤16.0A
2. Zero crossing functionary (V_{OX} : MAX. 35V)
3. 4 pin SIP package
4. High repetitive peak off-state voltage
   (V_{DRM} : 600V, **S216S02 Series**)
   (V_{DRM} : 400V, **S116S02 Series**)
5. High isolation voltage between input and output
   (V_{iso}(rms) : 4.0kV)
6. Lead-free terminal components are also available
   (see Model Line-up section in this datasheet)
7. Screw hole for heat sink

■ Agency approvals/Compliance

1. Recognized by UL508 (only for **S116S02 Series**), file No. E94758 (as models No. **S116S02**)
2. Approved by CSA 22.2 No.14 (only for **S116S02 Series**), file No. LR63705 (as models No. **S116S02**)
3. Package resin : UL flammability grade (94V-0)

■ Applications

1. Isolated interface between high voltage AC devices and lower voltage DC control circuitry.
2. Switching motors, fans, heaters, solenoids, and valves.
3. Power control in applications such as lighting and temperature control equipment.

Sheet No.:D4-A02701EN
Date Apr. 28. 2004

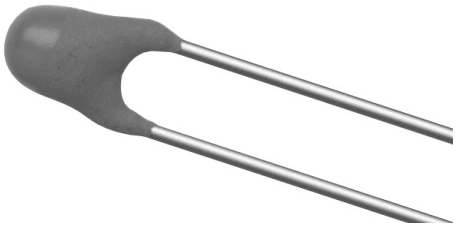
A.12.7. NTC Vishay B400

Das Datenblatt des NTC ist auf der CD unter *Datenblätter\NTC* in `NTC.pdf` abgespeichert.
In den folgenden Seiten sind Auszüge des Datenblatts enthalten, interessant ist hierbei Seite 79,80 des Datenblatts, da hier die Werte für den verwendeten 10kΩ NTC unter der Typbezeichnung 6.103 zu finden sind.

2381 640 3/4/6....

Vishay BCcomponents

NTC Thermistors, Accuracy Line



FEATURES

- Accuracy over a wide temperature range
- High stability over a long life
- Excellent price/performance ratio
- Old part number was 2322 640 3/4/6....
- Component in accordance to RoHS 2002/95/EC and WEEE 2002/96/EC

APPLICATIONS

- Temperature sensing and control

These thermistors have a negative temperature coefficient. The device consists of a chip with two tinned solid copper-plated leads. It is grey lacquered and colour coded, but not insulated.

PACKAGING

The thermistors are packed in bulk or tape on reel; see code numbers and relevant packaging quantities.

QUICK REFERENCE DATA	
PARAMETER	**VALUE**
Resistance value at 25 °C	3.3 Ω to 470 kΩ
Tolerance on R_{25}-value	±2%; ±3%; ±5%; ±10%
Tolerance on $B_{25/85}$-value	±0.5% to ±3%
Maximum dissipation	500 mW
Dissipation factor δ (for information only)	7 mW/K 8.5 mW/K (for 640..338 to 689)
Response time	1.2 s
Thermal time constant τ (for information only)	15 s
Operating temperature range: at zero dissipation; continuously at zero dissipation; for short periods at maximum dissipation (500 mW)	 −40 to +125 °C ≤150 °C 0 to 55 °C
Climatic category	40/125/56
Mass	≈0.3 g

ELECTRICAL DATA AND ORDERING INFORMATION

R_{25} (Ω)	$B_{25/85}$-VALUE	CATALOG NUMBER 2381 640 6....				COLOR CODE (see dimensions drawing and note 1)		
		R_{25} ±2%	R_{25} ±3%	R_{25} ±5%	R_{25} ±10%	I	II	III
3.3	2880 K ±3%	4338	6338	3338	2338	orange	orange	gold
4.7	2880 K ±3%	4478	6478	3478	2478	yellow	violet	gold
6.8	2880 K ±3%	4688	6688	3688	2688	blue	grey	gold
10	2990 K ±3%	4109	6109	3109	2109	brown	black	black
15	3041 K ±3%	4159	6159	3159	2159	brown	green	black
22	3136 K ±3%	4229	6229	3229	2229	red	red	black
33	3390 K ±3%	4339	6339	3339	2339	orange	orange	black
47	3390 K ±3%	4479	6479	3479	2479	yellow	violet	black
68	3390 K ±3%	4689	6689	3689	2689	blue	grey	black
100	3560 K ±0.75%	4101	6101	3101	2101	brown	black	brown
150	3560 K ±0.75%	4151	6151	3151	2151	brown	green	brown
220	3560 K ±0.75%	4221	6221	3221	2221	red	red	brown
330	3560 K ±0.75%	4331	6331	3331	2331	orange	orange	brown
470	3560 K ±0.5%	4471	6471	3471	2471	yellow	violet	brown
680	3560 K ±0.5%	4681	6681	3681	2681	blue	grey	brown
1000	3528 K ±0.5%	4102	6102	3102	2102	brown	black	red
1500	3528 K ±0.5%	4152	6152	3152	2152	brown	green	red

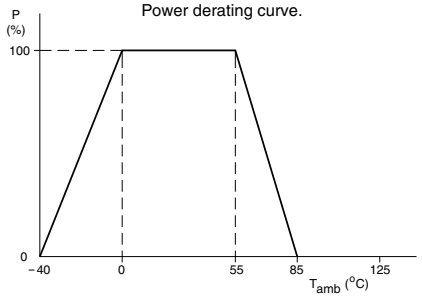
R_{25} (Ω)	$B_{25/85}$-VALUE	CATALOG NUMBER 2381 640 6....				COLOR CODE (see dimensions drawing and note 1)		
		R_{25} ±2%	R_{25} ±3%	R_{25} ±5%	R_{25} ±10%	I	II	III
2000	3528 K ±0.5%	4202	6202	3202	2202	red	black	red
2200	3977 K ±0.75%	4222	6222	3222	2222	red	red	red
2700	3977 K ±0.75%	4272	6272	3272	2272	red	violet	red
3300	3977 K ±0.75%	4332	6332	3332	2332	orange	orange	red
4700	3977 K ±0.75%	4472	6472	3472	2472	yellow	violet	red
6800	3977 K ±0.75%	4682	6682	3682	2682	blue	grey	red
10000	3977 K ±0.75%	4103	6103	3103	2103	brown	black	orange
12000	3740 K ±2%	4123	6123	3123	2123	brown	red	orange
15000	3740 K ±2%	4153	6153	3153	2153	brown	green	orange
22000	3740 K ±2%	4223	6223	3223	2223	red	red	orange
33000	4090 K ±1.5%	4333	6333	3333	2333	orange	orange	orange
47000	4090 K ±1.5%	4473	6473	3473	2473	yellow	violet	orange
68000	4190 K ±1.5%	4683	6683	3683	2683	blue	grey	orange
100000	4190 K ±1.5%	4104	6104	3104	2104	brown	black	yellow
150000	4370 K ±2.5%	4154	6154	3154	2154	brown	green	yellow
220000	4370 K ±2.5%	4224	6224	3224	2224	red	red	yellow
330000	4570 K ±1.5%	4334	6334	3334	2334	orange	orange	yellow
470000	4570 K ±1.5%	4474	6474	3474	2474	yellow	violet	yellow

Notes

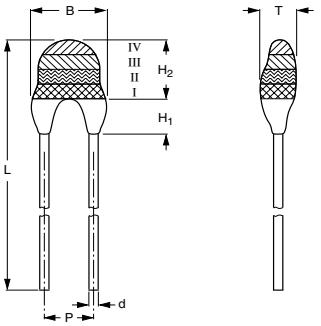
1. Dependent upon R_{25}-tolerance, the band IV is coloured as follows:
   a) for R_{25} ±2%, band IV is coloured red
   b) for R_{25} ±3%, band IV is coloured orange
   c) for R_{25} ±5%, band IV is coloured gold
   d) for R_{25} ±10%, band IV is coloured silver.

DERATING AND TEMPERATURE TOLERANCES

Power derating curve.



DIMENSIONS in millimeters



2381 640 6.338 to 6.474.

PHYSICAL DIMENSIONS FOR RELEVANT TYPE

CODE NUMBER 2381 640	B_{max}	d	H_1		H_2 max	L	P	T_{max}
			MIN.	MAX.				
6.338 to 6.221	5.0	0.6 ±0.06	1.0	4.0	6.0	24 ±1.5	2.54	4.0
6.331 to 6.474	3.3 ±0.5	0.6 ±0.06	–	2.0 ±1.0	6.0	24 ±1.5	2.54	3.0

MARKING

The thermistors are marked with coloured bands; see dimensions drawing and “Electrical data and ordering information”.

MOUNTING

By soldering in any position.

RESISTANCE VALUES AT INTERMEDIATE TEMPERATURES

T_{oper} (°C)	R_T/R_{25}	ΔR DUE TO B-TOLERANCE (%)	TC (%/K)	R_{25} (Ω)		
				2381 640 .....; see note 1 at end of tables		
				6.102	**6.152**	**6.202**
–40	23.3402	1.65	–6.06	23342	35013	46684
–35	17.3347	1.49	–5.84	17336	26004	34672
–30	13.0166	1.34	–5.62	13018	19526	26035
–25	9.8764	1.19	–5.42	9877	14816	19754
–20	7.5682	1.05	–5.23	7569	11353	15138
–15	5.8541	0.92	–5.05	5855	8782	11709
–10	4.5688	0.79	–4.87	4569	6854	9138
–5	3.5961	0.66	–4.71	3596	5395	7193
0	2.8533	0.54	–4.55	2854	4280	5707
5	2.2815	0.43	–4.40	2282	3422	4563
10	1.8376	0.31	–4.26	1838	2457	3675
15	1.4904	0.21	–4.12	1491	2236	2981
20	1.2169	0.10	–3.99	1217	1826	2434
25	1.0000	0.00	–3.87	1000	1500	2000
30	0.8266	0.10	–3.75	826.7	1240	1653
35	0.6873	0.19	–3.63	687.4	1031	1375
40	0.5746	0.28	–3.53	574.6	861.9	1149
45	0.4827	0.37	–3.42	482.7	724.1	965.0
50	0.4073	0.46	–3.32	407.4	611.0	814.7
55	0.3452	0.54	–3.23	345.2	517.8	690.5
60	0.2937	0.62	–3.14	293.7	440.6	587.5
65	0.2508	0.70	–3.05	250.8	376.2	501.7
70	0.2149	0.78	–2.97	214.9	322.4	429.8
75	0.1847	0.85	–2.89	184.8	277.1	369.5
80	0.1593	0.92	–2.81	159.3	238.9	318.6
85	0.1377	0.99	–2.73	137.7	206.6	275.5
90	0.11942	1.06	–2.66	119.4	179.1	238.9
95	0.10380	1.13	–2.59	103.8	155.7	207.6
100	0.09045	1.19	–2.53	90.46	135.7	180.9
105	0.07900	1.25	–2.46	79.00	118.5	158.0
110	0.06915	1.31	–2.40	69.16	103.7	138.3
115	0.06066	1.37	–2.34	60.66	90.99	121.3
120	0.05332	1.43	–2.29	53.32	79.98	106.6
125	0.04696	1.49	–2.23	46.96	70.44	93.9
130	0.04143	1.54	–2.18	41.44	62.15	82.9
135	0.03662	1.60	–2.13	36.63	54.94	73.3
140	0.03243	1.65	–2.08	32.43	48.65	64.9
145	0.02877	1.70	–2.03	28.77	43.16	57.5
150	0.02556	1.75	–1.98	25.56	38.34	51.1

RESISTANCE VALUES AT INTERMEDIATE TEMPERATURES

T_{oper} (°C)	R_T/R_{25}	ΔR DUE TO B-TOLERANCE (%)	TC (%/K)	R_{25} (kΩ)					
				2381 640 .....; see note 1 at end of tables					
				6.222	**6.272**	**6.332**	**6.472**	**6.682**	**6.103**
–40	33.21	2.66	6.57	73.06	89.67	109.6	156.1	225.8	332.1
–35	23.99	2.41	6.36	52.78	64.77	79.17	112.8	163.1	240.0
–30	17.52	2.17	6.15	38.55	47.31	57.82	82.35	119.1	175.2
–25	12.93	1.94	5.95	28.44	34.91	42.67	60.77	87.92	129.3
–20	9.636	1.71	5.76	21.20	26.02	31.80	45.30	65.53	96.36
–15	7.250	1.50	5.58	15.95	19.58	23.93	34.08	49.30	72.50
–10	5.505	1.29	5.40	12.11	14.86	18.16	25.87	37.43	55.05

T_{oper} (°C)	R_T/R_{25}	ΔR DUE TO B-TOLERANCE (%)	TC (%/K)	R_{25} (kΩ)					
				2381 640 .....; see note 1 at end of tables					
				6.222	**6.272**	**6.332**	**6.472**	**6.682**	**6.103**
–5	4.216	1.08	5.24	9.275	11.38	13.91	19.81	28.67	42.16
0	3.255	0.89	5.08	7.162	8.790	10.74	15.30	22.14	32.56
5	2.534	0.70	4.92	5.575	6.842	8.362	11.91	17.23	25.34
10	1.987	0.52	4.78	4.372	5.366	6.558	9.340	13.51	19.87
15	1.570	0.34	4.64	3.454	4.239	5.181	7.378	10.67	15.70
20	1.249	0.17	4.50	2.747	3.372	4.121	5.869	8.492	12.49
25	1.000	0.00	4.37	2.200	2.700	3.300	4.700	6.800	10.00
30	0.8059	0.16	4.25	1.773	2.176	2.660	3.788	5.480	8.059
35	0.6535	0.32	4.13	1.438	1.764	2.156	3.072	4.444	6.535
40	0.5330	0.47	4.02	1.173	1.439	1.759	2.505	3.624	5.330
45	0.4372	0.62	3.91	0.9618	1.180	1.443	2.055	2.972	4.372
50	0.3605	0.77	3.80	0.7932	0.973	1.190	1.694	2.451	3.606
55	0.2989	0.91	3.70	0.6575	0.807	0.9863	1.405	2.032	2.989
60	0.2490	1.05	3.60	0.5478	0.672	0.8217	1.170	1.693	2.490
65	0.2084	1.18	3.51	0.4586	0.562	0.6879	0.9797	1.417	2.084
70	0.1753	1.31	3.42	0.3857	0.473	0.5785	0.8239	1.192	1.753
75	0.1481	1.44	3.33	0.3258	0.399	0.4887	0.6960	1.007	1.481
80	0.1256	1.57	3.25	0.2764	0.339	0.4146	0.5905	0.8544	1.256
85	0.1070	1.69	3.16	0.2355	0.289	0.3532	0.5031	0.7278	1.070
90	0.09154	1.81	3.09	0.2014	0.247	0.3021	0.4303	0.6225	0.9154
95	0.07860	1.93	3.01	0.1729	0.212	0.2594	0.3694	0.5345	0.7860
100	0.06773	2.04	2.94	0.1490	0.182	0.2235	0.3183	0.4607	0.6773
105	0.05858	2.15	2.87	0.1289	0.158	0.1933	0.2753	0.3983	0.5858
110	0.05083	2.26	2.80	0.1118	0.137	0.1677	0.2389	0.3457	0.5083
115	0.04426	2.37	2.73	0.0974	0.1195	0.1461	0.2080	0.3010	0.4426
120	0.03866	2.47	2.67	0.0851	0.1044	0.1276	0.1817	0.2629	0.3866
125	0.03387	2.57	2.61	0.0745	0.0915	0.1118	0.1592	0.2303	0.3387
130	0.02977	2.67	2.55	0.0655	0.0804	0.0982	0.1399	0.2024	0.2977
135	0.02624	2.77	2.49	0.0577	0.0709	0.0866	0.1233	0.1784	0.2624
140	0.02319	2.86	2.43	0.0510	0.0626	0.0765	0.1090	0.1577	0.2319
145	0.02055	2.96	2.38	0.0452	0.0555	0.0678	0.0966	0.1398	0.2055
150	0.01826	3.05	2.33	0.0402	0.0493	0.0603	0.0858	0.1242	0.1826

RESISTANCE VALUES AT INTERMEDIATE TEMPERATURES

T_{oper} (°C)	R_T/R_{25}	ΔR DUE TO B-TOLERANCE (%)	TC (%/K)	R_{25} (kΩ)		
				2381 640 .....; see note 1 at end of tables		
				6.123	**6.153**	**6.223**
–40	25.78	6.81	6.09	309.4	386.8	567.2
–35	19.13	6.16	5.89	229.5	286.9	420.8
–30	14.32	5.53	5.70	171.8	214.8	315.0
–25	10.82	4.93	5.52	129.8	162.3	238.0
–20	8.245	4.35	5.35	98.93	123.7	181.4
–15	6.335	3.80	5.19	76.02	95.03	139.4
–10	4.907	3.26	5.03	58.88	73.60	107.9
–5	3.830	2.74	4.88	45.95	57.44	84.25
0	3.011	2.24	4.73	36.13	45.16	66.24
5	2.384	1.76	4.60	28.60	35.76	52.45
10	1.900	1.30	4.46	22.80	28.50	41.81
15	1.525	0.85	4.34	18.30	22.87	33.55
20	1.231	0.42	4.21	14.77	18.47	27.09
25	1.000	0.00	4.10	12.00	15.00	22.00
30	0.8170	0.41	3.98	9.804	12.26	17.97