HAW HAMBURG

Bachelorarbeit

Andreas Pauli

Energieeffizienz in 6LoWPAN - Implementierung der Generic Header Compression und ihre Einbindung in das IoT Betriebssystem RIOT

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Andreas Pauli

Energieeffizienz in 6LoWPAN - Implementierung der Generic Header Compression und ihre Einbindung in das IoT Betriebssystem RIOT

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Schmidt
Zweitgutachter: Prof. Dr. Martin Becke

Eingereicht am: 1. November 2018

Andreas Pauli

Thema der Arbeit

Energieeffizienz in 6LoWPAN - Implementierung der Generic Header Compression und ihre Einbindung in das IoT Betriebssystem RIOT

Stichworte

IoT, 6LoWPAN, IEEE 802.15.4, Datenkompression, Mikrocontroller

Kurzzusammenfassung

Der RFC 7400 beschreibt die „Generic Header Compression“(GHC), ein Verfahren für die Kompression und Übertragung von Paketen in Funk netzwerken mit dem Protokoll 6LoWPAN. Die Arbeit beschreibt eine Implementierung mit der Sprache C nach den Vorgaben des Standards. Die Anforderungen an die Programmierung von Ressourcen-beschränkten Systemen stehen in Vordergrund bei Entscheidungen zu Software Design.

Andreas Pauli

Title of Thesis

Energy efficiency in 6LoWPAN - Implementation of Generic Header Compression and Integration into the IoT operating system RIOT

Keywords

IoT, 6LoWPAN, IEEE 802.15.4, data compresssion, microcontroller

Abstract

RFC 7400 describes the „Generic Header Compression“(GHC), a method for compressing and transmitting packets over radio networks with the protocol 6LoWPAN. This paper describes a Implementation with language C according to the specifications of the standard. The requirements for the programming of resource-limited systems are at the forefront of software design decisions.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

CPU Central Processing Unit.

IETF Internet Engineering Task Force.

IoT Internet of Things.

IPv6 Internet Protocol version 6.

MSB *Most signifikant bit.*

MTU maximum transmission unit.

WPAN wireless personal area network.

Listings

1 Einleitung

Zum gegenwärtigen Zeitpunkt (2018) ist der Begriff Internet of Things (IoT), auch über die Grenzen der Informatik hinaus ein Bestandteil in Visionen von der näheren Zukunft. Damit verbunden sind Begriffe wie *Industrie 4.0 Gebäude- bzw. Heimautomation*, *Sensornetze* oder auch *intelligente Stromzähler*

1.1 Motivation

Allen gemeinsam ist die Kommunikation über das Internet und damit die Verwendung des *Internet Protocol*. Die Anzahl der dafür benötigten Adressen setzt die Verwendung von Internet Protocol version 6 (IPv6) voraus.

Viele der neuen Anwendungsgebiete, wie zum Beispiel Sensornetzen und mobile Geräte, bringen eine Beschränkung von Ressourcen mit sich. Daraus resultieren reduzierte Bereitstellung von Rechenleistung, Speicher, und Übertragungsraten.
Deshalb ist die Bestrebungen in der drahtlosen Vernetzung die übertragene Datenmenge, und damit auch den Energieverbrauch der Systeme, möglichst gering zu halten.

IEEE 802.15.4 ist der Standard für eine der drahtlosen Netzwerktechniken, die im Bereich der wireless personal area network (WPAN) angesiedelt ist. Den Anforderungen einer reduzierten Sendeleistung nachkommend, bietet sie nur eingeschränkte Datenrate und Reichweite.

Das Protokoll IPv6 over Low power Wireless Personal Area Network (6LoWPAN) ist eine Zwischenschicht, die entwickelt wurde um IPv6 drahtlosen Netzwerken nach IEEE 802.15.4 betreiben zu können. Der Request For Comment 7400 (RFC 7400) der Internet Engineering Task Force (IETF) beschreibt einen Standard zur Kompression der IPv6 Header für das 6LoWPAN Protokoll. [4]

1.2 Ziel

Ziel der Arbeit ist eine Implementierung der Funktionen des beschriebenen Standards. Die Zielplattform Ressourcen-beschränkter Systeme findet neben den Vorgaben des Standards besondere Berücksichtigung bei Konzeption und Entscheidungen zur Implementierung. RIOT OS (RIOT), ein netzwerkfähiges Betriebsystem für Microkontroller verfügt über eine Implementierng von 6LoWPAN. Die Implementierung soll unter RIOT lauffahig sein und in dem Standard entsprechend in 6LoWPAN eingebunden werden.

Dieses Dokument ist in folgende Abschnitte gegliedert:

Basistechnologien und Begriffe Als Grundlage für die kommenden Abschnitte werden Technologien allgemein vorgestellt und Begriffe festgelegt.

Anforderungsanalyse Ausgehend von dem RFC 7400, wird ermittelt welche Anforderungen die Implemetierung erfüllen soll.

Konzept Es wird festgelegt wie die Anforderungen des vorherigen Abschnittes erfüllt und geprüft werden.

Implementierung und Testläufe Die konkrete Implementierung wird hier mit Quellcodebeispielen und Testergebnissen vorgestellt.

2 Basistechnologien und Begriffe

Wie schon in der Einleitung angedeutet wurde, baut das Thema auf einen sehr weiten und vielschichtigem Themenbereich an Software-, Hardware- und Netzwerktechnologien auf.
Der Fokus des RFC 7400 beschränkt sich auf Teilbereiche, die in diesen Technologien eingebettet sind.

Während der Konzeption und Implementierung in den folgenden Abschnitten werden Vorgaben und Beschränkungen dieser Technologien einfließen. Es folgt in diesem Abschnitt die Vorstellung einiger Grundlagen und Begriffserklärungen. Dies soll dazu dienen einen gemeinsamen Ausgangspunkt zu schaffen und Missverständnisse zu vermeiden.

2.1 IoT

Das Internet of Things ist eine begriffliche Erweiterung des „klassischen“ Internet. Dies bestand viele Jahre fast ausschließlich aus Servern, Workstations und Geräten die die Infrastruktur bilden zum Beispiel Switches, Router. Bei einer Workstation galt die Beziehung dass an jeden Host ein Benutzer sitzt.
Die Erweiterung „of Things“ wurde eingeführt, da eine große Anzahl an eigenständigen Geräte hinzugekommen ist bzw. noch kommen wird.

In dieser Arbeit wird der Begriff IoT gebraucht um Geräte zu beschreiben die netzwerkfähige Schnittstellen besitzen und über das `Internet Protocol IP` kommunizieren.[18]

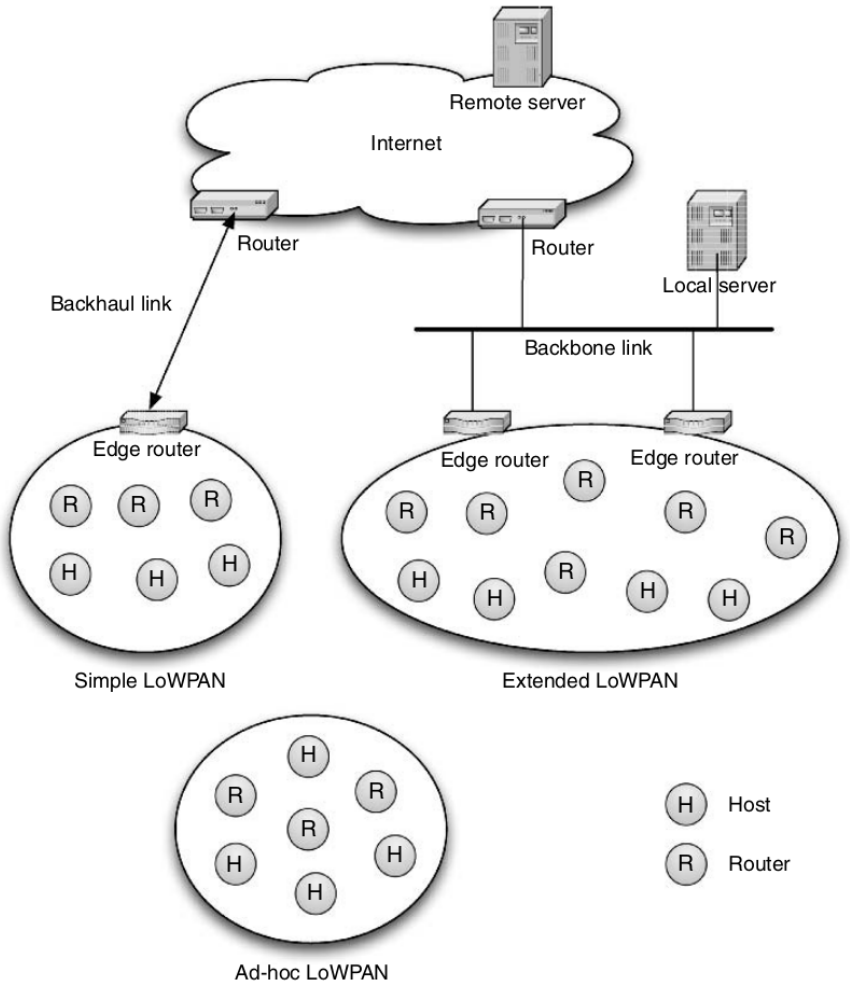

Abbildung 2.1: 6LoWPAN als Komponente des IoT und als Insel,Quelle: [18, Seite 14]

2.2 Ressourcen-beschränkte Systeme

2.2.1 Mikrocontroller und MCU

Moderne Central Processing Units (CPUs) sind über die Jahre leistungsfähiger geworden, was eine Zeitlang auch den Bedarf an elektrischer Leistung hat ansteigen lassen. Getrieben durch das allgemein gesteigerte Bewusstsein zur Effizienz ist es der Industrie gelungen den elektrischen Leistungsbedarf in Relation zur Rechenleistung in den vergangen Jahren zu verringern.

In den Anwendungsbereichen für Ressourcen-beschränkte Systeme sind aber die Verbrauchszahlen selbst der effizientesten CPUs, die in Notebooks eingesetzt werden, weit über dem was hier zur Verfügung gestellt werden kann.
Darüber hinaus sind auch dem physikalischen Volumen starke Grenzen gesetzt.

Microcontroller Units (MCUs) existieren seit den 1970'ger Jahren. Es sind auch heute noch System im Einsatz und auch erhältlich, bei denen die Registerbreite 8 bit umfasst. Diese haben meist kleineren Speicher. Sie sind daher für einen komplexen Netzwerkstack meist nicht geeignet. Moderne MCUs (PSP430, CC2xx, ARM Cortex) haben 16 oder auch 32 bit.

Architektur (siehe Abbildung 2.2) auf Seite 6

- Umfasst CPU, RAM, NVRAM, Power Management
- Im Gegensatz zu CPU's sind bei MCU's Adress- und Datenbus von außen nicht zugreifbar.
- GP(IO), Timer, (meist serielle) IO, SPI, I2C, UART, ADC, DAC, PWM.

Wenn alle diese Komponenten auf einem Chip integriert sind, wird dies als *System on Chip* SoC bezeichnet.

Beispielhaft ist in Abbildung 2.2 auf Seite 6 Eine Blockübersicht einer MCU aus der atmega8 Serie.

MCUs sind die primäre Plattform für die Implementierung der Generic header compression (GHC). Daher werden bei den Vorüberlegungen zur Implementierung neben den Spezifikationen des RFC 7400 die Vorgaben bzw. „Best Practices" der Programmierung von MCUs einen großen Einfluss haben.[21]

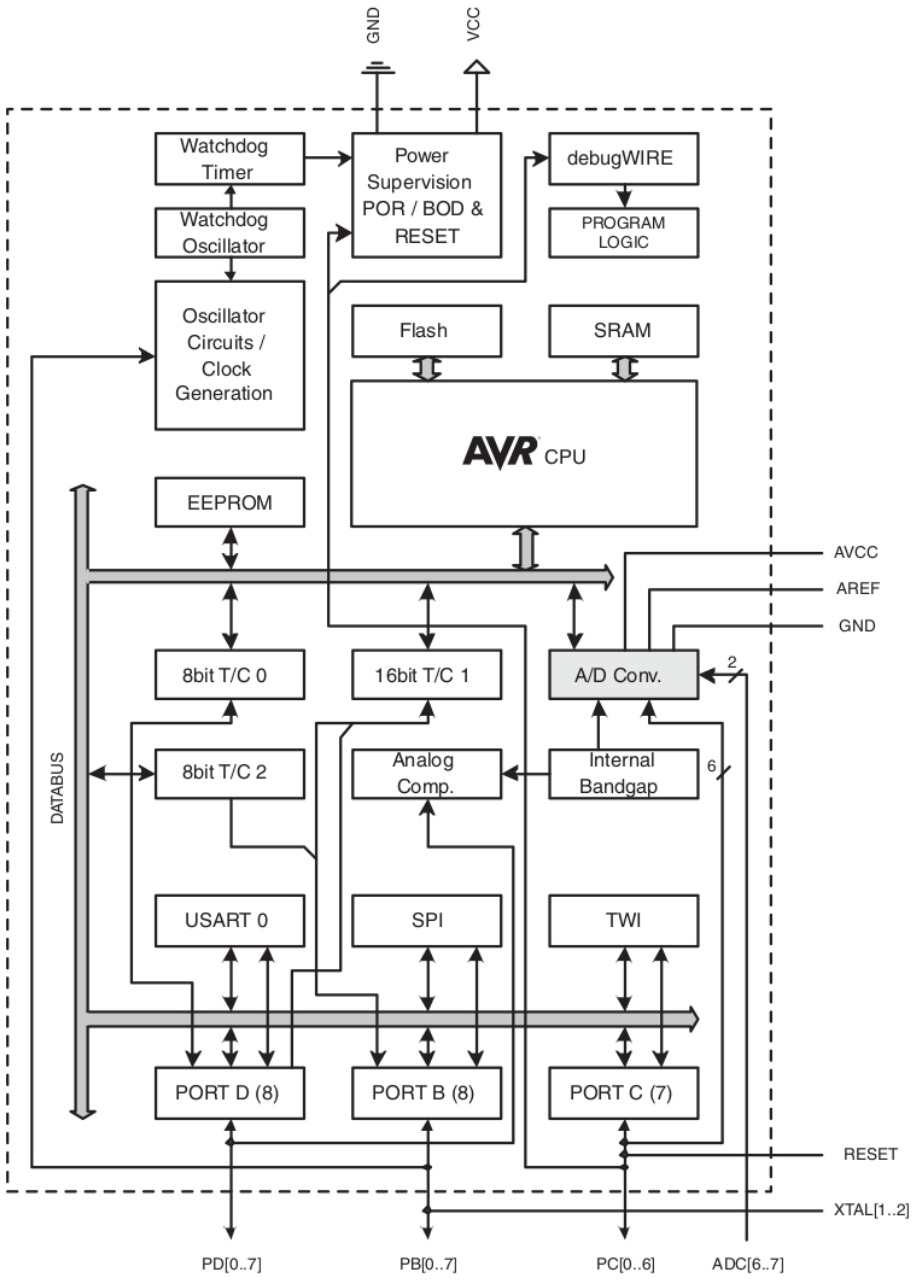

Abbildung 2.2: Atmel-8271J-AVR- ATmega-Datasheet 11/2015[2, Seite 7]

2.2.2 WPAN, LoWPAN und IEEE 802.15.4

In den vergangenen Jahren wird das (im Volksmund) Wi-Fi oder WLAN genannte drahtlose Netzwerk als ein beinahe allgegenwärtiger Ersatz für das Kabel-basierte Ethernet eingesetzt. Der Standard dieser Technik ist unter der Bezeichnung IEEE 802.11 öffentlich verfügbar. Die Spezifikationen werden ständig erweitert und die Übertragungsrate liegen fast gleichauf mit Ethernet (IEEE 802.1).

In Ressourcen-beschränkten Netzwerk ist eine hohe Bandbreite zwar wünschenswert, hat keine Priorität. In den Vordergrund tritt hier die Einsparung von Sendeenergie und damit auch des Energieverbrauchs.[5]

IEEE 802.15.4 gehört zu Gruppe der IEEE 802.15 in der diverse drahtlose Netzwerke standardisiert sind.

2.3 Datenkompression

Allgemein ist Datenübertragung immer eine Kommunikation zwischen einem Sender und Empfänger oder auch einem Schreiber und Leser. Die Informationen müssen strukturiert sein, um korrekt interpretiert zu werden. Strukturen beinhalten aber sehr häufig Redundanzen und belegen mehr Platz als es notwendig wäre.
In anderen Fällen kann es sein dass die Informationsmenge größer ist als es in einem gegebenen Szenario notwendig ist.
Mit Datenkompression wird versucht gefundene Redundanzen einzusparen oder unnötige Informationen zu filtern.

Eine wichtige Maßzahl ist die Kompressionsrate. Mathematisch ist sie das Verhältnis zwischen komprimierter und originaler Größe. Ein Datenblock mit 1280 B hat nach der Kompression noch 320 B. Dann ist der Kompressionsfaktor $1280/320 = 4$ oder auch als Verhältnis ausgedrückt $4 : 1$

2.3.1 Welche Vorteile resultieren aus der Datenkompression

Da bei gegebener Datenrate eines Mediums die Übertragungszeit proportional von der übertragenen Datenmenge abhängig ist, kann Kompression die Zeiten herabsetzten.
Dem gegenüber steht, dass die Kompression und Dekompression von einem Prozessor

durchgeführt werden muss. Dies zieht einerseits zusätzlich Latenz für die Rechenzeit, und in Folge auch zusätzlichen Energieverbrauch nach sich.

Schonung der gemeinsamen Bandbreite In drahtlosen Netzwerken, die mit Zeitmultiplex arbeiten, sind die Zugriffe der Teilnehmer nicht unabhängig, wie z. B. in einem Netzwerk in dem die Hosts mit Ethernet über einen Switch verbunden sind. Es kann ausschließlich ein Gerät auf dem gemeinsam genutzten Kanal senden. Die gemeinsame Nutzung ist also, wie auch auf klassischen Ethernet Installationen mit „Yellow cable“nicht frei von Kollisionen.
Daher ist es, nicht nur für den individuellen Knoten, sondern auch für die gesamte Gruppe an Teilnehmern im lokalen Netz, ein Vorteil wenn die Belegung des gemeinsamen Mediums so kurz wie möglich ist.

Reduktion der Sendeenergie Vorausgesetzt ein Knoten in einem Funk-basierten Netzwerk kann in Phasen in denen keine Daten übertragen werden den Energieverbrauch des Transmitters signifikant reduzieren kann. Unter dieser Bedingung bewirkt eine Verkürzung der Übertragungszeiten eine Verringerung der benötigten Energie über einem längeren Intervall.

2.3.2 Kategorien von Kompressionsverfahren

Ein sehr wichtiges Unterscheidungsmerkmal zwischen den Verfahren zur Datenkompression ist die Einordnung in *verlustfreie* und *verlustbehaftet*.

Verlustbehaftete Verfahren, auch als Daten*reduktion* bezeichnet, entfernen Informationen so, dass *subjektiv* kein Unterschied zum Original festgestellt werden kann. Der Ausdruck Reduktion macht hier deutlich, dass nicht mehr alle Ursprungsdaten zur Verfügung stehen. Bekannte Beispiele findet man in Audio- (mp3,opus) oder Videoformaten (JPEG,MPEG,H264,..).

Verlustfreie Verfahren haben die Eigenschaft, dass nach einer Encodierung und anschließender Dekodierung die Ausgangsdaten identisch wieder hergestellt werden.

Lauflängenkodierung Das Auftreten von Sequenzen identischer Symbole wird codiert, indem das Symbol und numerisch die Anzahl der Wiederholungen gespeichert wird.

Statistisch Diese Verfahren nutzen die Verteilung von Häufigkeiten der Symbole.

Wörterbuch-basiert Diese Verfahren nutzen mögliche Redundanzen die durch das wiederholte Auftreten von identischen Symbolsequenzen entstehen.

Wird eine Sequenz gefunden, die identisch mit einer Sequenz im Wörterbuch ist, wird die Übereinstimmung mit der Startposition und Länge, als Referenz auf den Wörterbucheintrag, kodiert.
Wenn dieser Code weniger Speicher beansprucht als die Sequenz selbst, dann ist die Differenz zwischen diesen beiden Größen eingespart worden und eine Kompression wurde erreicht. Der eigentliche Kern dieses Verfahrens ist damit ein Suchalgorithmus.

Bei der Dekompression oder Dekodierung kann mit Hilfe der gespeicherten Referenz die Symbolkette wieder aus dem Wörterbuch kopiert werden. Damit sind die Originaldaten identisch wiederhergestellt.

[16, 17, 8]

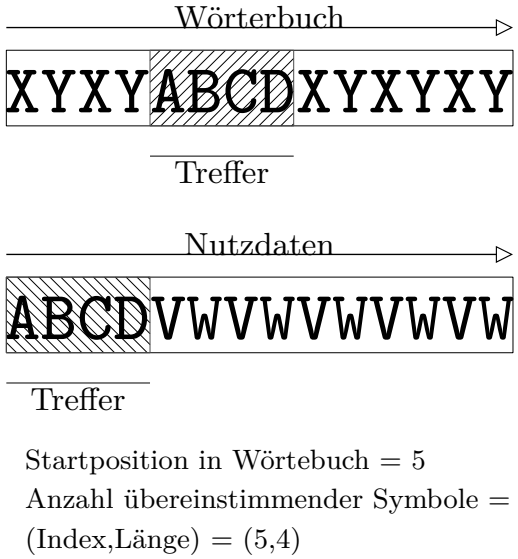

Abbildung 2.3: Codierung in Wörterbuch basierter Kompression

2.4 6LoWPAN

Die Bezeichnung 6LoWPAN steht für IPv6 over Low power Wireless Personal Area Network. Dies ist eine Spezifikation, die in dem RFC4944 mit dem Titel „Transmission of IPv6 Packets over IEEE 802.15.4 Networks“ veröffentlicht wurde. [11] IPv6 setzt eine maximum transmission unit (MTU) von 1280 B voraus. Die von IEEE 802.15.4 verfügbare MTU beträgt 127 B. Damit IPv6 trotzdem genutzt werden kann liegt 6LoWPAN als Übersetzungsschicht zwischen IPv6 und dem MAC Layer und tranformiert die Größen so dass die umgebenden Layer unverändert arbeiten können.

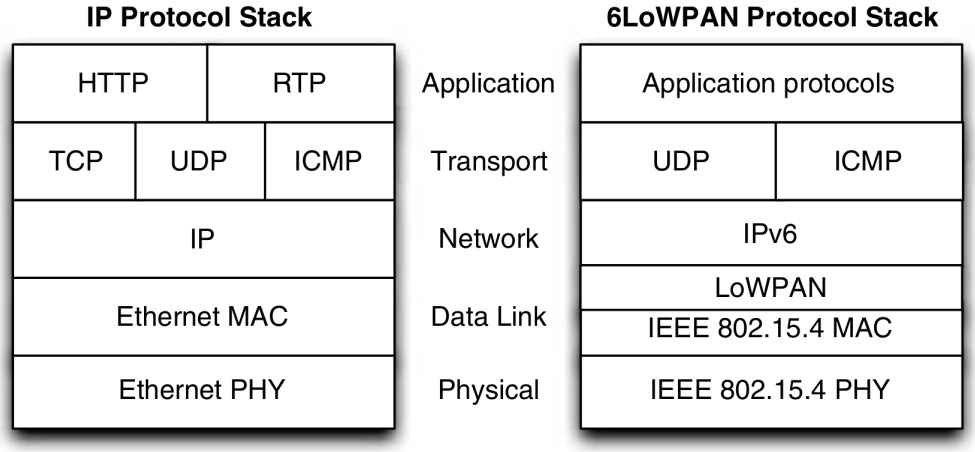

Abbildung 2.4: 6LoWPAN Position im OSI SchichtenmodellQuelle: [18]

2.5 Das IoT Betriebssystem RIOT

RIOT ist ein offenes IoT Betriebssystem für den Einsatz auf MCUs. Zu Beginn ein kurzer Überblick über die Historie:

2008 Projekt Wurzeln. Der Keim für RIOT war FeuerWare, ein Betriebssystem für drahtlose Sensornetzwerke. Es war Teil des FeuerWhere-Projekts, bei dem Feuerwehrleute überwacht werden sollten. Wichtige Konstruktionsziele waren Zuverlässigkeit und Echtzeit-Garantien.

2010 In Richtung Internet-Compliance. Um die Modularität zu erhöhen und neue IETF-Protokolle zu integrieren, wurde μkleos aus dem originalen FeuerWare-Repository gespalten. Unterstützung für 6LoWPAN, RPL und TCP wurde in den folgenden Jahren integriert.

2013 RIOT geht an die Öffentlichkeit. RIOT ist der direkte Nachfolger von μkleos. Wir haben uns für ein Rebranding entschieden, um Probleme mit der Rechtschreibung und Aussprache des Betriebssystemnamens zu vermeiden. Wir fördern ausdrücklich RIOT einer größeren Gemeinschaft zugänglich zu machen.

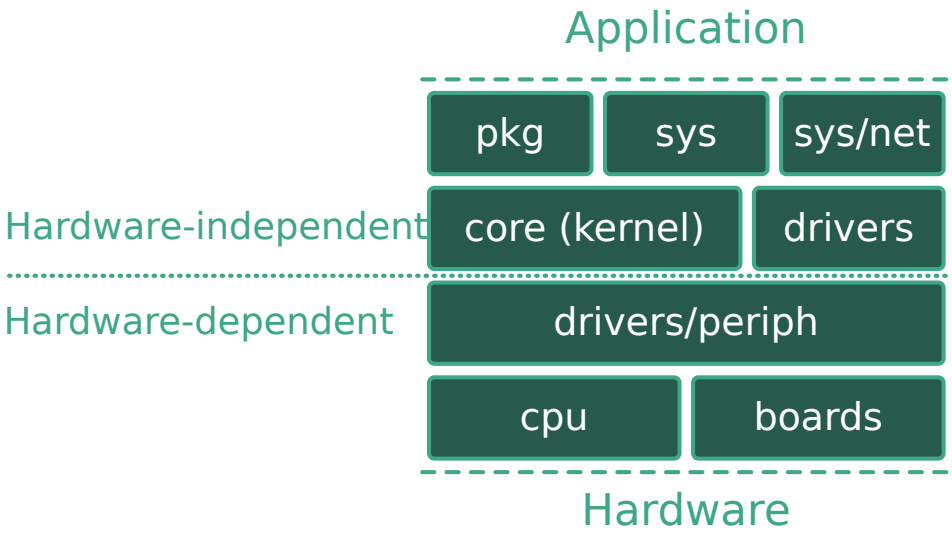

Abbildung 2.5: RIOT Architektur

RIOT ist ein minimalistisches Betriebssystem mit geringem Speicherbedarf.

- Unterstützung verschiedener MCU Platformen wie AVR atmega, MSP430, CC2xxx, ARM7, ARM Cortex. Trennung zwischen hardwareabhängigen und -unabhängigem Code. Siehe Abbildung 2.5
- Bereitstellung essentieller Kernel Abstraktionen wie IPC, Threads, Gerätetreiber und Netzwerk.
- IPv6 und 6LoWPAN sind integriert.
- Modularer Aufbau. Dadurch gute Entkopplung bei der Entwicklung. Siehe Abbildung 2.5

- Module z.B. Netzwerktreiber werden in Threads ausgeführt.
- Eigenes Buildsystem auf Basis von GNU make.
- Unterstützt die gängigen C/C++ Entwicklungsumgebungen gcc und clang.

[9],[10]

2.6 Verwendete Werkzeuge und Hilfsmittel

Als Programmiersprache für die Implementierung wird C in der Version C11 festgelegt. Das maßgebliche Kriterium sind die MCUs als Zielplattform. C bietet einen besseren Abstraktionsgrad als Assembler, Hardware-nahe Programmierung zu. Mit dem gcc existiert für die verschiedenen, in MCUs integrierten, CPU Architekturen durchgehende Standardbibliotheken und Compiler.[21] Bezüglich der Programmierung auf RIOT ist ebenfalls C/C++ vorgegeben.

- Betriebssystem: `Debian GNU/Linux 9.5 (stretch)`
- Kernel: `Linux 4.18.0-2-amd64 #1 SMP Debian 4.18.10-2 (2018-10-07) x86_64 GNU/Linux`
- RIOT: `2018.07-branch`[15]
- `gcc (Debian 8.2.0-8) 8.2.0`
- `arm-non-eabi-gcc version 7.3.1 20180622 (release)`
- `avr-gcc version 5.4.0 (GCC)`
- `GNU gdb (Debian 8.1-4+b1) 8.1`
- `Wireshark 2.6.4 (Git v2.6.4 packaged as 2.6.4-1)`
- `Python 2.7.15+`
- `ruby 1.9.3p551 (2014-11-13) [x86_64-linux]`
- `doxygen 1.8.13`

2.7 Verwandte Arbeiten

Bei der Recherche nach bereits existierenden Implementierungen der GHC sind die Repositorys der folgenden Opensource Betriebssysteme betrachtet worden:

Contiki[7], TinyOS[19], Linux[20], FreeBSD[13]

Keines hat derzeit eine Implementierung von GHC. Bei Linux findet man im Verzeichnis einige Dateien mit Bezug zu GHC. Der Inhalt beschränkt sich auf die Schnittstellen zum Laden als Kernelmodule und Konstanten für die Erkennung von GHC Formaten. Code für die Bearbeitung von Datenpaketen ist nicht zu finden.

2.7.1 Referenztool in Ruby

Von dem Autoren des RFC 7400, Carsten Bormann, ist freundlicherweise ein in Ruby realisierts Programm zur Verfügung gestellt worden, mit dem zumindest ein Teil der Funktionen vergleichen werden kann.

Das Programm ist noch mit Ruby in der Version 1.9 entwickelt worden. Mit der Einführung von Ruby Version 2.0 ist die interne Verarbeitung von Strings vollständig auf UTF-8 umgestellt worden. Aktuell ist als Standardinstallation Ruby 2.3 nur verfügbar.

Eine Anpassung des Programmcodes zur Verwendung der neuen API, ist aus Mangel an Detailkenntnissen von Ruby verworfen worden. Eine Lösung besteht in der Erzeugung einer Ruby Laufzeitumgebung in einer Version 1.9.X. Diese ist auch in der Liste der verwendeten Werkzeuge eingetragen.

3 Anforderungsanalyse

Im Folgenden werden die Anforderungen ermittelt, die durch die Implementierung erfüllt werden sollen. Der RFC 7400 [4], welcher die GHC spezifiziert, ist das zentrale Dokument und Ausgangspunkt für die Analyse.

Die GHC ist in einem Netzwerkprotokoll eingebettet und es ist zwangsläufig, dass verschiedene Implementierungen direkt miteinander kommunizieren. Damit diese interoperabel sind müssen die Spezifikationen genau eingehalten werden.

Zur Unterstützung des Verständnisses und Vermeidung von Missverständnissen beim Lesen von Request for comment (RFC)s dient eine, ebenfalls als RFC verfasste, Spezifikation. Dies ist der RFC 2119 mit dem Titel *Key words for use in RFCs to Indicate Requirement Levels.* [6]

Üblicherweise enthalten RFCs Verweise auf weitere Standards oder Normierungen, sogenannte *normative Referenzen.* Auf diese wird ebenfalls eingegangen werden, wenn es sich auf die Implementierung auswirkt.

Der RFC 7400 ist in drei wesentliche Teile gegliedert, für die es hier einen kurzen Überblick gibt. In den folgenden Sektionen werden diese, nach einer Einführung zu Lempel Ziv 1977 Kompression (LZ77), dann näher betrachtet.

Der GHC Codec In den Abschnitten *1.2. Compression Approach*,*1.4. Notation* und *2. 6LoWPAN-GHC* werden die Operationen beschrieben mit denen ein Decoder die originalen Nutzdaten wieder hergestellt. Der Encoder ist nicht explizit beschrieben.

Integration in den Netzwerkstack In den Abschnitten *3.1. Compressing Payloads (UDP and ICMPv6)* und *3.2. Compressing Extension Headers* wird für zwei ausgewählte Protokolle und IPv6 extension header festgelegt wie die Einbettung innerhalb der 6LoWPAN Pakete erfolgt.

Signalisierung GHC ist optional und die Fähigkeit entsprechende Paket zu empfangen muss anderen Knoten signalisiert werden. Die Abschnitte *3.3. Indicating GHC Capability* und *3.4. Using the 6CIO* legen diesen Teil fest.

3.1 LZ77

Im Standard wird die GHC als LZ77-ähnlich deklariert. Es folgt eine Erläuterung der wesentlichen Eigenschaften. In einem späteren Unterkapitel werden, für GHC spezifische, Unterschiede zu LZ77 erörtert.

Das Verfahren geht zurück auf eine Arbeit der Mathematiker *Abraham Lempel* und Jacob Ziv. Diese wurde 1977 im Journal *IEEE Transactions on Information Theory 23*[22] vorgestellt. Die Bezeichnung LZ77 ist eine Kombination der Initialen der beiden Nachnamen, sowie die Jahreszahl der Veröffentlichung.

LZ77 gehört zur Gruppe der *verlustfreien* Verfahren und ist Wörterbuch-basiert. Siehe auch Abschnitt 2.3.2 auf Seite 9.

Die Idee bei LZ77 ist die Benutzung eines *dynamischen* Wörterbuches. Dies ist zu Beginn leer und wird schrittweise mit dem bereits codierten Daten gefüllt.
Das Wörterbuch erhält in diesem Zusammenhang die Bezeichnung *Suchpuffer*. Der noch zu codierende Teil ist der *Vorschaupuffer*. Siehe Abbildung 3.1.

Die längste Übereinstimmung eines Präfixes des Vorschaupuffers mit einem Teil des Suchpuffers wird ermittelt. Der Treffer wird dann mit einem 3-Tupel codiert. Dieser besteht aus der Startposition und Länge des Treffers und dem ersten Symbol im Vorschaupuffer, das keine Übereinstimmung zeigte.
Die Startposition eines Treffers wird relativ zum Anfang des Vorschaupuffers nach links aufsteigend angegeben.

Im gezeigten Beispiel Abbildung 3.1 würde der Treffer mit (7,4,V) codiert.

Anschließend würde die Trennmarkierung hinter die gerade codierte Sequenz (hier 5 Felder weiter nach rechts) positioniert werden.

Meistens werden auch, zugunsten des Aufwandes für die Trefferermittlung und der notwendigen Bitbreite für Index und Länge, Fenster für die beiden Puffer festgelegt.

Damit verbunden ist ein Abwägung, weil kleinere Fenster die Chance auf Treffer und deren Länge verringern.

Für LZ77 findet sich ein Spezialfall bei dem sich die Sequenzen im Such- und Vorschaupuffer überlappen. Dieser ist in Abbildung 3.2 Seite 17 gezeigt. Die Codierung in diesem Beispiel wäre (3,6,V). [16, Seite 171]

3.2 6LoWPAN-GHC

Der RFC beschreibt lediglich den Decoder, als eine Maschine die definierte Bytecodes ausführt, um die ursprünglichen Nutzdaten wieder herzustellen.
Den Encoder ist nicht explizit beschrieben und dessen Eigenschaften müssen abgeleitet werden. Vorerst lässt sich festlegen, dass dieser aus Nutzdaten einen Bytecode erzeugt (compiliert) mit dem ein valider Decoder die originalen Nutzdaten wieder herstellt.

Die Definition als Bytecode legt 8Bit Breite (Oktetts) fest. Dies gilt ebenso für die Nutzdaten, die teilweise literal mit eingebettet werden.
Eine Berücksichtigung bezüglich CPUs mit *little-* oder *big-endian* Architekturen ist nur soweit notwendig, dass in der Implementierung Datentypen mit 8 Bit verwandt werden.

Für eine spätere Festlegung der Datentypen ist festzustellen, ob eine Interpretation des *Most signifikant bit* (MSB) als Vorzeichenbit notwendig ist.
Der Bytecode repräsentiert eine Struktur aus Befehlscode und ggf. Parametern. Und die Nutzdaten stellen an dieser Stelle ein Transportmedium dar, das von verschieden Protokollen genutzt wird. Eine spezifische Typenzuordnung ist nicht möglich.
Damit ist auch keine Behandlung des MSB als Vorzeichenbit gegeben.

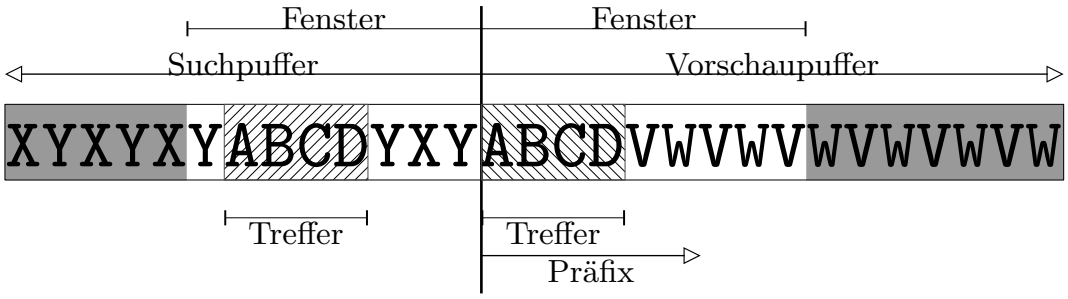

Abbildung 3.1: Vorschau- und Suchpuffer bei LZ77

Sowohl Encoder (Kompressor) als auch Decoder (Dekompressor) sind Funktionen mit einem Quell- und einem Zielpuffer.

Decoder Der Decoder verarbeitet eine geordnete Liste aus Codebytes und Nutzdaten die literal kopiert werden müssen. Es existiert keine Möglichkeit beide Aufgrund ihres Wertes zu unterscheiden. Es muss daher so etwas wie einen Zustand geben, von dem abhängt, wie ein gelesenes Oktett interpretiert wird.

Für den Decoder gilt die Zuordnung:

Quellpuffer Bytecode mit den komprimierten Nutzdaten.

Zielpuffer Wieder herzustellende Nutzdaten mit vorangestellten initialen Wörterbuch.

Des Weiteren existieren zwei Register *na* und *sa* für die Akkumulation des Offsets von zwei Parametern.

Im Folgenden werden Bezeichner für die *Codebytes* vergeben, die auf Seite 4, Tabelle 1 des RFC definiert sind. Im weiteren Verlauf der Arbeit dienen diese als Bezug. Die Terminologie für die *Codebytes* ist in der *Section 1.2* des RFC erläutert.

`GHC_COPY` 0b0kkkkkkk
: Kopiere die folgenden k Oktetts des Eingangspuffers in den Ausgangspuffer. Der Opcode belegt Bit 7 und hat den Wert 0. Die restlichen Bits (6–0) geben die Anzahl der zu kopierenden Oktetts an. Mit diesen sieben Bit könnten 127 Oktetts kopiert werden, das Maximum ist per Definition auf 95 beschränkt.

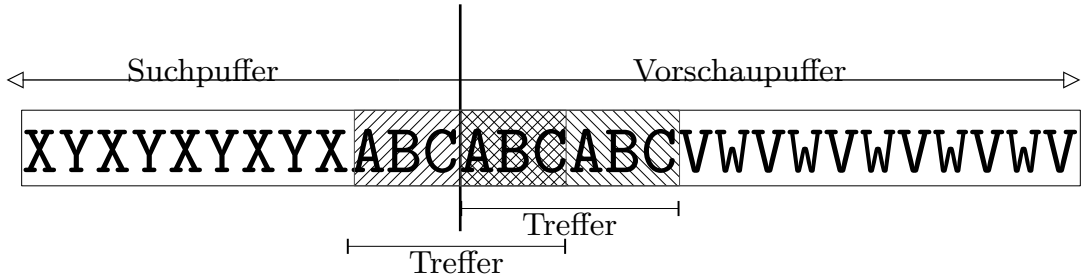

Abbildung 3.2: Spezialfall bei LZ77

Dies ist das einzige Codebyte, das mit literalen Nutzdaten im Eingangspuffer einhergeht.

`GHC_ZEROS` 0b1000nnnn
: Schreibe $n + 2$ Oktetts mit dem Wert 0 in den Zielpuffer. Der Opcode belegt die Bits 7–4. Die Anzahl der anzufügenden Oktetts wird mit den Bits (3–0) dargestellt und deckt den Bereich $[0, 15]$ ab. Mit dem vorgegebenen Offset von 2 ändert sich dieser zu $[2, 17]$.

`GHC_BREF` 0b11nnnkkk
: Bilde aus den Parametern *nnn* und *kkk* und den Registern *sa na* die Werte n und s. Dann kopiere n Oktetts ab der relativen Position s aus dem Wörterbuch in den Zielpuffer.
Als Nebeneffekt werden die Register *na* und *sa* auf 0 zurückgesetzt.
Diese Operation bildet das Verhalten des LZ77 Verfahrens ab.

`GHC_STOP` 0b10010000
: Beendet die Decodierung.

`GHC_EXT` 0b101nssss
: Akkumuliere die zwei globale Register *na* und *sa*, die zur Berechnung von Position und Länge der Wörterbuchreferenz in `GHC_BREF` herangezogen werden.
Der Opcode belegt die Bits (7–5).
Die Werte sind aber immer positiv. Für beide Register ist kein oberes Limit angegeben. Es ist nicht limitiert, wie oft diese Aktion aufgerufen werden darf bevor `GHC_BREF` die Werte einliest und, da diese auch nicht als Bytecode übertragen werden, nicht automatisch auf 8Bit beschränkt. Um später über Datentypen dieser Registervariablen entscheiden zu können gibt es die Möglichkeit einer Abschätzung. Diese muss nur so präzise sein um eine Entscheidung für 8, 16 oder 32 bit zu decken.
6LoWPAN sichert dem darüber liegenden IPv6 eine MTU von 1280 B zu. Die Nutzdaten können also vor der Codierung nicht größer gewesen sein, wobei noch der IPv6 Header abzuziehen wäre. Der maximale Index für die Rückreferenz kann nicht größer als diese Größe plus 48 B für das vordefinierte Wörterbuch sein. Ebenso kann die Länge der Sequenz im Wörterbuch nicht größer gewesen sein, genauer betrachtet sogar nur die Hälfte. Damit sind Datentypen mit 16 Bit ausreichend.

Eine generelle Beobachtung ist die variable Breite der Opcodes. Damit ist es nicht möglich diese mit einer einheitlichen Bitmaske abzutrennen. In der Konzeption wird dieser Umstand wieder aufgegriffen.

In Abbildung 3.3 auf Seite 19 sind die beschriebenen Aktionen in einem Zustandsmodel angeordnet.

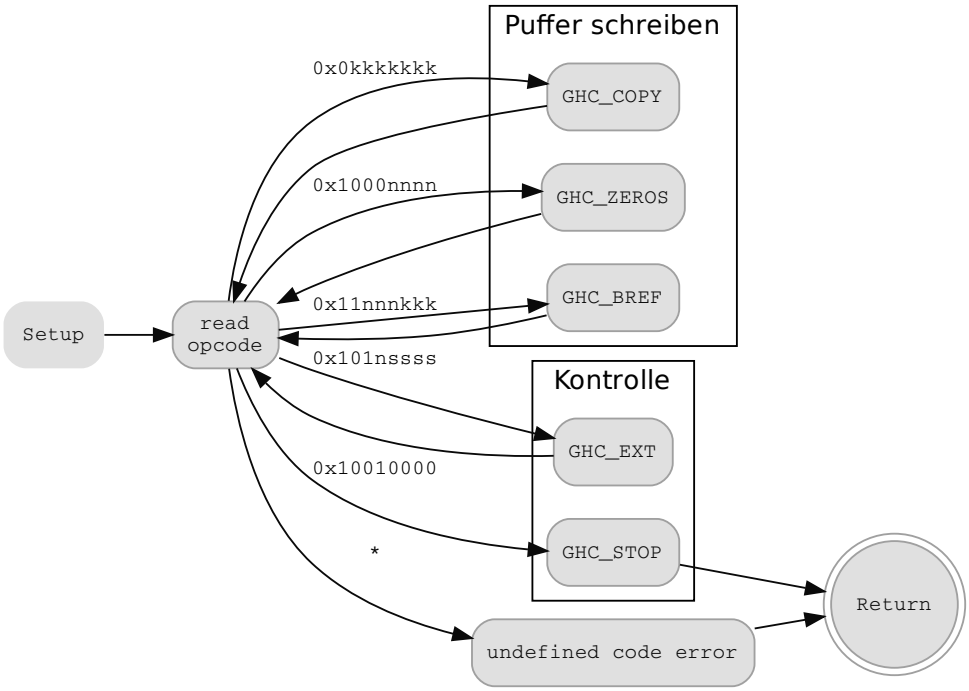

Abbildung 3.3: Zustandsmodel des GHC Decoders

Encoder Eine interessante Feststellung zum Encoder ist, dass er nicht zwingend implementiert sein muss. Ein Knoten der als GHC fähig registriert ist muss codierte Pakete korrekt verarbeiten können. Es existiert keine Definition die vorgibt, dass ein Knoten der GHC beherrscht nur noch codierte Pakete versenden muss.

Es gibt verschiedene Ansätze, wie auch schon im RFC erwähnt, den GHC Bytecode zu erzeugen. Mit dem Wissen um eine spezifische Struktur der Nutzdaten können zum Beispiel unveränderiche Teile einmal codiert, gespeichert und nach Bedarf wieder ausgelesen

werden.
Mit dieser Implementierung soll ein allgemeiner Ansatz verfolgt werden. Das bedeutet, dass die Nutzdaten eine Sequenz von Oktetts darstellen und ansonsten keinerlei bekannte Semantik oder Struktur besitzen.

Somit ist der Encoder eine Umkehrfunktion des Decoders und im Folgenden wird ermittelt, wie eine solche Sequenz in Teilsequenzen zerlegt werden kann, die sich mit einem oder mehreren der Codebytes ausdrücken lassen.

Für den Encoder gilt die Zuordnung:

Quellpuffer Originale Nutzdaten mit vorangestellten initialen Wörterbuch.

Zielpuffer Bytecode mit den komprimierten Nutzdaten.

Die elementare Aktion ist die Erzeugung des `GHC_COPY` Codebyte inklusive der literalen Nutzdaten. Dieses Codebyte ist das einzige, das unter allen Bedingungen Nutzdaten codieren kann und ausreichend um einen funktionalen Encoder zu implementieren. Die beiden anderen Codierungen sind nur teilweise anwendbar.

- Das `GHC_ZEROS` Codebyte kann nur Sequenzen von Nullen codieren.
- Das `GHC_ZEROS` Codebyte hat statistisch nur eine Chance kleiner als eins, dass sich eine zum aktuellen Präfix passende Sequenz im Wörterbuch befindet.
  Um einen Treffer mit der vorgegebenen Mindestlänge von zwei zu garantieren, müssten sich im Wörterbuch sämtliche Permutationen von zwei Elementen des Alphabetes befinden. Das sind bei dem gegebenen Alphabet mit 2^8 Elementen ein Wörterbuch mit mindestens $2^{8 \cdot 2} = 65536$ Elementen. Bei einem vordefinierten Wörterbuch mit 48 B als Startbedingung der Codierung ist das ausgeschlossen.

`GHC_COPY` ist gleichzeitig auch der Sonderfall bei dem das codierte Ergebnis mehr Platz beansprucht als die originalen Daten. Dieser Sonderfall existiert in jedem Kompressionsverfahren[8, Seite 227, Theorem 6.13].
Für eine Anzahl von n Symbolen müssen die Symbole selbst geschrieben werden und zusätzlich das Codebyte `GHC_ZEROS`. Damit ergibt sich ein Kompressionsfaktor von $\frac{n}{n+1} < 1$, das Gegenteil von dem was erreicht werden soll.

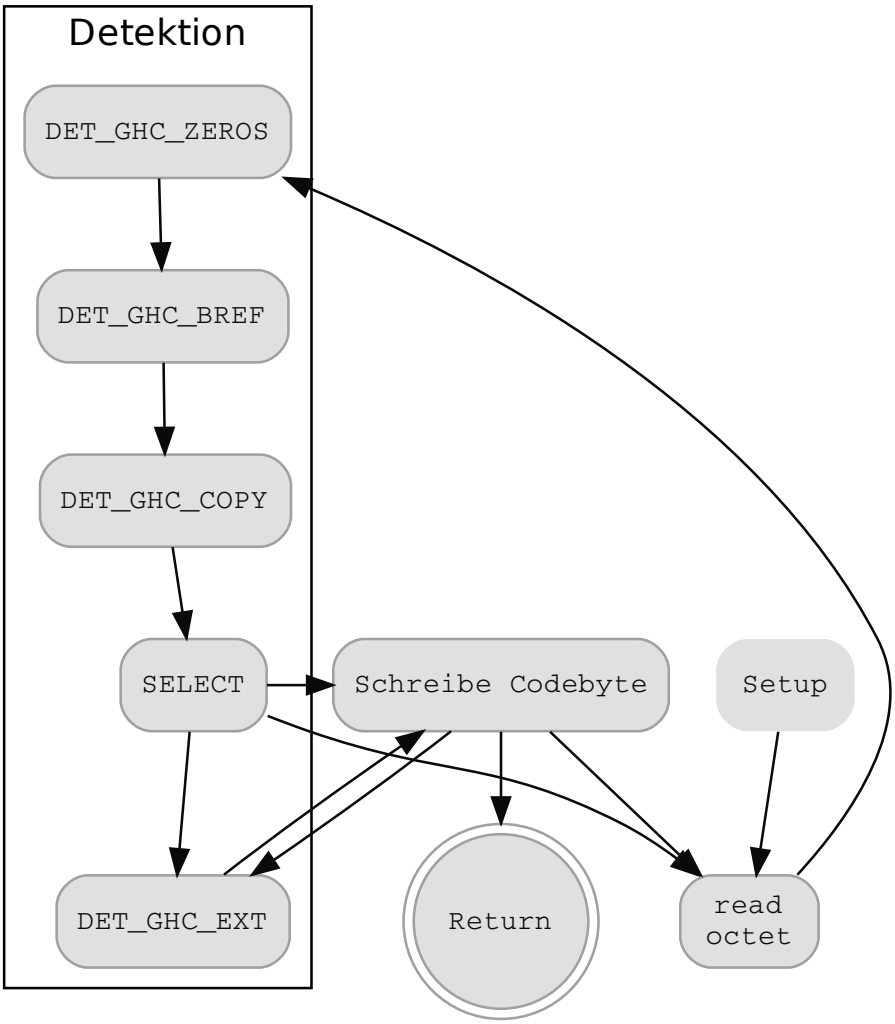

Abbildung 3.4: Zustandsmodel des GHC Encoders

Zerlegung in Teilsequenzen Der Encoder durchläuft den Quellpuffer mit den Nutzdaten elementweise und fügt das aktuelle Element an eine `GHC_COPY` Sequenz an, wenn nicht:

1. ein aktueller Präfix eine Sequenz von Nullbytes mit der Mindestlänge zwei ist.
2. ein aktueller Präfix mit der Mindestlänge zwei im Wörterbuch gefunden werden kann.
3. dieses Element das 96te einer Sequenz wäre.

Die Mindest- oder Maximallängen ergeben sich aus den Definitionen von `GHC_ZEROS` `GHC_BREF` und `GHC_COPY`. Bei der Encodierung müssen diese eingehalten werden, da sonst Parameter der Codebytes außerhalb der Definitionsbereiche liegen würden.

In allen drei Fällen wird ein entsprechendes Codebyte, bestehend aus dem Opcode und den jeweiligen Parametern, formatiert und in den Zielpuffer geschrieben. Im zweiten Fall müssen zusätzlich ein oder mehrere `GHC_EXT` erzeugt und im Zielpuffer angehängt werden, falls die ermittelten Parameter zu groß werden, um komplett im Codebyte von `GHC_BREF` untergebracht zu werden.

Komplexizitätsbetrachtung Für einen Quellpuffer mit n Elementen erfolgt eine Iteration über die Elemente des Quellpuffers. Für jedes Element wird geprüft, ob es eines der Elemente im Wörterbuch eine Übereinstimmung zeigt. Gefolgt von einem Vergleich der folgenden Elemente, um eine möglichst lange Sequenz zu finden. Damit hat dieses Verfahren eine Zeitkomplexität n^3.

3.2.1 Unterschiede zu LZ77

Das Verfahren wird als LZ77-ähnlich angegeben. Bei genauer Betrachtung gibt es jedoch Aspekte, die unterschiedlich sind. In Abschnitt 3.1 wurden die spezifischen Eigenschaften von LZ77 bereits beschrieben.

LZ77 beginnt mit einem leeren Suchpuffer am Anfang der zu codierenden Nutzdaten. Für GHC wird ein vordefiniertes Wörterbuch spezifiziert. Der Zustand ist so, als wären bereits 48 B verarbeitet, aber das aufgelaufene Ergebnis verworfen worden.

Die gängigen LZ77 Beschreibungen codieren mit einem 3-Tupel bestehend aus (Index, Länge, Symbol). GHC's `GHC_BREF` hat nur die ersten beiden. Für die nicht Treffer ist mit `GHC_COPY` eine dedizierte Operation spezifiziert.

Die Aktion `GHC_ZEROS` entspricht der Decodierung einer Lauflängencodierung und ist eine spezifische Erweiterung der GHC. LZ77 definiert nur die literale Kopie und die Referenz auf das Wörterbuch.

Der Parameter s der Aktion `GHC_BREF` ist von n abhängig. Durch die Zuweisung $s = 0b00000kkk + sa + n$ gilt immer $s \geqslant n$. Der in Abbildung 3.2 Seite 17 dargestellte Fall bei dem sich die Treffer überschneiden, muss jedoch so codiert werden, dass die Länge größer ist als der Index der Position.

3.3 Anwendung in der Next Header Compression

In den Abschnitten *3.1. Compressing Payloads (UDP and ICMPv6)* und *3.2. Compressing Extension Headers* wird definiert wie die Daten versandt und empfangen werden. Und es wird die Verbindung zu spezifischen Abschnitten der Pakete des Netzwerk Layers (IPv6) hergestellt.

Die Einbettung von GHC codierten Inhalten ist eine Erweiterung der Next Header Compression (NHC) die in dem RFC 6282 als Standard eingeführt wurde. NHC ist ein Bestandteil der, ebenfalls im RFC 6282 eingeführten, 6LoWPAN IPv6 Header Compression (IPHC).

Die Signaturen der Dispatch- oder Formatbytes und ihre Protokollzuordnungen sind in Abbildung 3.7 Seite 25 und Abbildung 3.6 Seite 25 in einer Baumstruktur dargestellt.

Im Fall des Encoders stammen die Nutzdaten, die im Quellpuffer übergeben werden aus der IPv6 Schicht und können verschiedenen Protokollen entnommen sein.

In Abbildung 3.7 Seite 25 ist ein IPHC Header dargestellt. Das markierte NH bit is die Kennung, ob im Anschluss an den ein NHC header folgt.

Unter der Voraussetzung, dass der Netzwerkstack bereits implementiert ist, kann die Einbindung von GHC mit der Erweiterung bestehender Mechanismen durchgeführt werden.

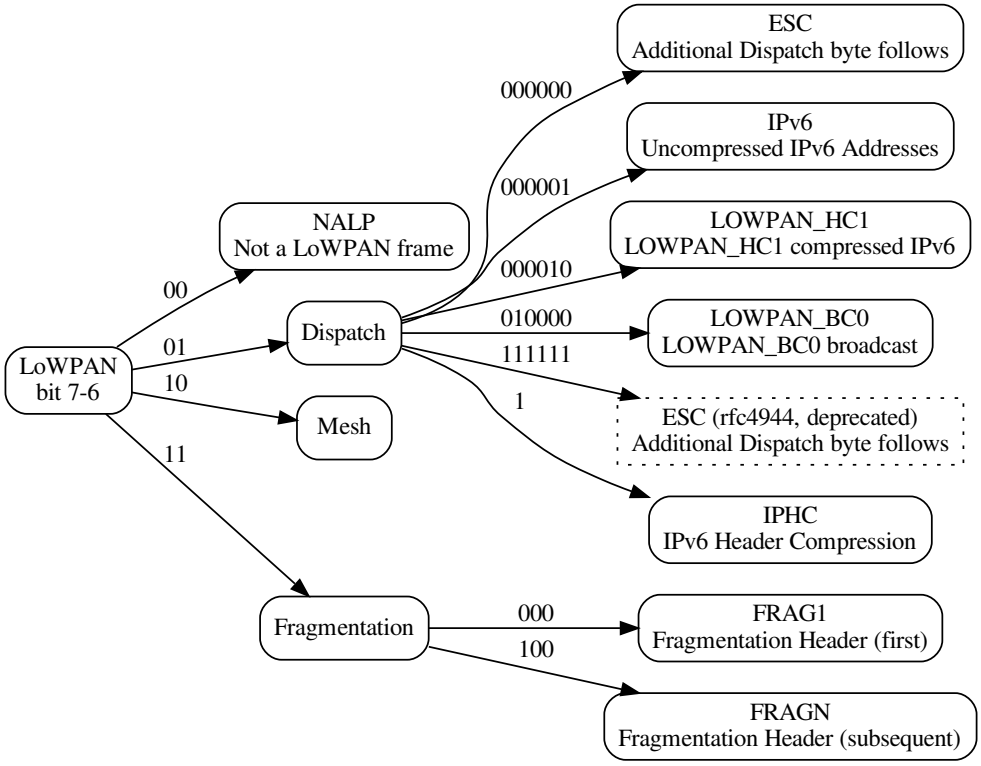

Abbildung 3.5: Zuordnungen der 6LoWPAN Dispatch Typen

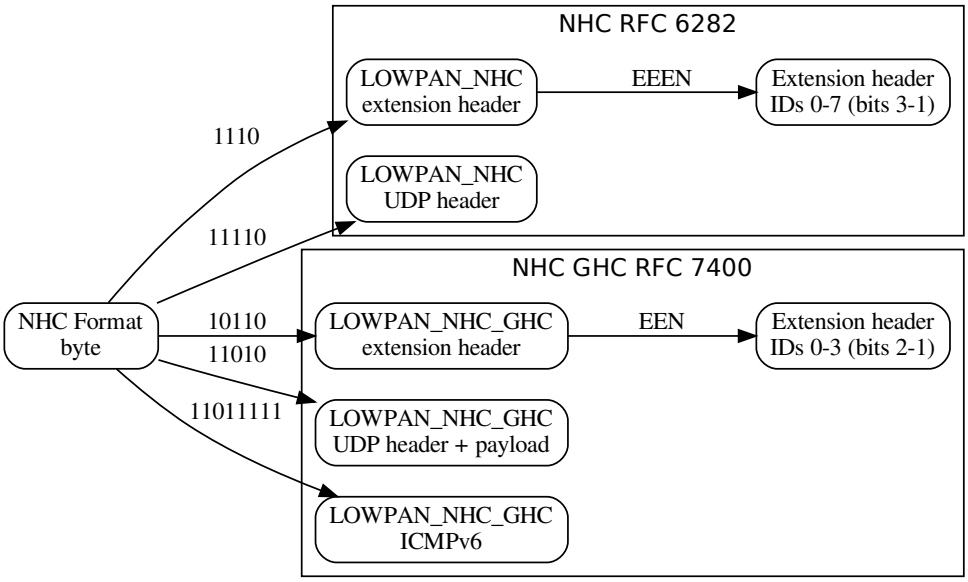

Abbildung 3.6: Zuordnungen der NHC Format Typen

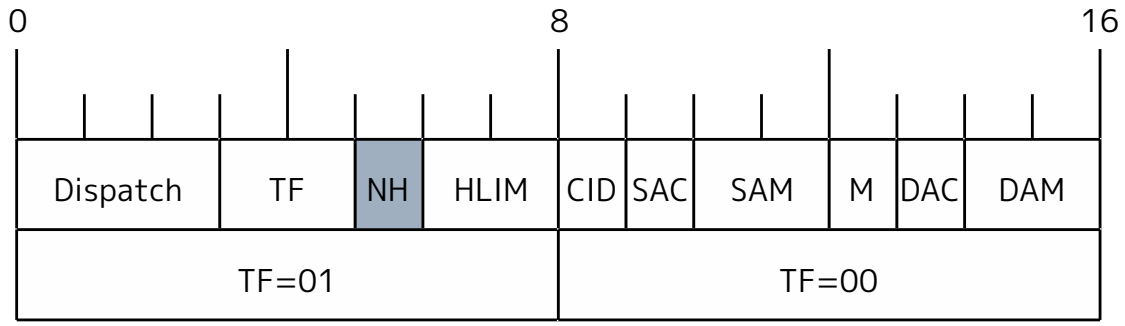

Abbildung 3.7: 6LoWPAN IPHC header

```
1  #ifdef MODULE_GNRC_SIXLOWPAN_IPHC_NHC
2      if (iphc_hdr[IPHC1_IDX] & SIXLOWPAN_IPHC1_NH) {
3          switch (iphc_hdr[payload_offset] & NHC_ID_MASK) {
4              case NHC_UDP_ID:
5                  payload_offset = iphc_nhc_udp_decode(pkt, dec_hdr,
                       ↪datagram_size,
6                                                       payload_offset +
                                                            ↪offset);
7
8                  if (payload_offset != 0) {
9                      payload_offset -= offset;
10                 }
11
12                 *nh_len += sizeof(udp_hdr_t);
13                 break;
14
15             default:
16                 break;
17         }
18     }
```

Listing 3.1: Auswertung des NHC Formates

3.3.1 Empfang von NHC

RIOT unterstützt bereits IPHC und NHC.

Von den aus RFC 6282 stammenden NHC Formaten wird in dem genutzten Release nur die UDP Header Compression unterstützt.

Die Datei `sys/net/gnrc/network_layer/sixlowpan/iphc/gnrc_sixlowpan_iphc.c` definiert die Funktion *gnrc_sixlowpan_iphc_decode()*. Dort ist mit einer `switch..case` Anweisung die Auswertung des NHC Format Bytes durchgeführt.

An dieser Stelle sind die Erweiterungen mit den GHC spezifischen Formaten für den Empfang durchzuführen. NHC extension header werden an dieser Stelle nicht detektiert.

Die NHC Format IDs haben per Definion variable Längen. Der Erweiterung mit weitern Formaten wird die Umwandlung des `switch..case` in eine Alternative vorausgehen müssen.

3.4 Nicht funktionale Anforderungen

Die primären Zielplattformen der Implementierung sind MCU basierte, Ressourcen-beschränkte eingebettete Systeme.[5] Neben den bindenden Vorgaben der Spezifikationen in RFC 7400 werden die Empfehlungen, die mit der Programmierung von Mikrocontrollern, eingebetteten Systemen und der systemnahen Programmierung verbunden sind.[21]

Lizenzfreiheit Nutzung von Open Source Bibliotheken und Werkzeugen. Eine vollständige Liste befindet sich in Abschnitt 2.6 Seite 12

Platformunabhängigkeit

- Nutzung von C oder evtl. C++ als Sprache.
- Beschränkung auf Standardbibliotheken.
- Benutzung von austauschbaren Datentypen (stdint.h).

Speichernutzung

-
- Statischer Speicher bei der Kompilierung zugewiesen. *malloc()* wird vermieden oder existiert nicht.
- Allozierung von Speicher in Funktionen vermeiden. Rückgabewerte als vorbelegte Puffer übergeben.
- Nutzung van Arrays statt Listen, falls diese nicht explizit gebraucht werden.
- Vermeidung der Duplizierung von Datenstrukturen. Schutz vor Modifikationen mit *const*
- Im Falle von C/C++ *inline* Funktionen nutzen. Das ermöglicht Strukturierung ohne Stapelspeicher zu belegen. Sehr gut mit *static inline*

4 Konzept

In diesem Kapitel wird es darum gehen die Vorgaben, die in dem vorhergehenden Kapitel gewonnen wurden für eine Konkretisierung des Programmcodes heranzuziehen.

Dazu gehören das Kapitel 3, Bestimmung der Werkzeuge Abschnitt 2.6 Seite 12 und zusätzlich für RIOT die „Coding conventions“

Es werden Festlegungen von wichtigen Datentypen, Entscheidungen über Programmstrukturen getroffen. Bezogen auf die RIOT Integration werden die bereits bestehenden Protokollteile bestimmt und Überlegungen zur Einbindung/Erweiterung angestellt.

Aus der Analyse hervorgehende Konstanten (numerische Werte, Bitmasken), werden später als einer der ersten Schritte der Implementierung als Präprozessor Makros abgelegt. In der Analyse eingeführte Bezeichner werden dort als Wortstamm von weiteren Bezeichnern wiederzufinden sein.

4.1 GHC Codec

Der Codec stellt kein eigenständiges Programm dar und soll als einen Modul implementiert werden, das in verschiedene Netzwerkstacks eingebunden werden kann.

Für das Interface reichen zwei Funktionsdefinitionen aus, von denen jeweils eine die Decodierung und Encodierung ausführt. Die Funktionsnamen werden mit *ghc_decode()* und *ghc_encode()* festgelegt.
Die Eingangsargumente sind für beide Funktionen die IPv6 Sender- und Empfängeradressen des Paketes und ein Array.
Das Ergebnis ist ebenfalls ein Array mit dem Zielpuffer und gegebenenfalls eine Statusrückgabe.

```
1  struct ghc_codec{
2      uint8_t *decoded;       /*!< dictionary + payload buffer */
3      uint8_t *encoded;       /*!< GHC bytecode buffer */
4      size_t size_deco;       /*!< payload size */
5      size_t size_enco;       /*!< bytecode size */
6      uint16_t pos_deco;      /*!< payload  cursor */
7      uint16_t pos_enco;      /*!< bytecode cursor */
8      uint16_t na;            /*!< length extension */
9      uint16_t sa;            /*!< index extension */
10 };
```

Listing 4.1: Struktur eines GHC Codecs. Enthalten sind Referenzen auf die Datenpuffer und Zustandsdaten für die schrittweise Abarbeitung

Das Quell- und Zielarray werden als Zeiger übergeben. Das verhindert die Duplizierung von Daten. Für den Zielpuffer hat dies den Vorteil, dass der Speicher nicht innerhalb der Funktionen alloziert werden muss.

In C müssen bei der Übergabe von Zeigern die Größen in separaten Parametern übergeben werden. Das bedeutet schon jetzt eine Anzahl von insgesamt 4 Parametern. Es ist empfehlenswert diese Parameter in einer Struktur zusammenzufassen und diese in Form eines Zeigers zu übergeben. In Listing 4.1 Seite 29 ist die endgültige Struktur dargestellt. Zusätzlich zu den angesprochenen Variablen befinden sich darin zusätzlich Zustandsinformationen, zum Beispiel Positionen zum Durchlaufen der Puffer.

4.1.1 Der Decoder

Virtuelle Maschine Wie aus der Analyse hervorgeht, hat der Decoder das Verhalten einer sehr einfachen virtuellen Maschine zur Ausführung von Bytecode mit insgesamt fünf Instruktionen. Ein Automat, der eine solche simple VM abbildet, ist in Abbildung 3.3 Seite 19 dargestellt.

Ein solcher Automat kann auch innerhalb einer MCU mit wenig Speicherbedarf Implementiert werden. Die Standardlösung hierfür ist in C eine `switch..case` Anweisung.

Im Fall des Decoders sind die Opcodes der verschiedenen Codebytes die Basis für die Verzweigung. Allerdings existiert keine einheitliche Maske für die Opcode Bits und der `switch..case` in C kann nur mit ganzzahligen Werten gesteuert werden. Eine funktionierende Lösung würde auf daher auf 265 Fälle hinauslaufen.

Ein Ersatz lässt aber sich mit einer `if..else` Kaskade herstellen. In Listing 4.2 Seite 30 ist eine der Verzweigungen dargestellt.

Die Zustände `GHC_BREF`, `GHC_COPY`, `GHC_EXT`, `GHC_STOP`und `GHC_ZEROS` aus Abbildung 3.3 Seite 19 gehen jetzt in Funktionen über, die die definierten Operationen auf den Puffern ausführen.

```
1  else if ( GHC_ZEROS_BC == (decoder->encoded[decoder->pos_enco] &
       ↪GHC_ZEROS_MASK)) {
2      /*
3       * Assertions: preconditions
4       * decoder->size_enco >= decoder->pos_enco + 1 (command byte)
5       * decoder->size_deco >= decoder->pos_deco + n
6       */
7      n = (decoder->encoded[decoder->pos_enco] & GHC_ZEROS_CNT_MASK) + 2;
8      if (decoder->pos_deco <= decoder->size_deco - n) {
9          append_zeros(decoder, n);
10     } else {
11         clean = 0;
12         retval = -31;
13     }
14     /*
15      * postconditions
16      * decoder->pos_deco += n
17      */
18 }
```

Listing 4.2: Beispiel des Codebytes `GHC_ZEROS`. Erkennung des Opcodes durch Maskierung und Vergleich, Prüfung der Überschreitung von Puffergrenzen vor dem Aufruf und Übergabe eines `struct ghc\_codec` an die Funktion

Da diese Funktionen nur einmalig innerhalb des Moduls aufgerufen werden können sie als `static inline` Speicher effizient umgesetzt werden.

Die Überschreitung von Puffergrenzen kann vor dem Aufruf erfolgen, da die aktuellen Positionen und Anzahl der auszuführenden Schritte bekannt sind.
Damit kann eine Einsparung linearer Laufzeit erreicht werden, wenn diese Prüfung nicht innerhalb der internen Schleifen bei jedem Durchlauf gemacht werden muss.

4.1.2 Der Encoder

Der Ablauf der Encodierung des Bytecodes, durch Zerlegung der Nutzdaten des Quellpuffers in Sequenzen, ist schon während der Analyse Abschnitt 3.2 Seite 19 beschrieben worden.

Präfixsuche mit dem „naiven“ Ansatz Die Suche nach einem möglichst langen Präfix entspricht dem Verfahren LZ77 und resultiert in einem $\Theta(n^3)$ Laufzeitverhalten. Es existieren verschiedene Ansätze, um das Laufzeitverhalten zu reduzieren. Diese Optimierungen sind teilweise sehr komplex und ohne weitere Vertiefung nicht einfach zu implementieren.

Die Optimierungen setzen die Verwendung von zusätzlichen Datenstrukturen voraus, die zusätzlichen Speicher belegen[3]. Abbildung 4.1 Seite 31

Table I. Theoretical Comparison of LZ Algorithms

Algorithm	Asymptotic Worst-Case Time	Space (words)	Output Feature	Special
KK	$\Theta(n)$	$\approx 2.75n$	(2)	$\sigma = 4$
AKO	$\Theta(n)$	$4.25n+$	(2)	
CPS1–1	$\Theta(n)$	$4.25n+$	(2)	
CPS1–2	$\Theta(n)$	$3.25n+$	(2)	
CPS1–3a	$\Theta(n)$	$3.25n$	(2)	
CPS1–3b	$\Theta(n)$	$2.25n+$	(2)	
CI1	$\Theta(n)$	$3.25n$	(3)	
CI2	$\Theta(n)$	$3n+$	(3)	
CIS	$\Theta(n)$	$3n+$	(3)	
CII	$\Theta(n)$	$3n+$	(3)	
	$\Theta(n)$	$3n$	(3)	
CPS2	$\Theta(n \log n)$	$1.5n$	(2)	
CPS3	$\Theta(n^2)$	$1.25 - 1.5n$	(2)	
OS	$\Theta(n \log^3 n)$	$\approx 1.2n$	(2)	on-line

Abbildung 4.1: Tabelle aus A Comparison of Index-Based Lempel-Ziv LZ77 Factorization [1]

Bezüglich der in Unterabschnitt 3.2.1 Seite 22 aufgelisteten Punkte müsste ein solcher optimierter Algorithmus erst analysiert und gegebenenfalls modifiziert werden.

All diese Einwände zusammen genommen, wird im Rahmen dieser Arbeit entschieden den „naiven“ Algorithmus für die Suche im Wörterbuch zu Implementieren. Damit wird die hohe Komplexität der Laufzeit bewusst in Kauf genommen, um der Anforderung nach minimalen Speicherverbrauch nachzukommen. Gleichzeitig können alle Anforderungen des RFC 7400 unmittelbar implementiert werden.
Die Implementierung soll so strukturiert werden, dass der Encoder leicht durch eine Alternative ersetzt werden kann. Damit ist zumindest ein funktionaler Ausgangspunkt geschaffen.

4.1.3 Testvorbereitung

Beispiele Im *Appendix A.* des RFC sind neun Beispiele für verschiedene Datenpakete aufgelistet. Dies sind konkrete Headerdaten von Protokollen, die auch über GHC transportiert würden.
Neben Pseudocode für den schrittweisen Ablauf der Codierung beinhalten sie auch die bearbeiteten Binärdaten in hexadezimaler Darstellung.
Folgende Blöcke sind jeweils für jedes Beispiel vorhanden:

Payload Die originalen Nutzdaten.

Dictionary Quell- und Zieladresse + vordefiniertem statischem Teil.

Compressed Eine der möglichen Codierungen des *Payload*.

Diese Datenblöcke bieten die Möglichkeit als Referenzdaten in Testfällen für den Decoder und Encoder eingesetzt zu werden. Dafür sollen Sie im Quellcode in passenden Arrays abgelegt werden.

```
1 /* RFC7400, Appendix A.  Examples, Figure 08 */
2 /* 8 bytes */
3 uint8_t ghc_expl_08_plod[] = {
4     0x9b, 0x00, 0x6b, 0xde, 0x00, 0x00, 0x00, 0x00
5 };
```

Listing 4.3: Array mit den Payload eines Beispiel

Die zu einem Beispiel gehörenden Blöcke werden dann strukturiert zusammengefasst. In Listing 4.4 Seite 33 ist die Struktur mit den Referenzdaten eines Testfalls abgebildet. Neben Zeigern auf die Datenblöcke sind ebenfalls deren jeweilige Größen gespeichert.

```
1  /*!
2   * rfc7400 test/example case.
3   *
4   * Reference data collection contains pointers to byte arrays (uint8_t*)
5   * with the data for:
6   * - payload Reference for source of a encoding, target of a decoding.
7   * - dictionary Pre filled lookup buffer.
8   * - encoded Reference for target of a encoding, source of a decoding.
9   */
10 struct ghc_case_ref {
11     /*! @{ \name Array size data.
12     *  Lengh fields for the correspondig arrays.
13     */
14     size_t   payload_len;       /*!< #payload array size */
15     size_t   dictionary_len;    /*!< #dictionary array size */
16     size_t   encoded_len;       /*!< #encoded array size */
17     /*! @} */
18     /*! @{ \name Test data array references. */
19     uint8_t* payload;           /*!< Unencoded header payload */
20     uint8_t* dictionary;        /*!< Predefined lookup buffer */
21     uint8_t* encoded;           /*!< Compressed #payload. */
22     /*! @} */
23 };
```

Listing 4.4: Struktur für die Daten und Metadaten eines Testfalls

Die *Testfälle* und -suiten sollen noch vor der eigentlichen Implementierung realisiert werden. Der Aufruf der Tests kann im Buildsystem als Target hinterlegt werden. Danach können während der Implementierung die Tests leicht wiederholt werden. Zu Beginn werden diese scheitern, aber es besteht der Vorteil, beim Debuggen gute Hinweise auf die Herkunft von Fehlern zu bekommen.

Testfall für *ghc_decode()*

1. Erstelle einen Quellpuffer mit dem codierten Datenblock der Referenz `ghc_case_ref.encoded`.
2. Erstelle einen leeren Zielpuffer und fülle den Anfang mit dem statischen Wörterbuch der Referenz `ghc_case_ref.dictionary`.
3. Initialisiere einen `struct ghc_codec` mit Quell und Zielpuffer.
4. Rufe *ghc_decode()* mit dem struct als Parameter auf.

5. Vergleiche Länge und Inhalt des Zielpuffers mit den Originaldaten der Referenz `ghc_case_ref.payload`.

Testfall für *ghc_encode()*

1. Erstelle einen Quellpuffer aus dem Wörterbuch und den Originaldaten der Referenz (`ghc_case_ref.dictionary` + `ghc_case_ref.payload`)
2. Erstelle einen leeren Zielpuffer.
3. Initialisiere einen `struct ghc_codec` mit Quell und Zielpuffer.
4. Rufe *ghc_encode()* mit dem struct als Parameter auf.
5. Vergleiche Länge und Inhalt des Zielpuffers mit dem codierten Datenblock der Referenz `ghc_case_ref.encoded`.
6. Führe einen Testfall für *ghc_decode()*! durch. Erstelle jetzt aber den Quellpuffer an Stelle der Referenzdaten mit dem gerade codierten Datenblock.

Testsuite für *ghc_decode()* und *ghc_encode()* Es existieren insgesamt 10 Sätze an Testdaten. Es wird für die vorher beschriebenen Testfälle jeweils eine *Testsuite* eingerichtet bei der über die Testdatensätze iteriert, und mit jedem der entsprechende Testfall durchgeführt wird.

4.2 NHC Einbettung in der IP Header Compression

4.2.1 Testvorbereitung

Für die Überprüfung des Netzwerkverkehrs soll ein Monitoring mit einem sogenannten Sniffer und Wireshark einem Tool zu Netzwerkanlyse durchgeführt werden. Als Hardware dient ein IEEE 802.15.4 Netzwerk Interface mit prinzipiell dem gleichen Aufbau wie ein Wi-Fi USB-Stick. Aus dem Linux-WPAN Projekt existiert für aktuelle Kernel ein Treiber mit dem ein Monitor Interface eingerichtet werden kann. Der Vorteil in diesem Modus ist dass der Sniffer nicht sendet, und damit auch nicht die Kommunikation beeinflussen kann.

Mit Wireshark kann auf dieses Interface zugegriffen werden, wie auf auch auf jede Netzwerkkarte.

5 Implementierung und Testläufe

5.1 Das Modul ghc_codec

Die Entscheidungen zur Interface, Datentypen, Programmstrukturen und erwähnenswerten Implementierungsdetails sind bereits im Kapitel zur Konzeption beschrieben. Der realisierte Quellcode befindet sich auf der beiliegenden CD als ZIP-Archiv und ist in einem dafür eingerichteten Repository zu finden.[12] Er sollte mit den auf Abschnitt 2.6 Seite 12 aufgelisteten Programmen lauffähig sein.

Für die Implementierung wurde eine einfache Verzeichnisstruktur angelegt wie in Abbildung 5.1

```
ghc_codec
├── Makefile
├── test_ghc_decode.c
├── test_ghc_encode.c
├── src
│   ├── Makefile
│   ├── ghc_decode.c
│   └── ghc_encode-naive.c
└── include
    ├── ghc.h
    ├── ghc_codec_intern.h
    ├── ghc_codec.h
    └── tests
        ├── rfc7400_examples_testsuite.h
        └── rfc7400_examples.h
```

Abbildung 5.1: Verzeichnis mit Quellcode für `ghc\_codec`

Verzeichnis <ghc_codec/inlude>

<ghc_codec.h> beinhaltet die Deklarationen für des Interface des Moduls. Die muss für die Verwendung inkludiert werden.
<ghc_codec_intern.h> deklariert die „privaten“ Funktionen, zur ausschließlichen Verwendung innerhalb von *ghc_decode()* und *ghc_decode()*.
Die Datei <ghc.h> beinhaltet alle Konstanten in Form von Präprozessor Definitionen.

Verzeichnis <ghc_codec/src>

Hier liegen die Dateien, mit denen das Modul mit den Funktionen *ghc_decode()* und *ghc_encode()* implementiert ist.
Durch den Namenszusatz der Datei für *ghc_encode()* können alternative Implementierungen nebeneinander abgelegt und vom Buildsystem ausgewählt werden.

Verzeichnis <ghc_codec>

Die Dateien <test_ghc_decode.c> und <test_ghc_encode.c> realisieren die Testsuiten.

5.1.1 Testgestützte Implementierung

Die Headerdateien in <includes/tests> sind zu einem frühen Zeitpunkt mit Informationen aus dem RFC erstellt worden.
Die beiden Testsuiten Unterabschnitt 4.1.3 Seite 32 Nach der Erstellung der Header in <ghc.h> und <ghc_codec.h>

Bei Implementierung ist die gleiche Reihenfolge eingehalten worden, wie bereits in Analyse und Konzept. Erst ist die Realisierung des Decoders erfolgt. Nach erfolgreichem Abschluss aller Tests für den Decoder wird dieser in die Testsuite für den Encoder zusätzlich eingebunden.

Ein Ausschnitt der Konsolenausgabe ist im Anhang zu sehen. An mehreren Punkten der Implementierung hat die nummerierte Ausgabe der Vergleiche nützliche Rückschlüsse für Haltepunkte und das Beobachten von Variablen mit dem *gdb* erlaubt.

Codierung nicht eindeutig Bemerkenswert ist die Tatsache, dass der Vergleich der Decodierergebnisse mit der Referenz Unterschiede, und somit vermutliche Fehler zeigt. Die anschließende Decodierung erzeugt hingegen wieder fehlerfreie Originaldaten.

Die ist ein Hinweis darauf, dass die Codierung nicht eindeutig ist. Ein leicht nachvollziehbares Beispiel ist ein Zustand, an dem ein Präfix aus drei Nullbytes besteht. Im Wörterbuch befinden sind ebenfalls eine gleiche Sequenz.
Als Codebyte kann jetzt `GHC_BREF` mit Referenz auf das Wörterbuch oder `GHC_ZEROS` mit dem Parameter 3 erzeugt werden. Dies zeigt bei einem Vergleich eine Differenz, wird aber bei korrekter Decodierung in beiden Fällen wieder zu einer Sequenz von drei Nullbytes wiederhergestellt.

5.2 Testlauf mit RIOT auf einer MCU

Nachdem Entwicklung und Test auf einem PC mit Linux durchgeführt worden sind, ist der nächste Schritt ein Testlauf mit dem Betriebssystem RIOT.

In dem Verzeichnis befindet sich ein Makefile, das den Konventionen des Buildsystem von RIOT entspricht. Ein hiermit erzeugtes Image von jeweils einer der kompletten Testsuiten läuft erfolgreich mit identischer Ausgabe auf einer SAMR21 MCU mit einer ARM Cortex CPU[14].

6 Zusammenfassung und Ausblick

GHC Decoder und Encoder Die Implementierung des Codec ist umgestzt. Das Modul ist lauffähig und zeigt mit Tests gegen Referenzdaten positive Ergebnisse.

Das Modul ist sowohl unter Linux, in der ursprünglichen Entwicklungsumgebung, als auch in RIOT OS auf einer MCU mit ARM Architektuer ohne Änderungen kompilierbar und zeigt bei den Tests identische Ergebnisse.

In der Konzeption ist entschieden worden einen geringen Speicherverbrauch dem Laufzeitverhalten vorzuziehen. In weiteren Arbeiten könnten zusätzliche Implementierungen des Encoders eingebunden und auf bessere Kompromisse überprüft werden.

RIOT Netzwerkstack Mit der Lauffähigkeit der Implementierung unter RIOT ist ein Teilschritt zur Integration erreicht worden. Für die Einbindung in den Netzwerkstack ist noch weitere Arbeit notwendig. Die Protokollebenen auf denen GHC aufsetzt sind in RIOT vorhanden. Bezüglich NHC ist nur ein Teil der Formate implementiert.

Literaturverzeichnis

[1] Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Evguenia Kopylova, W.F. Smyth, German Tischler, and Munina Yusufu. A Comparison of Index-based lempel-Ziv LZ77 Factorization Algorithms. *ACM Computing Surveys (CSUR)*, 45(1):1–17, December 2012. URL http://doi.acm.org/10.1145/2379776.2379781.

[2] Atmel. *Atmel-8271J-AVR- ATmega-Datasheet*. Atmel Corporation, November 2015. URL http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A\-168PA-328-328P_datasheet_Complete.pdf.

[3] Timothy Bell and David Kulp. Longest-match string searching for ziv-lempel compression. *Software: Practice and Experience*, 23(7):757–771, 1993.

[4] C. Bormann. 6LoWPAN-GHC: Generic Header Compression for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 7400, IETF, November 2014.

[5] C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks. RFC 7228, IETF, May 2014.

[6] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, IETF, March 1997.

[7] Contiki-NG. Contiki-NG: The OS for Next Generation IoT Devices - contiki-ng/contiki-ng, October 2018. URL https://github.com/contiki-ng/contiki-ng/tree/189fd594d3c368aace684f6e5992ed2f573074ac. Aufgerufen: 24.10.2018.

[8] Volker Heun. *Grundlegende Algorithmen. Einführung in den Entwurf und die Analyse effizienter Algorithmen*. Vieweg, Wiesbaden, 2nd edition, 2003.

[9] Peter Kietzmann, Martin Landsmann, Thomas C. Schmidt, Hauke Petersen, Martine Lenders, and Matthias Wählisch. Leistungsmessung eines modularen Netzwerk-Stacks für das IoT-Betriebssystem RIOT. In *Proc. of the 14. GI/ITG KuVS Fachgespräch Sensornetze (FGSN2015)*, pages 19–22, Erlangen-Nürnberg, Germany, Sep 2015. Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science.

[10] Peter Kietzmann, Thomas C. Schmidt, and Matthias Wählisch. RIOT - das freundliche Echtzeitbetriebssystem für das IoT. In *Internet der Dinge*, pages 43–52, Berlin, Nov. 2016. Springer Vieweg.

[11] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, IETF, September 2007.

[12] Andreas Pauli. Implementierung der 6LoWPAN Generic Header Compression, October 2018. URL https://github.com/A-Paul/ghc_codec/tree/631d33504b25258cd4a048f72f6dffec3af48939.

[13] The FreeBSD Project. Freebsd src tree (read-only mirror). contribute to freebsd/freebsd development by creating an account on github., October 2018. URL https://github.com/freebsd/freebsd/tree/c7b538ee7268350626d575b629e26191bbb22b33. Aufgerufen: 21.10.2018.

[14] RIOT. Board: SAMR21 xpro, October 2018. URL https://github.com/RIOT-OS/RIOT/wiki/Board%3A-SAMR21-xpro. Aufgerufen: 21.10.2018.

[15] RIOT-OS. 2018.07-branch, August 2018. URL https://github.com/RIOT-OS/RIOT/tree/bb6bedd708bbccc16d4a0cd9750f56efcd5f9244.

[16] David Salomon. *Data Compression: The Complete Reference*. Springer Science & Business Media, 3 edition, 2004.

[17] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmanm, 3 edition, 2006.

[18] Zach Shelby and Carsten Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley Publishing, 2009.

[19] TinyOS. Main development repository for TinyOS (an OS for embedded, wireless devices). - tinyos/tinyos-main, May 2018. URL https://github.com/tinyos/tinyos-main/tree/ad056121cc51c449d6de154d98604e2c167dbfd8. Aufgerufen: 12.05.2018.

[20] Linus Torvalds. Linux kernel source tree. Contribute to torvalds/linux development by creating an account on GitHub., October 2018. URL https://github.com/torvalds/linux/tree/35a7f35ad1b150ddf59a41dcac7b2fa32982be0e. Aufgerufen: 15.10.2018.

[21] Jörg Wiegelmann. *Softwareentwicklung in C für Mikroprozessoren und Mikrocontroller : C-Programmierung für Embedded-Systeme*. VDE Verlag GmbH, 7 edition, 2017.

[22] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977. ISSN 0018-9448. doi: 10.1109/TIT.1977.1055714.

A Anhang

```
Check encoding of RFC7400 example 14:
ghc_encode(ghc_suite_case_refs[06])
[size_enco:107],[size_deco:096]

returned with -01
!comp_size_diff: 0
000:0x0C,      001:0x86,      002:0x00,      003:0x55,      004:0xC9,      005:0x40,      006:0x00,      007:0x0F,
008:0xA0,      009:0x1C,      010:0x5A,      011:0x38,      012:0x17,      013:0x80,      014:0x06:0x04, 015:0x07,
016:0xD0,      017:0x01,      018:0x01,      019:0x11:0xA6, 020:0x22:0xC2, 021:0x82,      022:0x06,      023:0x03,
024:0x04,      025:0x40,      026:0x40,      027:0xFF,      028:0xFF,      029:0xC0,      030:0xD0,      031:0x82,
032:0x04,      033:0x20,      034:0x02,      035:0x0D,      036:0xB8,      037:0x8A,      038:0x04:0xA1, 039:0x20:0xC6,
040:0x02,      041:0x40,      042:0x10,      043:0xA4,      044:0xCB,      045:0x01,      046:0xE8,      047:0xA2,
048:0xF0,      049:0x02,      050:0x21,      051:0x03,      052:0xA9,      053:0xE6,      054:0xB3,      055:0xCD,
056:0xAF,      057:0xDB,

Check decoding of RFC7400 example 14:
ghc_decode(ghc_suite_case_refs[06])

returned with 105
000:0x86,      001:0x00,      002:0x55,      003:0xC9,      004:0x40,      005:0x00,      006:0x0F,      007:0xA0,
008:0x1C,      009:0x5A,      010:0x38,      011:0x17,      012:0x00,      013:0x00,      014:0x07,      015:0xD0,
016:0x01,      017:0x01,      018:0x11,      019:0x22,      020:0x00,      021:0x00,      022:0x00,      023:0x00,
024:0x03,      025:0x04,      026:0x40,      027:0x40,      028:0xFF,      029:0xFF,      030:0xFF,      031:0xFF,
032:0xFF,      033:0xFF,      034:0xFF,      035:0xFF,      036:0x00,      037:0x00,      038:0x00,      039:0x00,
040:0x20,      041:0x02,      042:0x0D,      043:0xB8,      044:0x00,      045:0x00,      046:0x00,      047:0x00,
048:0x00,      049:0x00,      050:0x00,      051:0x00,      052:0x00,      053:0x00,      054:0x00,      055:0x00,
056:0x20,      057:0x02,      058:0x40,      059:0x10,      060:0x00,      061:0x00,      062:0x03,      063:0xE8,
064:0x20,      065:0x02,      066:0x0D,      067:0xB8,      068:0x00,      069:0x00,      070:0x00,      071:0x00,
072:0x21,      073:0x03,      074:0x00,      075:0x01,      076:0x00,      077:0x00,      078:0x00,      079:0x00,
080:0x20,      081:0x02,      082:0x0D,      083:0xB8,      084:0x00,      085:0x00,      086:0x00,      087:0x00,
088:0x00,      089:0x00,      090:0x00,      091:0xFF,      092:0xFE,      093:0x00,      094:0x11,      095:0x22,
```

Die Blöcke resultieren aus einem Vergleich der Referenzdaten mit dem Ergebnis des Testkandidaten. Bei Gleichheit wird nur das Muster (Index,Wert1,) angezeigt. Nur bei einer Differenz wird der abweichende Wert ebenfalls mit ausgegeben (Index,Wert1,Wert2).

Abbildung A.1: Konsolenausgabe des Testfalles 06.

Glossar

6LoWPAN Implementierung des IPv6 Protokols auf Funknetzwerken mit geringem Energieverbrauch.

Bytecode Code mit einer Breite von 8 bit zur Ausführung in einer virtuellen Maschine..

GHC 6LoWPAN Generic header compression (RFC 7400).

IEEE 802.15.4 Spezifikation für Low-Rate Wireless Personal Area Networks (LR-WPAN).

IPHC 6LoWPAN IPv6 Header Compression (RFC 6282).

LZ77 Verlustfreies Kompressionsverfahren, basierend auf einem dynamischen Wörterbuch. 1977 vorgestellt von den Mathematikern Lempel und Ziv..

MCU Microcontroller Unit.

NHC Next Ceader Compression (RFC 6282) Siehe RFC 7400.

RFC Memos in der RFC-Dokumentenreihe enthalten technische und organisatorische Hinweise zum Internet..

RFC 7400 6LoWPAN-GHC: Generic Header Compression for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs).

RIOT RIOT The friendly OS for the IoT.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Ort	Datum	Unterschrift im Original