# Effiziente Rekonfiguration von Flash-basierten FPGAs in drahtlosen Sensornetzwerken

**Efficient Reconfiguration of Flash-based FPGAs in Wireless Sensor Networks**
Master-Thesis von Manuel Krönig
Tag der Einreichung:

1. Gutachten: Prof. Dr.-Ing. Andreas Koch
2. Gutachten: Dr.-Ing. Andreas Engel

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Eingebettete Systeme
und ihre Anwendungen

Effiziente Rekonfiguration von Flash-basierten FPGAs in drahtlosen Sensornetzwerken
Efficient Reconfiguration of Flash-based FPGAs in Wireless Sensor Networks

Vorgelegte Master-Thesis von Manuel Krönig

1. Gutachten: Prof. Dr.-Ing. Andreas Koch
2. Gutachten: Dr.-Ing. Andreas Engel

Tag der Einreichung:

**Erklärung zur Master-Thesis**

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 31. Mai 2017

(Manuel Krönig)

## Zusammenfassung

Drahtlose Sensornetzwerke haben vielfältige Einsatzmöglichkeiten, u. a. die Überwachung der Struktur von Bauwerken oder die Untersuchung von Bewegungsmustern seltener Tierarten. Um eine lange Laufzeit von batterie- oder akkugespeisten Netzwerkknoten zu ermöglichen, werden energieeffiziente Bauteile verwendet und es wird versucht, die Menge an kabellos übertragenen Daten durch Vorberechnungen auf dem Sensorknoten zu senken. Eine Möglichkeit, diese Berechnungen zu beschleunigen und somit energiesparende Ruhephasen des Netzwerkknotens möglich zu machen, ist die Verwendung eines Koprozessors. Wegen ihrer durch ihre Rekonfigurierbarkeit großen Flexibilität und ihrem Potential zu parallelen Berechnungen sind FPGAs als Koprozessoren gut geeignet. Eine Rekonfiguration eines FPGAs in einem bereits ausgebrachten Sensornetzwerk kann, je nach Einsatzort, sehr aufwändig sein, deshalb stellt die Möglichkeit einer drahtlosen Rekonfiguration (*OTAP*, *Over The Air Programming*) eine deutliche Arbeits- und damit Kostenersparnis dar. Da diese Rekonfiguration nur eine zusätzliche Aufgabe des Sensorknotens ist, soll diese ohne die Verwendung zusätzlicher Bauteile möglich sein.

In dieser Arbeit wird die Möglichkeit der drahtlosen Rekonfiguration eines Microsemi IGLOO AGL1000 durch einen Texas Instruments CC2531 (beide Teil eines HaLOEWEn-Sensorknotens der Version 3) untersucht. Wichtige Ziele sind dabei die Minimierung des Energiebedarfs und der Ausfallzeit (*Downtime*) des Netzwerks. Die Rekonfiguration wird über JTAG durch eine modifizierte Version der von Microsemi zur Verfügung gestellten Bibliothek DirectC durchgeführt. Da der interne Speicher des verwendeten Mikrocontrollers nicht zur vollständigen Speicherung einer Konfigurationsdatei des FPGAs ausreicht, werden die Konfigurationsdaten segmentweise übertragen. Um den Übertragungsvorgang zu beschleunigen, werden verschiedene Kompressionsverfahren auf ihre Eignung untersucht. Eine starke Beschränkung ist der für das Entpacken der Daten notwendige Rechenaufwand, da der verwendete TI CC2530 sehr langsam ist. Zusätzlich wird ein Vorhersageverfahren untersucht, bei dem zuvor aufgezeichnete Anfragen an Segmente der Programmierdateien verwendet werden, um die Übertragung der Daten während des Konfigurationsvorgangs im Hintergrund durchzuführen. Verglichen werden sowohl die Programmierzeiten also auch der Energieaufwand bei den unterschiedlichen Kombinationen aus verschiedenen Kompressionsverfahren, jeweils mit oder ohne Segmentvorhersage. Um die Auswirkungen der Übertragungsgeschwindigkeiten zu untersuchen, werden die Messungen jeweils mit kabelgebundener und kabelloser Übertragung durchgeführt. Aus Beobachtungen der Konfigurationszeit bei verschiedenen Übertragungsparametern wird eine Hypothese über einen optimalen Übertragungsmodus im Falle eines Übertragungsfehlers aufgestellt und mit einem gezielt verursachten Fehler untersucht. Die Auswertung der dazu durchgeführten Messungen zeigt, dass dieser neue Modus keine messbare Verbesserung bringt.

Die kabellosen Konfigurationszeiten liegen für den Fall einer unkomprimierten Übertragung ohne Vorhersage bei ca. 711 s. Durch die Verwendung der Segmentvorhersage kann die benötigte Zeit auf ca. 427 s gesenkt werden. Werden die Segmente ohne Vorhersage komprimiert übertragen ist die Konfigurationszeit abhängig von der Komprimierbarkeit der Konfigurationsdaten und liegt zwischen ca. 543 s für die einfachste und ca. 636 s für die komplexeste Konfigurationsdatei. Die für einen Konfigurationsvorgang benötigte Energie liegt im Fall der unkomprimierten Übertragung ohne Segmentvorhersage bei ca. 76,6 J. Mit der Segmentvorhersage kann die benötigte Energie auf ca. 45,7 J gesenkt werden. Wird die Segmentvorhersage nicht genutzt, aber komprimiert übertragen, hängt der Energiebedarf von der Komplexität der Konfigurationsdatei ab und liegt zwischen ca. 48,2 J und 68,2 J.

## Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Motivation

Sensornetzwerke können für eine Vielzahl verschiedener Aufgaben verwendet werden. Sie bestehen aus mehreren Sensorknoten, die miteinander gekoppelt sind. Ein naheliegendes Beispiel für ein kabelgebundenes Sensornetzwerk ist eine Feuermeldeanlage in einem Gebäude. Die einzelnen Rauchmelder sind hier die Sensorknoten. Während es bei dieser Anwendung noch (relativ einfach) möglich ist, dieses System mit kabelgebundenen Daten- und Stromleitungen zu implementieren, wenn dies beim Entwurf des Gebäudes berücksichtigt wurde, ist ein nachträglicher Einbau sehr aufwändig und damit teuer. Diese Problematik kennt man aus dem Alltag, wenn ein Kabel nachträglich unter Putz verlegt werden soll. Ein weiteres Problem bei der Verwendung von Kabeln kann durch die Starrheit und das Gewicht der Kabel und durch die mechanische Stabilität von Steck- und Lötverbindungen auftreten. Im Alltag begegnet man dieser Problematik u. a. bei den Kabelbäumen im Auto: Durch die wiederholten Bewegungen der Heckklappe kann es zu Problemen (z. B. Kabelbruch) mit den Kabeln, an denen der Scheibenwischer angeschlossen ist, kommen. Ebenso können die verwendeten Steckverbinder durch bei der Fahrt erzeugte Vibrationen gelockert werden. Bei Einsatz eines Sensornetzwerkes an abgelegenen Orten im Freien kann das Verlegen von Kabeln ebenso sehr aufwändig und teuer sein, da u. U. große Strecken überbrückt werden müssen.

Eine Lösung dieser Probleme ist die Verwendung eines drahtlosen Sensornetzwerkes (*Wireless Sensor Network*, *WSN*). Es besteht aus Sensorknoten, die per Funk miteinander und/oder mit einem (oder mehreren) Basisknoten (*Sink*) kommunizieren. Eine Möglichkeit, energieeffiziente und flexibel einsetzbare Sensorknoten zu konstruieren, ist die gemeinsame Verwendung eines Mikrocontrollers und eines FPGAs, wobei das FPGA zur Beschleunigung von Rechnungen des Mikrocontrollers verwendet wird. Eine kabelgebundene Rekonfiguration eines bereits ausgebrachten Sensornetzes ist zeit- und kostenaufwändig, deshalb bietet sich eine Rekonfiguration über die bereits vorhandene Funkverbindung an, solange dies aus energetischer Sicht möglich bzw. praktikabel ist.

## 1.2 Aufgabenstellungen

Diese Arbeit untersucht die drahtlose Rekonfiguration eines FPGAs (*OTAP*, *Over The Air Programming*) durch einen Mikrocontroller, die beide in einem Knoten eines drahtlosen Sensornetzwerkes verbaut sind. Da die verfügbaren Ressourcen (Rechenleistung, Speicher und verfügbare Pins) des Mikrocontrollers stark limitiert sind, muss die Übertragung der Konfigurationsdaten segmentweise erfolgen. Das hauptsächliche Ziel für mögliche Optimierungen ist es, den Energieverbrauch und die Ausfallzeit (Downtime) des Netzwerks während der Rekonfiguration des FPGAs zu minimieren. Dazu wird die Eignung verschiedener Kompressionsverfahren und die Möglichkeit der Vorhersage zukünftig benötigter Segmente untersucht. Als Maße für den Vergleich der Messergebnisse werden hauptsächlich die Konfigurationszeit und die für die Rekonfiguration benötigte Energie verwendet.

## 1.3 Gliederung der Arbeit

In Kapitel 2 wird der theoretische Hintergrund der Arbeit erläutert. In Kapitel 3 wird die Implementierung des Konfigurationsvorgangs dargestellt. In Kapitel 4 wird die Durchführung der Messungen beschrieben, deren Ergebnisse in Kapitel 5 vorgestellt und ausgewertet werden. Anschließend werden in Kapitel 6 andere Arbeiten zum Thema OTAP vorgestellt und, soweit möglich, mit den Resultaten dieser Arbeit verglichen. Ein Ausblick auf weiterführende Untersuchungen schließt diese Arbeit ab.

## 2 Grundlagen

In diesem Kapitel werden zunächst ausgewählte Anwendungsbeispiele von drahtlosen Sensornetzwerken aufgeführt. Danach wird ein Anforderungskatalog aus einer anderen Arbeit vorgestellt. Anhand dieser Kriterien werden einige Herausforderungen, vor allem im Zusammenhang mit dem Energiebedarf, genauer betrachtet. In Abschnitt 2.2 wird die verwendete Hardware und in Abschnitt 2.3 die verwendete Funkverbindung vorgestellt. Darauf folgt eine Abwägung zwischen den zwei vom Hersteller des FPGAs bereitgestellten Projekten zur Ansteuerung des FPGAs über die JTAG-Schnittstelle und in Abschnitt 2.5 eine Untersuchung der Verwendbarkeit externer Speichermodul(e) zum lokalen Zwischenspeichern einer verwendeten Konfigurationsdatei. Danach werden verschiedene Maße zur Bewertung der Konfigurationsvorgänge vorgestellt. Zuletzt werden verschiedene Algorithmen zur Kompression der Konfigurationsdaten ausgewählt und ihre Funktionsweise erläutert.

### 2.1 Drahtlose Sensornetzwerke

Die meisten Arbeiten zu drahtlosen Sensornetzwerken gehen davon aus, dass keine kabelgebundene Stromversorgung zur Verfügung steht und verwenden verschiedene in Kapazität und maximalem Stromfluss limitierte Energiequellen.

**Anwendungen:**

Mögliche Anwendungen für drahtlose Sensornetzwerke sind u. a. *Structural Health Monitoring* (*SHM*), Habitatüberwachung, Vulkanüberwachung und verschiedene Aufgaben in Landwirtschaft und im Gesundheitswesen: Auf dem Gebiet der Habitatüberwachung haben Liu et al. ein WSN namens MiluNet aus 85 festen und 30 mobilen Knoten entwickelt. Es dient der Erforschung des Verhaltens des Milu (Davidshirsch), einer seltenen Tierart in China [1]. Song et al. haben ein per Helikopter aus der Luft ausbringbares WSN zur Vulkanüberwachung entwickelt. Fünf Knoten wurden innerhalb einer Stunde am Mount St. Helens abgesetzt [2]. In der Landwirtschaft kann ein WSN verwendet werden, um die Bewirtschaftung eines Weingutes zu vereinfachen. Burrell et al. schlagen u. a. Konzepte zur Überwachung des Reifeprozesses und zur Warnung vor Befall durch Echten Mehltau an besonders anfälligen Reben-Standorten vor [3]. Chipara et al. haben ein WSN zur Kontrolle von Puls- und Sauerstoffsättigungswerten über einen Zeitraum von 7 Monaten mit 41 Patienten in einer Station für Patienten, die aus der Intensivstation entlassen wurden (*step-down cardiology unit*), untersucht [4].

SHM, also die engmaschige automatische Überwachung des Zustandes eines Gebäudes oder Objektes wird immer wichtiger, da die Bereitschaft, vermeidbare Risiken einzugehen, sinkt und auch das Bestreben, die Langlebigkeit bestehender Strukturen sicherzustellen, zunehmend steigt [5]. Abruzzese et al. nennen als vermeidbare Negativbeispiele im Hinblick auf den Erhalt von historischen Gebäuden den Einsturz des Markusturms in Venedig im Jahr 1902 und den Einsturz des Torre Civica (Civic Tower) in Pavia im Jahr 1989 [6]. Es gibt eine Vielzahl an Anwendungsbeispielen für SHM; u. a. gibt (bzw. gab) es ein WSN aus 64 festen Knoten an der Golden Gate Bride in San Francisco [7] und ein aus 63 Knoten bestehendes WSN an einer chinesischen Hängebrücke [1]. Ein weiteres drahtloses Sensornetzwerk, das auch zur Zustandsüberwachung verwendet werden kann, ist das in dieser Arbeit verwendete HaLOEWEn, das in Abschnitt 2.2 vorgestellt wird.

**Anforderungen:**

Potdar et al. nennen die folgenden möglichen Anforderungen an ein drahtloses Sensornetzwerk [8]:

1. Fehlertoleranz: Robustheit gegen Ausfälle der Knoten. Akustische Meldung über Fehlfunktion.
2. Skalierbarkeit: Je nach Anwendung sollen (sehr) viele Knoten verwendet werden können.

3. Lange Lebensdauer: Die Lebensdauer der Knoten bestimmt die Lebensdauer des Netzwerks. Dazu sind energieeffiziente Kommunikation, Berechnung, Messung und Aktuierung/Steuerung notwendig.

4. Programmierbarkeit: Durch Programmierbarkeit/Konfigurierbarkeit nach der Herstellung sind die Knoten flexibler einsetzbar.

5. Sicherheit:
   a) Zugriffskontrolle: Schutz vor unautorisiertem Zugriff auf die Knoten.
   b) Integrität: Schutz vor Manipulation der Daten.
   c) Vertraulichkeit: Schutz vor unautorisiertem Zugriff auf die Daten.
   d) Schutz vor Replay- und Man-in-the-Middle-Angriffen.

6. Bezahlbarkeit: Je teurer die einzelnen Knoten sind, desto unwahrscheinlicher ist der Praxiseinsatz des Sensornetzwerkes.

Auffällig ist hierbei, dass keine Anforderungen zur Qualität der Datenübertragung und zur korrekten Messung/Verarbeitung der Daten formuliert werden.

**Herausforderungen:**

In diesem Abschnitt werden einige der Herausforderungen, die vor allem Funkübertragung, Energieversorgung und -verbrauch betreffen, bei der Umsetzung der oben genannten (sowie weiterer) Anforderungen genannt bzw. erläutert.

Zur Sicherstellung einer korrekten Übertragung muss eine Kontrolle der Verbindungsqualität und ein darauf optimiertes Routing stattfinden [2]. Eine weitere Herausforderung ist die Ablaufplanung (Scheduling) verschiedener Aufgaben der Recheneinheit (Messen, Datenverarbeitung und Kommunikation) [1].

Zu Anforderung 1, der Ausfallsicherheit, gibt es Arbeiten, die sich mit Strategien zur Platzierung von Ersatzknoten [9] und mit der Erkennung von Störungen von Knoten bzw. ihren Sensoren [10] beschäftigen. Ebenso gibt es mechanische Probleme zu lösen: Die Zuverlässigkeit der Hardware (z. B. von Steckverbindern) kann im realen Einsatz deutlich schlechter ausfallen als im Laborexperiment. Liu et al. erwähnen den Ausfall von mehr als 20 der verbauten 5-Pin-Verbindern bei 200 eingesetzten Sensorknoten [1].

Zu Anforderung 3 (lange Lebensdauer) gehören die Energieversorgung bzw. das Energiemanagement und die Senkung des Energieverbrauchs. Zur Lösung des Energieversorgungsproblems gibt es eine Vielzahl an Arbeiten. Einige kombinieren mehrere Energiequellen (z. B. mehrere Akkumulatoren, Superkondensatoren, Solarmodule) mit intelligenter Steuerung [5]. Andere verwenden ein piezoelektrisches Bauteil als Sensor für Vibrationen und als Energiequelle [11] oder konstruieren einen Generator, der Schwingungen in elektrische Energie umwandelt, z. B. PFIG (Parametric Frequency Increased Generator) [12].

Um den Energieverbrauch der Sensorknoten zu senken, gibt es mehrere gängige Ansätze: Der offensichtlichste ist die Verwendung energiesparender Komponenten und sparsamer Module oder Transceiver zur Herstellung von Funkverbindungen, wie z. B. auf Basis des IEEE 802.15.4 Standards [13]. Ebenso kann durch Ruhephasen, also das temporäre Deaktivieren der Knoten, weniger Energie verwendet werden [1]. Dondi et al. verwenden einen Blackfin Microcontroller der Firma Analog Devices, der in zwei unterschiedlichen aktiven Modi mit dynamischer Änderung der Taktfrequenz (150 MHz oder 30 MHz) und Versorgungsspannung (0.85 V oder 1.2 V) verwendet wird [14].

Die Übertragung großer Datenmengen per Funkverbindung kostet viel Energie, deshalb kann durch eine Vorberechnung zur Verkleinerung der zu übertragenden Datenmenge Energie gespart werden, solange diese zusätzliche Berechnung nicht zu viel Energie benötigt [15]. Zur Beschleunigung von Berechnungen kann ein Koprozessor verwendet werden: Dessen Realisierung als anwendungsspezifische

integrierte Schaltung (*ASIC, Application Specific Integrated Circuit*) hat den Nachteil, dass die Sensorknoten nicht (oder nur schwer) wiederverwendbar sind (Anforderung 4). Eine flexiblere Alternative dazu ist die Verwendung eines *FPGA*s [1]. Ein FPGA (*Field Programmable Gate Array*) ist eine integrierte Schaltung, deren logische Schaltung rekonfiguriert werden kann.

## 2.2 Heterogene Zielplattform HaLOEWEn

Ein Beispiel für eine Sensorknotenarchitektur, die ein FPGA als Koprozessor verwendet, ist der in dieser Arbeit verwendete *HaLOEWE*n (*Hardware accelerated LOw Energy Wireless Embedded Sensor-Actuator node*) [16].

Engel et al. haben in einer Arbeit die Eignung des HaLOEWEn zur Strukturüberwachung untersucht. Dabei wird das FPGA verwendet, um die Menge an zu übertragenden Daten zu verringern. Dazu sind auf dem FPGA schnelle Fourier Transformation (*fast Fourier Transformation, FFT*) und Random Decrement Technique (*RDT*) implementiert [15].

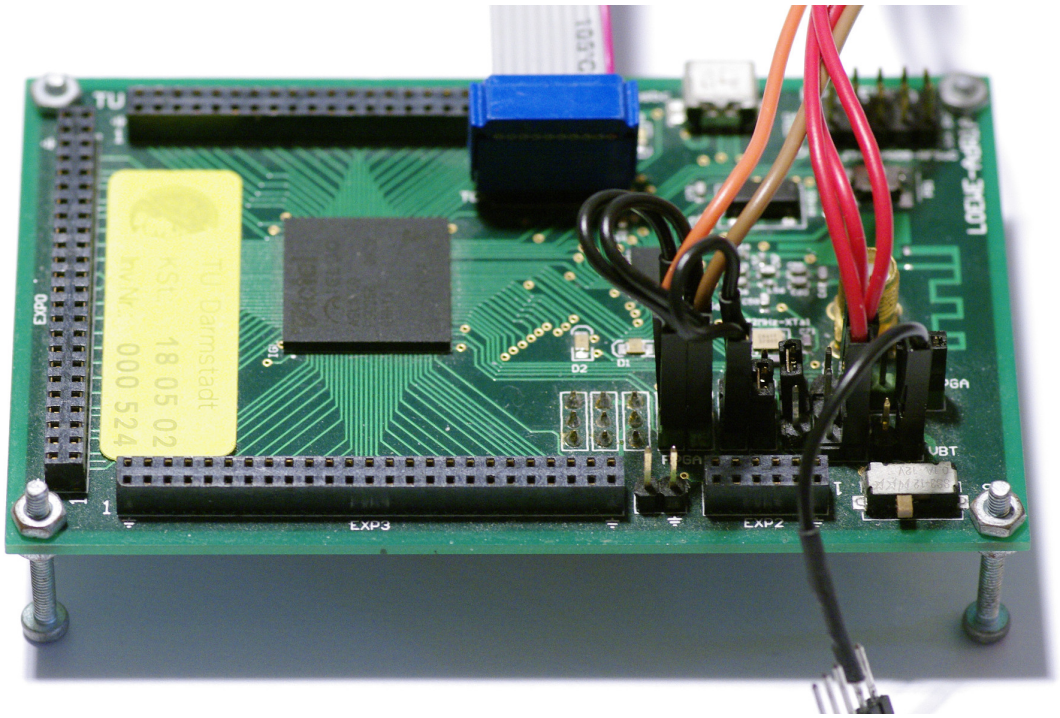


**Abbildung 2.1:** HaLOEWEn 3.

Die in dieser Arbeit verwendete Hardware entspricht der mittlerweile überholten HaLOEWEn Version 3 eines drahtlosen Sensorknotens, der aus einem Mikrocontroller mit Funkfunktionalität und einem gekoppelten FPGA zur Beschleunigung von aufwändigen Rechnungen besteht. Die veraltete Version wurde verwendet, da von der aktuellen Version in der Arbeitsgruppe nur ein Exemplar verfügbar war und dies nicht verwendet werden sollte. In Version 3 ist der Mikrocontroller ein CC2531 der Firma Texas Instruments und das FPGA ein IGLOO AGL1000V2 der Firma Microsemi. Zum Zeitpunkt des Designs wurde die Möglichkeit der Rekonfiguration des FPGAs durch den Mikrocontroller nicht vorgesehen, deshalb existieren auf der gemeinsamen Platine keine Leiterbahnen, die die benötigten Verbindungen bereitstellen. Unter Ausnutzung *aller* verfügbaren Pins und der damit verbundenen Deaktivierung aller LEDs kann über Kabelverbindungen eine JTAG-Verbindung zwischen dem in HaLOEWEn 3 (siehe Abbildung 2.1) verbauten Mikrocontroller und FPGA hergestellt werden. Der große blaue Stecker hinten im Bild ist mit dem JTAG-Sockel des FPGAs verbunden. Die bunten Kabel im Vordergrund, die zum oberen Bild-

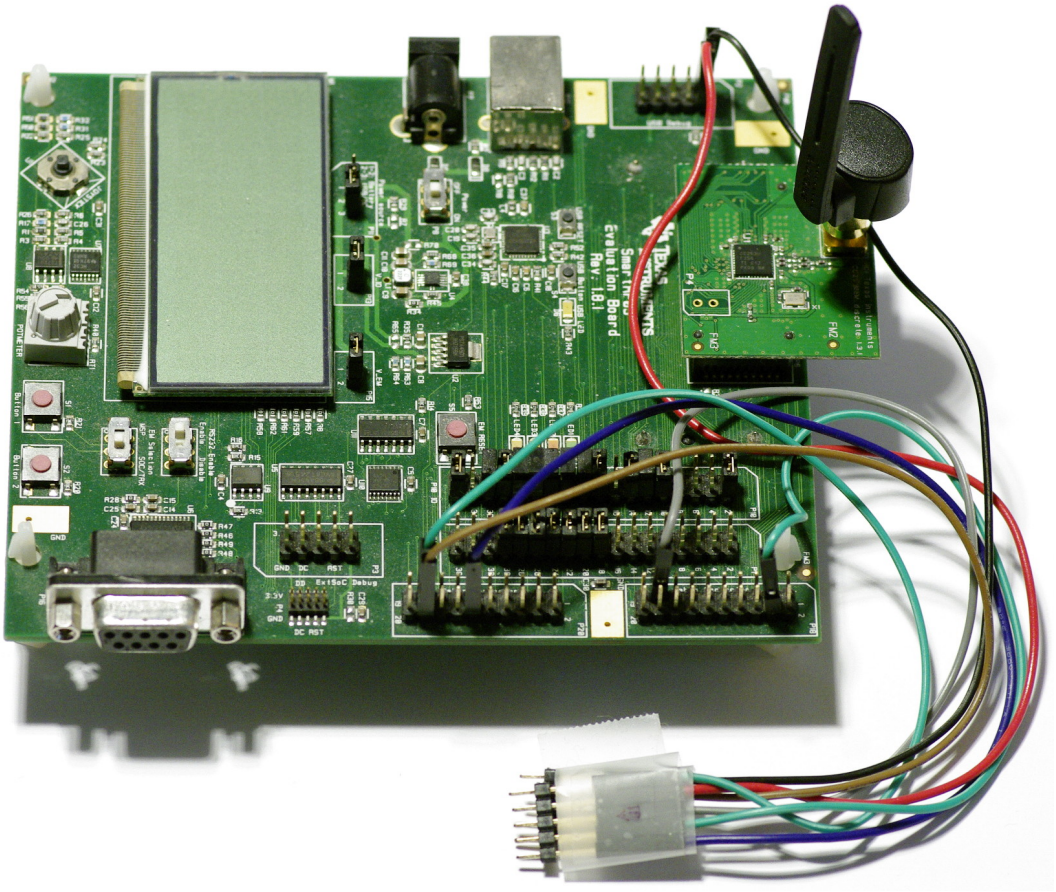

**Abbildung 2.2:** SmartRF05EB.

rand gehen, sind am Mikrocontroller zur Herstellung der JTAG-Verbindung angeschlossen. Eine genaue Beschreibung der Verkabelung inklusive einer Auflistung der verwendeten Pins folgt in Abschnitt 4.4.2.

Während dieser Arbeit werden, um die Fehlersuche während der Implementierung zu vereinfachen, deshalb, wenn nicht anders vermerkt, die Untersuchungen mit einem separaten Mikrocontroller auf einer Evaluationsplatine durchgeführt. Hierzu wurde ein SmartRF05EB Rev. 1.8.1 mit einem CC2530 von Texas Instruments verwendet. Ein Bild der Platine befindet sich in Abbildung 2.2. Dies bietet mehrere Vorteile: Die Evaluationsplatine besitzt drei LEDs, Taster und einen eingebauten Level-Shifter, der verwendet werden kann, um eine UART-Verbindung zu einem PC herzustellen. Die bunten Kabel, die zu der geklebten Verbindung vorne im Bild führen, bilden die JTAG-Verbindung auf Seite des Mikrocontrollers. Dabei sind die vom vorderen Bereich der Platine ausgehenden Leitungen mit dem Mikrocontroller verbunden. Die beiden Leitungen, die vom hinteren Bereich der Platine ausgehen, dienen zur Stromversorgung während der Konfiguration des FPGAs. Lediglich die Energiebedarfsmessungen lassen sich nicht mit der Evaluationsplatine durchführen, da es nicht möglich ist, die nicht benötigten Komponenten der Evaluationsplatine zu deaktivieren. Außerdem kann der Mikrocontroller des HaLOEWEn 3 nicht deak-

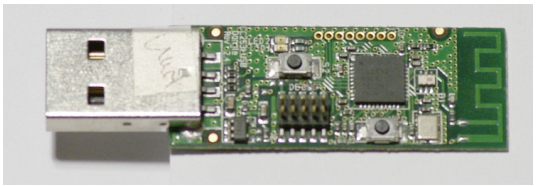

**Abbildung 2.3:** CC2531 USB.

tiviert werden, sodass eine genaue Messung nur durchgeführt werden kann, wenn der Mikrocontroller auf der HaLOEWEn-Platine benutzt wird. Die verwendete Verkabelung folgt in Abschnitt 4.4.1.

Zur Herstellung der Funk-Verbindung zum PC wird ein CC2531 USB Dongle verwendet. Ein Foto des Moduls befindet sich in Abbildung 2.3.

### 2.2.1 TI CC2530/CC2531

Der Mikrocontroller CC2530 der Firma Texas Instruments *(TI)* besteht aus einem 8051 Kern mit einigen Erweiterungen (wie z. B. DMA-Controller und AES-Koprozessor) und einem integrierten 2.4-GHz Sende-/Empfangsgerät, das IEEE 802.15.4 konform ist. Der Hersteller bewirbt den Mikrocontroller u. a. mit Robustheit durch Verwendung des 8051 Kerns und Befehlssatzes, mit geringer Leistungsaufnahme (6.5 mA bei mittlerer Rechenaktivität, 24 mA im Empfangsmodus ohne gleichzeitige Rechenaktivität, 29 mA im Sendemodus ohne gleichzeitige Rechenaktivität und 0.2 mA, 1 $\mu$A bzw. 0.4 $\mu$A je nach aktivem Stromsparmodus bei einem möglichen Versorgungsspannung zwischen 2.0 V und 3.6 V) und mit geringer Anzahl benötigter externer Bauelemente beim Aufbau von kabellosen Netzwerkknoten [17]. Während der CC2530 seit 2009 hergestellt wird, ist der im Jahr 1980 auf den Markt gebrachte 8051 der erste Mikrocontroller aus Intels MCS 51 Familie und verwendet daher einen deutlich älteren Befehlssatz [18].

Der CC2530 verfügt über 8 kByte SRAM und, je nach Ausstattungsvariante, bis zu 256 kByte programmierbaren Flash-Speicher. Er verfügt über 21 GPIO-Pins, von denen zwei typischerweise, wie auch in dieser Arbeit, für einen externen Oszillator verwendet werden. Zur freien Verwendung bleiben also lediglich 19 Pins.

Der 8051 verfügt über drei verschiedene Speicherzugriffsbusse für den Zugriff auf vier verschiedene Speicherbereiche: `SRF`, `DATA` und `CODE`/`XDATA`. Der `SRF`-Speicherbereich wird für die Hardware-Register verwendet. Der `DATA`-Speicherbereich umfasst 256 Byte. Er erlaubt Lese- und Schreibzugriff und liegt vollständig im SRAM. Die Instruktion zum Zugriff auf `SRF` und `DATA` benötigen lediglich einen CPU-Zyklus. `CODE` ist der nicht schreibbare Speicherbereich für den Code und liegt auf dem Flash-Speicher. `XDATA` ist ein größerer Addressbereich mit Lese- und Schreibzugriff. Der Zugriff erfolgt mit Instruktionen die vier bis fünf CPU-Zyklen benötigen. Über `XDATA` kann auf Teile des SRAM, des Flash-Speichers und der Hardware-Register zugegriffen werden [19].

Der ebenfalls angebotene CC2531 unterscheidet sich vom CC2530 lediglich durch eine zusätzliche USB-Funktionalität [20]. Deshalb wird im Rest der Arbeit für den Mikrocontroller, der die Rekonfiguration des FPGAs durchführt, der Begriff CC2530 verwendet, auch wenn mit dem auf dem HaLOEWEn verbauten CC2531 gearbeitet wird.

Das für die Programmierung des Mikrocontrollers benötigte Programm „SmartRF Flash Programmer“ wird von TI lediglich in einer Version für Windows bereitgestellt.

### 2.2.2 Microsemi IGLOO AGL1000

Das IGLOO AGL1000 der Firma Microsemi ist ein FPGA, das vom Hersteller vor allem mit seiner niedrigen Leistungsaufnahme beworben wird. Ursprünglich wurde er von der Firma Actel entwickelt und ab 2008 vertrieben, bis diese im November 2010 von Microsemi akquiriert wurde [21]. Das FPGA ist Flash-basiert, d. h. die Konfiguration (bzw. das Design) ist nicht flüchtig und ist deshalb direkt nach dem Anschalten verwendbar. Es kann mit Versorgungsspannungen zwischen 1.2 V und 1.5 V betrieben werden und verfügt über einen sogenannten Flash*Freeze Modus, in dem es lediglich 53 $\mu$W benötigt. Flash*Freeze ist ein Stromsparmodus, in dem das FPGA deaktiviert ist, ohne das aktuelle Design, den Inhalt des SRAMs und den Zustand der I/O-Bänke zu verlieren. Dieser Modus kann durch einen Pin aktiviert und deaktiviert werden. Der Hersteller verspricht, dass dies weniger als 1 $\mu$s dauert. Das FPGA verfügt über eine Million Gatter, 24 576 VersaTiles (D-flip-flops), 144 kBit RAM, 1024 Bit FlashROM und 4 I/O-Bänke mit (je nach Chipgehäuse) bis zu 300 Pins [22].

## 2.3 IEEE 802.15.4 konforme Funkübertragung

Ein IEEE 802.15.4 konformes Funkpaket besteht aus Kopfdaten, Nutzlast und Fußdaten. Die Fußdaten bestehen aus einer 2 Byte langen Prüfsumme. Die Länge der Kopfdaten ist variabel, beträgt aber mindestens 10 Byte. Die Kopfdaten bestehen aus 1 Byte Paketlänge, 2 Byte Einstellungen in Form eines Bitfeldes, 1 Byte Sequenznummer, 2 Byte Ziel-Pan-Identifikation, Zieladresse variabler Länge, optionaler Quell-Pan-Identifikation und Quell-Adresse variabler Länge. Falls Ziel-Pan und Quell-Pan übereinstimmen, kann nach Setzen des Einstellungsbits die Quell-Pan-Identifikation weggelassen werden. Die Quell- und die Zieladresse können in verschiedenen Formaten angegeben werden, es werden jeweils mindestens 2 Byte bei Verwendung kurzer Adressen benötigt. Die Länge der Nutzdaten ist durch die maximale Länge eines Funkpaketes von 128 Byte und die Längen der Fuß- und Kopfdaten begrenzt. Für die in dieser Arbeit gewählten Einstellungen erhält man die minimale Kopfdatenlänge und damit eine maximale Länge der Nutzdaten von 116 Byte.

Wird diese Länge überschritten, findet eine Fragmentierung statt und die Daten werden in mehreren Paketen gesendet. Im ungünstigsten Fall bedeutet dies, dass ein zweites Paket mit vollständigen Kopf- und Fußdaten und einer Nutzlast von einem Byte versendet wird, was das Verhältnis von Nutzlast zu Gesamtdaten verschlechtert und bei einer effizienten Übertragung zu vermeiden ist.

## 2.4 Ansteuerung der JTAG-Schnittstelle des FPGAs

Der Hersteller des verwendeten FPGAs, Firma Microsemi, stellt zwei generische Projekte zur Konfiguration ihrer FPGAs durch einen Mikrocontroller über die JTAG-Schnittstelle zur Verfügung: DirectC und STAPL Player. Beide unterscheiden sich grundlegend in ihrer Funktionsweise. DirectC arbeitet mit zuvor kompilierten Daten, während STAPL Player mit interpretierten Daten arbeitet. Dadurch ist DirectC laut Hersteller auf leistungsschwächeren Mikrocontrollern deutlich schneller und benötigt zudem weniger Speicher. Genaue Angaben zum Speicherbedarf des STAPL Players werden jedoch nicht gemacht. Es wird lediglich für verschiedene FPGA-Typen eine Mindestgröße des RAMs angegeben, der zur Speicherung der STAPL-Dateien benötigt wird. Als Hauptvorteil von STAPL Player wird die Unabhängigkeit vom verwendeten FPGA-Typ genannt [23]. Die Anleitung von DirectC [24] gibt für verschiedene Mikrocontroller den erforderlichen RAM- und ROM-Bedarf bei Aktivierung verschiedener Funktionalitäten genau an. Da wegen der fehlenden Angaben keine Vergleiche zu STAPL Player durchgeführt werden können und zudem der in dieser Arbeit verwendete Mikrocontroller in der Anleitung nicht berücksichtigt wurde, ist die Wiedergabe dieser Herstellerdaten an dieser Stelle nicht zweckdienlich.

Die Unabhängigkeit vom verwendeten FPGA-Typ spielt für die Zielsetzung dieser Arbeit keine Rolle, da nur mit einem festen Typ, dem Microsemi IGLOO AGL1000, gearbeitet wird. Der verfügbare Speicher ist durch den verwendeten Mikrocontroller stark begrenzt (siehe Abschnitt 2.2.1). Zudem soll die Rekonfiguration parallel zur eigentlichen Anwendung des Mikrocontrollers durchgeführt werden können, deshalb darf nur ein kleiner Teil der knappen Ressourcen verwendet werden. Aus diesen Gründen wurde statt des STAPL Players das Projekt DirectC gewählt. Im weiteren Verlauf wird, außer bei der Dateigröße der Konfigurationsdateien, nicht mehr auf STAPL Player eingegangen.

## 2.5 Speicheranforderungen und Speichermodule

**Vergleich:**

Beim Vergleich der FPGA-Konfigurationsdaten, die von DirectC und STAPL Player verwendet werden, fällt auf, dass die von DirectC verwendeten binären DAT-Dateien in ihrer unverschlüsselten Variante eine feste Größe von 936524 Bytes haben, während die Größe der interpretierten Daten (STP-Dateien) des STAPL Players abhängig von der Anzahl der verwendeten FPGA-Komponenten (Ressourcen) ist. Für alle untersuchten Konfigurationen, die in Abschnitt 4.1 genauer erläutert werden, ist die STP-Datei kleiner als

| Datei | Größe der DAT-Datei in Byte | Größe der STP-Datei in Byte |
|---|---|---|
| D1 | 936524 | 165117 |
| D2 | 936524 | 358748 |
| D3 | 936524 | 706992 |

**Tabelle 2.1:** Dateigrößenvergleich der von DirectC und STAPL Player verwendeten Formate.

die DAT-Datei (in einem Fall beträgt die Größe sogar weniger als 20 %). In Tabelle 2.1 sind die Dateigrößen von drei ausgewählter Hardware-Konfigurationen für beide Formate aufgetragen. Auch wenn die kleinere Dateigröße der STP-Dateien im Vergleich zu den DAT-Dateien ein Argument für die Verwendung des STAPL Players wäre, wenn man dessen Anforderungen an Laufzeitspeicher und Rechenleistung des Mikrocontrollers nicht beachten würde, lässt sich dieser Nachteil durch den Einsatz von Kompression, der in Abschnitt 2.7 beschrieben wird, ausgleichen.

**Analyse:**

Die Größe einer der benötigten DAT-Dateien entspricht einem Vielfachen des verfügbaren Speichers (8 KB) des CC2530. Somit kann der Mikrocontroller ohne externen Speicher nur einen kleinen Teil der benötigten Daten zwischenspeichern. Zur vollständigen Speicherung einer Konfigurationsdatei im DAT-Format sind daher ein oder mehrere Speichermodule mit einer Größe von 8 Mbit erforderlich.

Die auf dem Markt verfügbaren Speichermodule lassen sich über die Art ihrer Kommunikationsschnittstelle in zwei Kategorien aufteilen: Module mit serieller (z. B. *SPI* für *Serial Peripheral Interface*) und solche mit paralleler Anbindung. Jede dieser Kategorien kann durch die verwendeten Technologie bzw. den Speichertyp weiter unterteilt werden. Serielle Module benötigen i. a. deutlich weniger Datenleitungen, deshalb werden sie wegen der geringen Zahl verfügbarer Pins im Folgenden hauptsächlich betrachtet:

Potentiell kommen als Speichertypen u. a. SRAM, FRAM (bis 2 Mbit mit SPI verfügbar) oder MRAM (bis 4 Mbit mit SPI verfügbar) in Frage. In Tabelle 2.2 ist die maximal verfügbare Größe von seriellen Speichermodulen verschiedener Technologien mit Kenndaten einiger Beispielmodule dargestellt. Die Stromaufnahme während eines Lese- oder Schreibzugriffs und im Standbyzustand ist in den Datenblättern für unterschiedliche Spannungen (oder sogar ohne Nennung der Spannung angegeben), was den Vergleich erschwert. Zudem wird in einigen der Datenblätter keine Angabe zur minimalen Versorgungsspannung gemacht.

| Technologie | Größe | Taktrate | Stromaufnahme | | | Spannung |
|---|---|---|---|---|---|---|
| | | | Lesen | Schreiben | Standby | |
| SRAM | 1 Mbit | | 1 mA | 1 mA | 1 $\mu$A | 2,2 V |
| | | | 3 mA | 3 mA | 4 $\mu$A | 5,5 V |
| FRAM | 2 Mbit | 40 MHz | 10 mA | 10 mA | 80 $\mu$A | 3,3 V |
| MRAM | 4 Mbit | 50 MHz | 14 mA | 33 mA | 650 $\mu$A | 3,3 V |
| PCM | 128 Mbit | 33 MHz | 7 mA | 50 mA | 200 $\mu$A | 3,3 V |
| Flash | 8 Mbit | 66 MHz | 11 mA | 12 mA | 25 $\mu$A | 3,6 V |

| Technologie | Energie pro Byte | | benötigte Module | Leistung Standby | Modul |
|---|---|---|---|---|---|
| | Lesen | Schreiben | | | |
| SRAM | 0,88 mJ | 0,88 mJ | 8 | 17,6 $\mu$W | 23A1024 |
| | 6,60 mJ | 6,60 mJ | 8 | 176,0 $\mu$W | 23LC1024 |
| FRAM | 6,60 mJ | 6,60 mJ | 4 | 1056,0 $\mu$W | FM25H20 |
| MRAM | 7,39 mJ | 17,42 mJ | 2 | 4290,0 $\mu$W | MR25H40 |
| PCM | 5,09 mJ | 36,36 mJ | 1 | 600,0 $\mu$W | NP5Q128A13ESFC0E |
| Flash | 4,80 mJ | 5,24 mJ | 1 | 90,0 $\mu$W | AT45DB081D |

**Tabelle 2.2:** Übersicht über serielle Speichermodultechnologien.

Die Energie, die zum Lesen oder Schreiben eines Bytes benötigt wird, wird aus den Datenblattwerten zur Taktfrequenz, Stromaufnahme und Spannung berechnet. Analog dazu ist die Leistung aller für die Speicherung einer DAT-Datei benötigten Module für den Standbyzustand aus den gegebenen Werten berechnet. Die für die Löschung des Flash-Speichers zusätzlich benötigte Energie wurde nicht berücksichtigt. Sie würde die Energiebilanz des Flash-Speichers noch weiter verschlechtern.

Da der Vergleich lediglich auf Datenblattwerten basiert, die unterschiedlich gemessen wurden und zudem die Annahme getroffen wurde, dass es wegen der Anzahl verfügbarer Mikrocontrollerpins nicht möglich ist, die Speichermodule separat an- und auszuschalten, muss diese Übersicht natürlich mit einer gewissen Skepsis betrachtet werden. Trotzdem sind die Unterschiede in der Energieaufnahme pro Byte während des Lesens und Schreibens und der Leistung im Standbyzustand der anderen aufgeführten Technologien im Vergleich zum SRAM so groß, dass selbst mehrere parallel betriebene SRAM-Module, die länger im Betrieb sind, insgesamt einen niedrigeren Energieverbrauch als diese besitzen. Aus diesem Grund wird im Folgenden nur die Möglichkeit der Verwendung von SRAM-Modulen untersucht.

Parallele SRAM-Module benötigen eine große Zahl an Verbindungen. Ein Beispiel ist das Bauteil AS6C8008 der Firma Alliance Memory Inc., ein 1024K X 8 BIT SRAM-Modul, das über ausreichend Speicher verfügt, aber für den Gebrauch $20+8+2+2=32$ Anschlussleitungen benötigt und somit nicht direkt an den CC2530 angeschlossen werden kann.

Der zum Zeitpunkt der Recherche größte auf dem Markt verfügbare serielle SRAM (23xx1024 des Herstellers Microchip) hat eine Größe von 1 MBIT $\equiv$ 128 KB. Somit werden zur vollständigen Speicherung der DAT-Datei acht dieser Speichermodule benötigt. Zur Ansteuerung dieser Module werden je nach Art $3+7\cdot1=10$ (SCK, MISO, MOSI, je 1 CS) Verbindungen für einen sequentiellen Zugriff oder $2+7\cdot2=16$ (SCK, CS, je 1 MISO und MOSI) Verbindungen für einen parallelen Zugriff benötigt.

Der CC2530 verfügt ohne die Verwendung zusätzlicher Bauteile nicht über eine ausreichende Anzahl unbenutzter Pins, um ausreichend externen Speicher zu verbinden. In der von Microsemi zur Verfügung gestellten Version von DirectC ist ein simpler Mechanismus für die partielle Zwischenspeicherung (Paging) der DAT-Datei bereits implementiert. Wegen dieses zu erwartenden Bauteileaufwands und dem daraus resultierenden Platz- und Energiebedarfs wird in dieser Arbeit auf den Einsatz eines externen Speichers zugunsten partieller Zwischenspeicherung verzichtet. Die Verwendung des Mechanismus wird in Abschnitt 3.4 beschrieben, während große Teile von Abschnitt 3.8 dessen Optimierung thematisieren.

## 2.6 Maße für Bewertung

In dieser Arbeit werden verschiedene Maße verwendet, um die Qualität unterschiedlicher Konfigurationsvorgänge miteinander zu vergleichen. Am einfachsten zu messen ist die **Konfigurationszeit**, die die Laufzeit des gesamten Vorgangs auf dem Mikrocontroller inklusive Löschen des Speichers, Übertragung, Speichern und Verifikation umfasst. Mit ihr eng verwandt ist die verwendete **Energie**. Sie ist in den betrachteten Größenordnungen aufwändiger zu messen und beinhaltet den Energiebedarf von Mikrocontroller und FPGA während des gesamten Konfigurationsvorgangs. Die **übertragene Datenmenge** ist (wenn man den Overhead durch die Kopfdaten der Funkpakete mit einrechnet) proportional zu dem für die Datenübertragung verwendeten Anteil an Konfigurationszeit. Interessant ist auch das Verhältnis zwischen Kopfdaten und Nutzlast der einzelnen Pakete.

## 2.7 Kompression

Die Eigenschaft, dass die Größe der von DirectC verwendeten unverschlüsselten DAT-Dateien unabhängig vom zu konfigurierenden FPGA-Netz ist, suggeriert ein erhebliches Kompressionspotential. In manchen Situationen, so z. B. bei der Verwendung eines FPGAs mit einem Cortex-M1 erzwingt Microsemi eine Verschlüsselung der Konfigurationsdateien, was das Kompressionspotential weitestgehend reduziert.

Da der Dekodierungsvorgang auf dem Mikrocontroller durchgeführt werden muss, muss die Dekodierung möglichst wenig rechenintensiv sein und möglichst wenig Laufzeitspeicher benötigen. Könnte man

die kompletten Daten unter Verwendung eines externen Speichers zwischenspeichern, könnte das FPGA theoretisch verwendet werden, um den Dekodierungsvorgang zu beschleunigen. Dies hätte allerdings den Nachteil, dass die nötige Funktionalität in alle verwendeten Netze eingebaut sein müsste oder das FPGA vor dem Dekodieren von Daten rekonfiguriert werden müsste. Eine Dekodierung durch das FPGA während des Konfigurationsvorgangs ist nicht möglich, da das FPGA während des ganzen Vorgangs nicht aktiv ist.

Die Konfigurationsdaten bestehen aus mehreren Blöcken variabler Länge mit hoher Entropie, die in einer Datei fester Länge durch Blöcke getrennt sind, die weitestgehend aus Nullen bestehen, also eine niedrige Entropie besitzen. Unter den Beschränkungen für die Komplexität der Dekodierung fiel unter Berücksichtigung der Struktur der zu komprimierenden Konfigurationsdaten die Wahl auf zwei verschiedene Formen der Lauflängenkodierung und auf die Huffman-Kodierung, die im Folgenden untersucht werden.

Andere Algorithmen, deren Arbeitsprinzip darauf basiert, mehrere Byte umfassende Muster zu suchen, wie z. B. Wörterbuch- oder Prädiktor-basierte Kompression, wurden aus verschiedenen Gründen nicht berücksichtigt. Die Verwendung einer Wörterbuchkompression fällt aus, da das Wörterbuch komplett auf dem Mikrocontroller gespeichert werden müsste, was aus Laufzeitspeichergründen nicht wünschenswert ist oder bei entsprechend großem Wörterbuch nicht möglich ist. Prädiktor-basierte Kompressionsverfahren wurde nicht untersucht, da die zu komprimierenden Blöcke je nach Adresse in den Konfigurationsdaten in zwei Kategorien fallen: Die erste Kategorie umfasst Daten, bei denen eine gute Komprimierbarkeit mittels Prädiktoren erwartet wird. Gleichzeit wird erwartet, dass diese mittels Lauflängenkodierung besser komprimierbar sind. Die zweite Kategorie umfasst Daten, bei denen eine schlechte Komprimierbarkeit mittels Prädiktoren erwartet wird. In beiden Fällen bietet eine Prädiktor-basierte Kompression keinen Vorteil. Ein weiteres Problem ist die je nach Implementierung variierende Laufzeitspeicheranforderung durch die Vorhersagetabelle.

In den folgenden Beschreibungen ist mit Zeichen immer eine 8-Bit-Zahl (ein Wert von 0 bis 255) gemeint.

### 2.7.1 Lauflängenkodierung (Run Length Encoding, RLE)

Die Lauflängenkodierung auf Basis von Zeichen ist einfach zu kodieren und zu dekodieren. Während des Kodierens wird die Anzahl von aufeinanderfolgenden Wiederholungen eines Zeichens gezählt und zusammen mit dem Zeichen gespeichert. Es gibt verschiedene Konventionen, wie diese Informationen kodiert werden. Im verwendeten Fall werden immer abwechselnd die Anzahl und das wiederholte Zeichen notiert. Dadurch müssen bei der Dekodierung jeweils zwei aufeinanderfolgende Werte betrachtet werden.

Beispiel: Aus `ABCCDDDDE` wird `1A1B2C4D1E`.

An diesem Beispiel kann ein großer Nachteil der Lauflängenkodierung beobachtet werden. Obwohl Wiederholungen auftreten, ist die Länge der kodierten Nachricht genau so groß wie die der unkodierten Nachricht. Im schlimmsten Fall, bei einer Nachricht ohne Wiederholungen, kann die Lauflängenkodierung die Länge verdoppeln.

### 2.7.2 Lauflängenkodierung mit Marke (Run Length Encoding with Marker, RLEM)

Um diesen Nachteil der Lauflängenkodierung zu vermeiden, kann eine zusätzliche Marke verwendet werden. Im Idealfall ist dies ein Zeichen, das nicht (oder sehr selten) in der zu kodierenden Nachricht vorkommt. Bei der Kodierung wird entweder bei einer Sequenz von wiederholten Zeichen die Sequenz aus Marke, Anzahl von Wiederholungen und wiederholten Zeichen ausgegeben oder, falls es keine Wiederholung gibt, nur das Zeichen ausgegeben. Falls die Marke in der Nachricht vorkommt, muss, auch ohne Wiederholung, die Kodierung für Wiederholungen verwendet werden, d. h. die Sequenz „Marke,

Anzahl, Marke“. Die Dekodierung funktioniert analog zur Dekodierung bei der einfachen Lauflängenkodierung mit einer zusätzlichen Fallunterscheidung.

Beispiel: Aus `ABCCDDDDE` mit Marke `F` wird die kodierte Nachricht `ABF2CF4DE`. Verwendet man die Marke `E`, muss beim letzten Zeichen der Nachricht die Kodierungsvariante für Wiederholungen verwendet werden (`E1E`). Insgesamt ergibt sich die kodierte Nachricht `ABE2CE4DE1E`.

### 2.7.3 Huffman-Kodierung (Huffman)

Bei der Huffman-Kodierung wird unter Verwendung eines Binärbaumes jedem möglichen Zeichen der Nachricht eine Bitfolge (Codewort) unterschiedlicher Länge anhand der Verwendungshäufigkeit (bzw. der entsprechenden Entropie) zugeordnet. Die von Shannon [25] entwickelte Grundidee lautet: Je häufiger ein Zeichen ist, desto kürzer wird die zugeordnete Bitfolge gewählt, d. h. dass für das häufigste Zeichen die kürzeste Bitfolge verwendet wird. In Abbildung 2.4 ist eine beispielhafte Verteilung der Codewort-Längen aufgetragen, die für eine in Abschnitt 5.1 verwendete Huffman-Kodierung auf Basis der gemeinsamen Häufigkeitsverteilung zweier DAT-Dateien ermittelt wurde. Man sieht, dass es 19 Codewörter gibt, die kürzer als ihr zugehöriges Zeichen sind. 3 Codewörter sind genau so lang wie ihr Zeichen, während der Großteil der Codewörter (234 von 256) für selten auftretende Zeichen (deutlich) länger sind. Das Zeichen mit Codewort-Länge 1 hat eine Wahrscheinlichkeit von $\frac{1436160}{1873304} \approx 0{,}77$. Der Erwartungswert der Bitlänge pro kodiertem Zeichen beträgt $\sum_{i=0}^{255} p_i \cdot l_i \approx 2{,}17$ (mit der Wahrscheinlichkeit für das $i$-te Zeichen $p_i$und der Länge des Codewortes des $i$-ten Zeichens $l_i$).

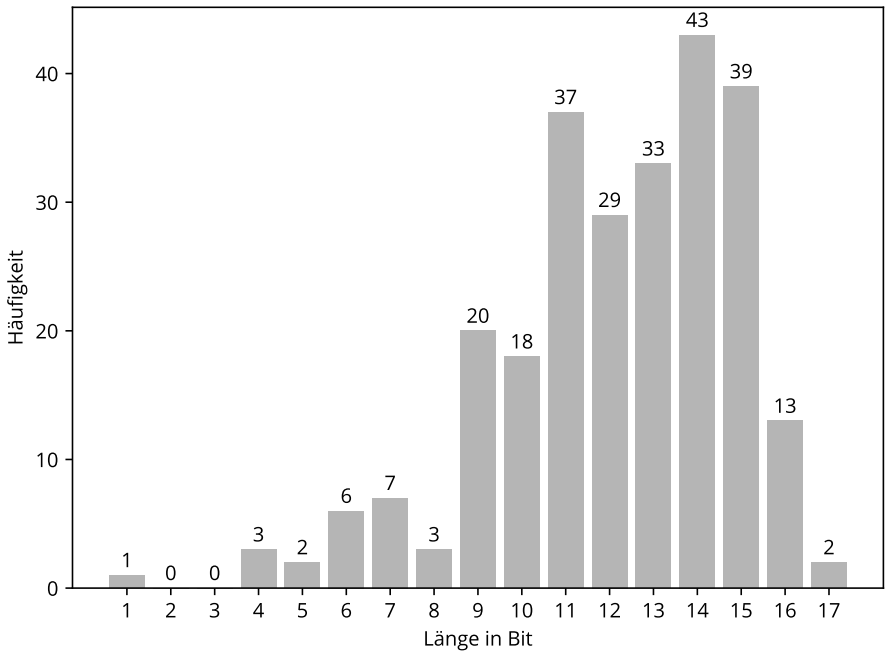


**Abbildung 2.4:** Verteilung der Längen der Codewörter am Beispiel der Huffman-Kodierung für die in Abschnitt 5.1 verwendete gemeinsame Häufigkeitsverteilung.

Der Binärbaum (Huffman-Baum) wird wie folgt erstellt: Alle möglichen Zeichen bilden die Blätter des Baums. Iterativ werden solange die beiden Blätter oder inneren Knoten mit der geringsten Wahrscheinlichkeit als Kinder eines neuen (inneren) Knotens zusammengefasst, bis alle Elemente einen Baum bilden. Dabei ist die Wahrscheinlichkeit eines inneren Knotens die Summe der Wahrscheinlichkeiten seiner Blätter [26].

Für die **Kodierung** kann wiederholtes Traversieren des Baumes durch Verwendung einer Tabelle vermieden werden, in der jedem Blatt des Baumes der Weg von der Wurzel zugeordnet wird. Bei der Ko-

dierung wird jedes Zeichen der Nachricht durch die jeweils zugeordnete Bitfolge ersetzt. Das Ergebnis wird dann wieder zu Zeichen zusammengefasst. In der verwendeten Implementierung werden dazu, falls nötig, um auf eine ganze Zahl an Bytes zu kommen, vorne an der kodierten Bitfolge Nullen ergänzt. Zusätzlich wird die Anzahl der verwendeten Bits des ersten kodierten Byte an den Anfang der kodierten Zeichenfolge geschrieben. Alternativ dazu können ans Ende der kodierten Bitfolge nicht dekodierbare Zeichen geschrieben werden. In diesem Fall kann man entweder die Dekodierung abbrechen, falls kein vollständiges Zeichen erkannt werden kann, was den Nachteil hat, dass nicht festgestellt werden kann, ob dies durch einen Fehler oder durch das Ende der Nachricht ausgelöst wird. Um dies zu vermeiden, kann die Länge der dekodierten Nachricht erfasst werden, was bei den meisten Nachrichten mehr Platz erfordert als für die Speicherung der Anzahl der gültigen Bits des ersten Bytes benötigt wird. Bei der **Dekodierung** wird die Zeichenfolge wieder als Bitfolge betrachtet. Anhand dieser Folge wird der Binärbaum traversiert, bis ein Blatt erreicht wird. Dann wird das dem Blatt zugeordnete Zeichen ausgegeben und wieder an der Wurzel begonnen.

Die Huffman-Kodierung ist optimal, wenn der verwendete Binärbaum für die Zeichen-Häufigkeitsverteilung der zu kodierenden Nachricht erstellt wurde [26]. Der zu einer Häufigkeitsverteilung gehörende Binärbaum ist allerdings nicht eindeutig, da bei der Erstellung frei zwischen Zeichen und Teilbäumen gleicher Häufigkeit gewählt werden kann.

Für die bisher verwendete Beispielnachricht `ABCCDDDDE` kann man mit der folgende Kodierung

| Zeichen | A | B | C | D | E |
|---|---|---|---|---|---|
| Bitfolge | 001 | 0000 | 01 | 1 | 0001 |

die kodierte Nachricht (der Einfachheit als Bitfolge) `001 0000 01 01 1 1 1 1 0001` erhalten. Anders gruppiert und mit fünf Nullen aufgefüllt ergibt dies `00000001 00000101 11110001`. Zusammen mit der notierten Anzahl der verwendeten Bits ist die kodierte Nachricht 4 Zeichen lang. Für die Dekodierung werden die ersten fünf Nullen übersprungen und danach wird der Baum traversiert. In Abbildung 2.5 sieht man die erste Traversierung. Ausgehend von der Wurzel wird den markierten Kanten mit den Beschriftungen `0`, `0` und `1` gefolgt, bis das Blatt mit dem Zeichen `A` erreicht wird.

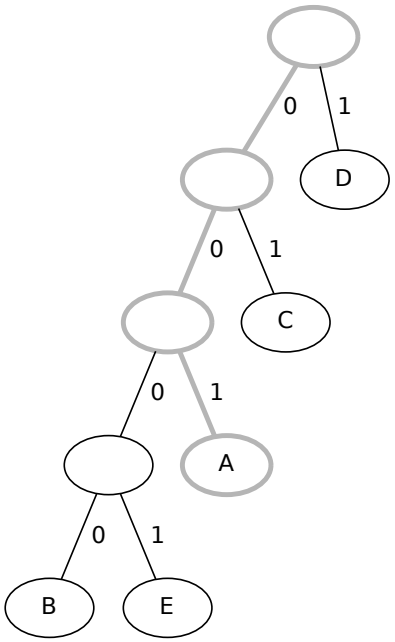


**Abbildung 2.5:** Beispielhafte Huffman-Dekodierung der Bitfolge `001` zum Zeichen `A`.

# 3 Implementierung

In diesem Kapitel wird die Kommunikation zwischen den beteiligten Komponenten für den Fall einer kabelgebundenen und kabellosen Übertragung und die Funktionsweise der einzelnen Komponenten erklärt. Der größte Teil des Kapitels befasst sich mit der Verwendung und Optimierung von DirectC.

## 3.1 Aufbau

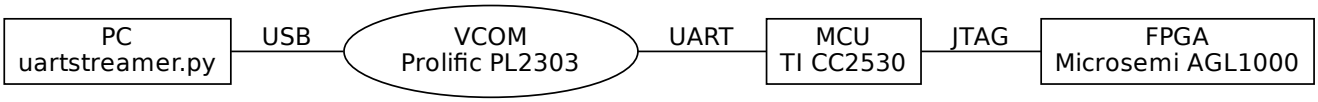


**Abbildung 3.1:** Aufbau für kabelgebundene Übertragung.

Der größte Teil der Experimente dieser Arbeit wurde sowohl kabelgebunden als auch kabellos durchgeführt. Zu Beginn hat der in Abbildung 3.1 dargestellte kabelgebundene Aufbau viele zu erledigende Arbeiten erleichtert, da die Funkübertragung als Quelle möglicher Fehler und außerdem zusätzliche Programmkomplexität vermieden werden kann. Auf dem PC läuft die in Python 3 implementierte Gegenstelle zu DirectC (`uartstreamer.py`). Diese kommuniziert per UART über einen per USB mit dem Computer verbundenen Protokoll-Converter (hier PL2303 der Firma Prolific), der als virtueller COM-Port *(VCOM)* fungiert und über ein serielles Kabel mit dem Mikrocontroller *(MCU)* verbunden ist. Die Gegenstelle wird im folgenden Abschnitt genauer erläutert. Das FPGA ist über seine JTAG-Schnittstelle mit dem Mikrocontroller, auf dem DirectC läuft, verbunden.

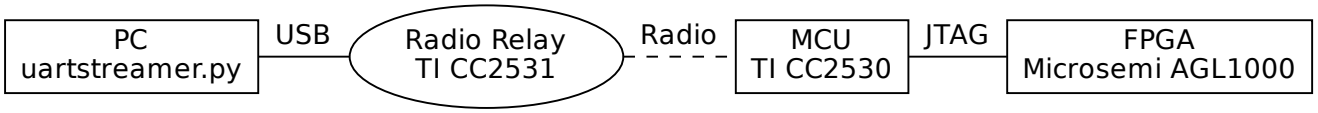


**Abbildung 3.2:** Aufbau für drahtlose Übertragung.

Beim kabellosen Aufbau, der in Abbildung 3.2 gezeigt wird, kommuniziert die Gegenstelle per UART mit einem per USB angeschlossenen Texas Instruments CC2531 Mikrocontroller *(Radio Relay)*, der als IEEE 802.15.4 Sende-/Empfangsgerät des PCs dient und im Folgenden mit Gateway bezeichnet wird. Gateway und Mikrocontroller kommunizieren über eine Funkverbindung. Das FPGA ist, wie im kabelgebundenen Fall, über seine JTAG-Schnittstelle mit dem Mikrocontroller verbunden.

## 3.2 Gegenstelle (PC)

Die Gegenstelle zu DirectC besteht aus mehreren in Python 3 geschriebenen Skripten. Für den Konfigurationsvorgang selbst ist das zentrale Skript `uartstreamer.py` verantwortlich, das die Bibliothek pySerial 3.0.1 [27] für die Kommunikation über die serielle Schnittstelle verwendet. Das Skript steuert den Ablauf des Konfigurationsvorgangs (siehe Abschnitte 3.8.4 und 3.8.6) und sendet die angefragten Segmente an den Mikrocontroller. Dabei erstellt es ein Ereignisprotokoll, in dem die wichtigsten PC-seitigen Einstellungen, die von DirectC gesendeten Statusmeldungen mit Zeitstempel und ermittelte Daten zum Konfigurationsvorgang aufgeführt werden. Zu den ermittelten Daten gehören unter anderem die Konfigurationszeit, die Menge der gesendeten und empfangenen Daten und eine Statistik über die für die Kompression der Segmente verwendeten Algorithmen. Falls mehrere Konfigurationsvorgänge

ausgeführt wurden, wird der Mittelwert und die Standardabweichung der Konfigurationszeiten ermittelt und dem Ereignisprotokoll angehängt. Die Ereignisprotokolle (`uartstreamer_logNNN.txt`) werden in einem einstellbaren Verzeichnis abgelegt und automatisch durchnummeriert. Zusätzlich wird eine Zugriffsstatistik mit der Anzahl der Zugriffe auf Adressen der Konfigurationsdatei im JSON-Format (`uartstreamer_logNNN.access`) und eine Datei mit den Vorhersagedaten, die aus Adresse und Segmentlänge bestehen, im JSON-Format (`uartstreamer_logNNN.requests`) erstellt. Die Verwendung der Vorhersagedaten wird in Abschnitt 3.8.5 genauer erläutert.

`uartstreamer.py` verfügt über eine große Zahl an Einstellungen, die entweder durch Editieren von globalen Variablen im Abschnitt Optionen am Anfang der Datei gesetzt werden können oder beim Aufruf des Skriptes per Kommandozeile übergeben werden können. Eine vollständige Beschreibung der Kommandozeilenparameter, die auch von den gesetzten globalen Optionen abhängt, kann mit dem Argument `--help` angezeigt werden. Falls in den Optionen die Verwendung von RLE aktiviert ist, kann sie beispielsweise mit dem Argument `--no-rle` deaktiviert werden. Über die Argumente --baud und --port können die verwendete Baudrate und der COM-Port bzw. unter Linux die Gerätedatei (z. B. `/dev/ttyUSB0` für kabelgebundene Übertragung oder `/dev/ttyACM0` für die kabellose Übertragung) eingestellt werden. Die zu verwendende Konfigurationsdatei kann entweder mit `--dat-no` aus einer Liste von voreingestellten Dateien ausgewählt werden oder es kann mit `--dat` der Pfad zu einer anderen Datei angegeben werden. Mit `--predictions` kann der Pfad zu einer Datei mit Vorhersagedaten übergeben werden, die statt der voreingestellten Datei verwendet wird.

Für die Verwendung der Huffman-Kodierung ist das Skript `huffman_preparation.py` wichtig. Mit ihm kann aus einer Häufigkeitsverteilung, die aus einer Liste von Konfigurationsdateien erstellt wird, ein Huffman-Baum generiert und in verschiedenen Formaten abgespeichert werden. Die verwendeten Konfigurationsdateien und die Namen der generierten Dateien können über globale Variablen am Anfang des Skriptes eingestellt werden.

## 3.3 JTAG

Obwohl die JTAG-Schnittstelle (Joint Test Action Group) in IEEE 1149.1 standardisiert ist, ist die verwendete Pinbelegung nicht festgelegt. Die HaLOEWEn-Platine verwendet die in Abbildung 3.3 dargestellte 10-Pin-Belegung, die Microsemi für den FlashPro-Adapter verwendet. `TDI` (Test Device Input, Pin 9) und `TDO` (Test Device Output, Pin 3) sind die Datenleitungen. `TCK` (Test Clock, Pin 1), `TMS` (Test Mode Select, Pin 5) und `TRST` (Test Reset, Pin 8) sind Steuerleitungen, von denen `TRST` optional ist, aber bei dieser Arbeit verwendet wird. `PROG_MODE` (Pin 4) hat für die Konfiguration des verwendeten FPGAs keine Funktion [28]. An `VPUMP` (Pin 7) muss während des Konfigurationsvorgangs eine Spannung von +3,3 V anliegen [29], diese sollte laut Microsemi von der Platine, auf der das FPGA verbaut ist, bereitgestellt werden. Da dies beim HaLOEWEn nicht der Fall ist, muss diese Spannung extern zugeführt werden. Die Versorgung von `VJTAG` (Pin 6) ist auf dem HaLOEWEn intern angeschlossen und muss deshalb bei der Verkabelung nicht beachtet werden. `GND` (Pin 2 und Pin 10) ist die Erdung.

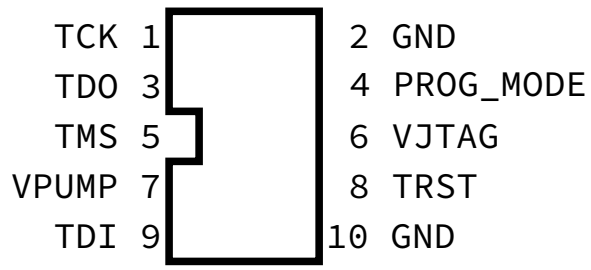


**Abbildung 3.3:** JTAG-Pinbelegung nach [28].

Am Mikrocontroller werden TDI, TCK, TMS und TRST als Ausgänge und TDO als Eingang angeschlossen. Zudem muss VPUMP mit der korrekten Spannung versorgt werden und über eine der GND-Leitungen eine Erdungsverbindung hergestellt werden.

Wie man an den verwendeten Datenleitungen TDI und TDO erkennen kann, findet die Übertragung über die JTAG-Schnittstelle seriell statt. Es ist prinzipiell möglich, mehrere Geräte gleichzeitig anzusteuern. Dabei werden die Geräte über TDO und TDI verkettet, d. h. TDO eines Gerätes wird mit TDI des nächsten verbunden. Die restlichen Verbindungen werden für alle Geräte geteilt. Da in dieser Arbeit zu jedem Zeitpunkt nur ein Gerät rekonfiguriert wird, kann der große Vorteil der JTAG-Schnittstelle nicht genutzt werden, jedoch kommt der Nachteil eines seriellen Protokolls, nämlich die im Vergleich zu einer parallelen Ansteuerung niedrigeren Übertragungsrate, zum Tragen.

## 3.4 Verwendung von DirectC

Um DirectC in ein Projekt, wie z. B. zur Nutzung auf der HaLOEWEn-Architektur, einzubinden, müssen einige in der Anleitung [24] beschriebene Anpassungen vorgenommen werden. Zunächst müssen die für die JTAG-Kommunikation verwendeten Pins in Form von Bitmasken definiert werden. Jedoch wird in der Anleitung nicht explizit beschrieben, ob die Pins als Pin-Nummern oder als Bitmasken definiert werden müssen. Die Entwickler haben dabei die Annahme getroffen, dass alle Pins auf der gleichen I/O-Bank des Mikrocontrollers liegen und somit alle Daten als Byte übergeben werden können. Dies ist bei der Verwendung eines CC2530 nicht möglich, daher besteht die einfachste Lösung dieses Problems darin, die Verwendung einer einzigen Bank vorzutäuschen, indem beliebige, paarweise unterschiedliche Bitmasken für die einzelnen Pins definiert werden. Alternativ dazu müsste man viele Änderungen an DirectC vornehmen. Weiter müssen eine Reihe von Funktionen angepasst bzw. implementiert werden: Eine Ein- und eine Ausgabefunktion, eine Verzögerungsfunktion, eine Segmentanforderungsfunktion dp_get_page_data, die segmentweise (seitenweise) die Konfigurationsdaten in den benötigten Puffer lädt, und (optional) die Funktionen zur Anzeige der Statusmeldungen. Bei der Implementierung der Ein- und Ausgabefunktion, die den Zustand des Eingabepins lesen bzw. der Ausgabepins schreiben, kann die zuvor willkürlich getroffene Wahl der Bitmasken bzw. Pins einer imaginären I/O-Bank leicht kompensiert werden. Bei der Implementierung der Verzögerungsfunktion ist es wichtig, die mindestens geforderte Zeit einzuhalten. Eine zu lange Wartezeit ist für die korrekte Funktion des Konfigurationsvorgangs nicht störend, kann diesen allerdings drastisch verlängern. Die Anforderungsfunktion bekommt als Argument eine Adresse innerhalb der Konfigurationsdaten und fragt dann über die in Abschnitt 3.1 beschriebene Verbindung die benötigten Daten beim PC an. Nach Erhalt der Daten werden diese in den Puffer geschrieben und der Konfigurationsvorgang fortgesetzt. Die Verwendung der Anforderungsfunktion und die Arbeit der Anforderungsfunktion bieten einige Verbesserungsmöglichkeiten, deshalb sind sie das hauptsächliche Thema von Abschnitt 3.8.

Nach Durchführung dieser Anpassung kann DirectC durch Setzen einer globalen Variable, die den Operationsmodus wie beispielsweise PROGRAM oder VERIFY festlegt, und den Aufruf der zentralen Funktion dp_top verwendet werden.

Bei der Entwicklung von DirectC wurde eine Vielzahl von Präprozessordirektiven verwendet, mit denen große Teile des Programmcodes deaktiviert werden können, um den Speicherbedarf (DATA, XDATA und CODE) von DirectC drastisch zu senken.

Für den in dieser Arbeit verwendeten Aufbau, das verwendete FPGA und die davon benötigten Komponenten braucht man folgende DirectC-Funktionalität nicht: Die Konfiguration mehrerer FPGAs gleichzeitig (CHAIN_SUPPORT), die Unterstützung für andere FPGA-Typen (ENABLE_G4_SUPPORT, NVM_SUPPORT, NVM_PLAIN, NVM_ENCRYPT, SILSIG_SUPPORT, ENABLE_DAS_SUPPORT), die Möglichkeit zur Verwendung von verschlüsselten Konfigurationsdaten (SECURITY_SUPPORT, CORE_ENCRYPT, FROM_ENCRYPT) und die Möglichkeit der Verwendung anderer Verkabelungsarten (ENABLE_IAP_SUPPORT). Die folgende Funktionalität wird definitiv benötigt bzw. gewünscht: Die Unterstützung von partiellem Zugriff auf die Konfigurationsdaten (USE_PAGING), die Unterstützung des verwendeten FPGAs (ENABLE_G3_SUPPORT,

CORE_PLAIN) und einige zusätzliche Optimierungen auf Kosten der Unterstützung anderer FPGAs (ENABLE_CODE_SPACE_OPTIMIZATION). Zudem existieren weitere Präprozessordirektiven zur Deaktivierung von unnötigen Funktionalitäten, die aber durch die vorher gesetzten Optionen keine Auswirkung haben: DISABLE_SEC_SPECIFIC_ACTIONS hat Wechselwirkungen mit SECURITY_SUPPORT und DISABLE_NVM_SPECIFIC_ACTIONS hat Wechselwirkungen mit NVM_SUPPORT.

Dadurch können grundsätzlich ca. 24 Kilobyte im CODE-Segment und ca. 100 Byte im DATA- und XDATA-Segment eingespart werden. Um die Übersichtlichkeit des Quelltextes zu verbessern, wurden mit Coan (The C Preprocessor Chainsaw. Version 6.0.1. [30]) einige dieser Präprozessordirektiven auf einen festen Wert gesetzt und dadurch nicht erreichbare Abschnitte automatisch entfernt. Nach der automatischen Bearbeitung sind einige Dateien leer, diese wurden per Hand entfernt.

Die Funktionalität zum unverschlüsselten Schreiben des FROMs ist aktiviert, wird aber bei den verwendeten Tests nicht genutzt. Die zur Deaktivierung des FROM-Schreibens erforderlichen Präprozessordirektiven wurden nicht wegoptimiert, sodass sie mit geringem Aufwand verwendet werden könnten.

## 3.5 Nachrichtenformate zur Kommunikation zwischen PC und Mikrocontroller

Die Kommunikation zwischen PC und Mikrocontroller findet als Sequenz von Byte-Werten statt, die eine für diese Arbeit erstellte Nachrichtenformate verwenden. Alle möglichen Nachrichten beginnen mit einer eindeutigen Marke. Je nach Typ der Nachricht folgen darauf benötigte Kopfdaten wie Länge und Adresse sowie Nutzdaten. In diesem Abschnitt folgt eine Beschreibung einiger wichtiger Nachrichtentypen. Im Anhang befindet sich eine vollständige Beschreibung der definierten Nachrichten. Damit die Nachrichten bei der Fehlersuche für den Benutzer einfach lesbar sind, werden alle Werte im Big-Endian-Format kodiert. Die hier genannten Beispiele enthalten Vorgriffe auf vorgenommene Optimierungen, die im folgenden Abschnitt vorgestellt werden.

Zur Anforderung eines neuen Segments der Konfigurationsdatei sendet der Mikrocontroller eine Nachricht, die aus der Marke 0x04, drei Byte Adresse und einem Byte Segmentlänge besteht. Für die Adresse 0xABCDEF und die Läge 0xF4 ergibt sich beispielsweise die Nachricht 0x04ABCDEFF4. Zur Anforderung des nächsten vorhergesagten Segments werden keine Adresse und Länge benötigt; die Nachricht besteht lediglich aus der Marke 0x14.

Für das Senden der Segmente wird eine größere Zahl möglicher Marken verwendet. Dies ergibt sich aus der Kombination der möglichen Kompressionsmethoden (unkomprimiert, RLE, RLEM, Huffman-Kodierung) und der Unterscheidung nicht vorhergesagter und vorhergesagter (siehe Abschnitt 3.8) Segmente. Nachrichten für nicht vorhergesagte Segmente bestehen aus der Marke, einem Byte Länge und dem Segment. Bei Verwendung von Huffman-Kodierung, wie in Abschnitt 2.7 beschrieben, wird zusätzlich ein Byte zwischen Länge und Segment zur Übertragung der Anzahl der verwendeten Bits des ersten Byte der kodierten Daten verwendet. Dieses zusätzliche Byte wird bei der Berechnung der Länge mitgezählt. Bei der Übertragung von vorhergesagten Segmenten muss die Adresse übertragen werden. Die Adressinformationen werden als drei Byte, die zur Adressierung der 936524 Byte der DAT-Dateien ausreichen, nach der Längeninformation eingefügt. Ein Huffman-kodiertes Segment ohne Vorhersage mit Länge 0x07 wird beispielsweise als 0x0807020123456789AB übertragen, während eine unkomprimiertes Segment mit Vorhersage mit Länge 0x06 und Adresse 0xABCDEF als 0x1806ABCDEF0123456789AB übertragen wird.

Statusmeldungen in Textform werden nach der in Abschnitt 3.8.7 beschriebenen Optimierung unter Verwendung einer Wörterbuchkompression als zwei Byte Nachrichten kodiert. Sie bestehen aus der Marke 0x10 und einem weiteren Byte für die Nachrichtennummer. Für die Meldung „Verifying FPGA Array...“ ergibt sich die Nachricht 0x100B.

Zum Start des Konfigurationsvorgangs sendet der PC eine Nachricht aus einer zwei Byte Marke 0x6060 an den Mikrocontroller. Wenn der Konfigurationsvorgang unabhängig von den Einstellungen des Mikrocontrollers ohne Vorhersage erfolgen soll, kann die Marke 0x6061 verwendet werden, um die Deaktivierung zu erzwingen.

## 3.6 Funktionsweise des Gateways

Das Gateway, also der als IEEE 802.15.4 Sende- und Empfangsgerät dienende CC2531 Mikrocontroller hat die Aufgabe, Funkpakete in UART-Daten zu wandeln und umgekehrt. Wird ein Funkpaket empfangen, kann es direkt per UART ausgegeben werden. Die Gegenrichtung ist etwas aufwändiger: Nach Empfang von einigen UART-Daten werden je nach möglichem Datentyp solange weiter Daten gesammelt, bis die Nachricht vollständig ist. Bei Nachrichten mit fester Länge muss diese in der Firmware des Gateways hinterlegt werden. Bei Nachrichten mit variabler Länge muss analog dazu das Auslesen der Länge aus der Nachricht implementiert werden. Wird dies nach Hinzufügen von neuen Nachrichten vergessen, können Fehler auftreten, deren Ursache schwer zu finden sind.

## 3.7 Automatisierung der Messungen

Um die automatisierte Durchführung von Testreihen zu ermöglichen, musste dafür gesorgt werden, dass die Gegenstelle auf dem PC die Möglichkeit hat, einige der im folgenden Abschnitt vorgestellten Optimierungen zu deaktivieren. Zudem muss auf dem Mikrocontroller DirectC nach Abschluss eines Konfigurationsvorgangs in den Ursprungszustand zurückgesetzt werden, damit im Anschluss ohne Benutzereingriff eine weitere Messung vorgenommen werden kann. Dabei wird der Softreset des Mikrocontrollers nicht verwendet. Die relevanten Variablen werden manuell auf Werte gesetzt, die einen erneuten Konfigurationsvorgang erlauben.

## 3.8 Optimierungen

Der grundlegende Ablauf von DirectC während des Konfigurationsvorgangs umfasst mehrere Schritte:

1. Überprüfung des Typs des angeschlossenen FPGAs
2. Prüfsummenberechnung *(CRC)* und Vergleich
3. Auslesen weiterer FPGA-Daten
4. Löschen des Flash-Speichers
5. Schreiben des Flash-Speichers
6. Verifizieren des Flash-Speichers

Da die vollständigen Konfigurationsdaten nicht vom Mikroprozessor zwischengespeichert werden können und DirectC im Ursprungszustand ineffizient arbeitet, was aber bei nicht-lokaler Speicherung nicht vermeidbar ist, wird bei der Durchführung der Schritte 1, 5 und 6 jeweils mindestens einmal die komplette Konfigurationsdatei übertragen. Dies bietet erhebliches Optimierungspotential, da für die essentielle Durchführung lediglich eine Übertragung für Schritt 5 und eine für Schritt 6 erforderlich ist.

Die folgenden Abschnitte 3.8.1 bis 3.8.3 beschäftigen sich mit der Verringerung der vom PC zum Mikrocontroller gesendeten Datenmenge. Der Ablauf des Konfigurationsvorgangs unter Betrachtung der Kommunikation wird in Abschnitt 3.8.4 erklärt. In Abschnitt 3.8.5 wird eine Strategie zur Vermeidung von Wartephasen des Mikrocontrollers vorgestellt. Der daraus folgende geänderte Ablauf des Konfigurationsvorgangs wird in Abschnitt 3.8.6 unter Berücksichtigung eventueller Probleme dargestellt. In Abschnitt 3.8.7 wird die Kommunikation in Rückrichtung, d. h. vom Mikrocontroller zum PC untersucht. Danach werden in Abschnitt 3.8.8 kleinere Änderungen an DirectC erläutert, die Fehler beheben oder die Größe des Programms des Mikrocontrollers senken. Zuletzt wird in Abschnitt 3.8.9 eine aus den Beobachtungen resultierende, vermeintlich „optimale Strategie“ zur Verwendung von Kompression und Segmentvorhersage vorgestellt.

### 3.8.1 Verlagerung der Prüfsummenberechnung

Die Durchführung der Prüfsummenberechnung auf dem Mikrocontroller hat zwei Nachteile: Die Berechnung ist dank der 8-Bit-Architektur extrem langsam und zudem ist eine weitere Übertragung der Konfigurationsdaten notwendig. Dadurch ergibt sich sogar bei kabelgebundener Übertragung eine Verzögerung von ungefähr 90 s.

Wenn die Prüfsummenberechnung in Python vor der eigentlichen Übertragung auf dem PC mit um mehrere Größenordnungen höherer Rechenleistung durchgeführt wird, fällt diese dritte Übertragung weg. Dazu wurde der verwendete Algorithmus und eine Aufschlüsselung der Kopfdaten zum Auslesen der in der Konfigurationsdatei abgelegten Prüfsumme in Python implementiert. Ein weiterer Vorteil ist, dass hierdurch die analoge Funktionalität auf dem Mikrocontroller entfernt werden kann und somit weniger Speicherkapazität benötigt wird.

Prinzipiell bietet die Prüfsummenberechnung auf dem Mikrocontroller eine stärkere Kontrolle der korrekten Übertragung. Da in Schritt 1 Einträge der Kopfdaten der Konfigurationsdateien, die bei der Prüfsummenberechnung auf dem PC verifiziert wurden, mit den vom FPGA ausgelesenen Daten verglichen werden und zusätzlich zum Speichern in Schritt 5 die Verifikation in Schritt 6 durchgeführt wird, ist auch bei diesem geänderten Ablauf immer noch genügend Kontrolle über die Qualität der Übertragung sichergestellt. Die andere wichtige Aufgabe der Prüfsummenberechnung besteht darin, sicherzustellen, dass vom Benutzer eine gültige Konfigurationsdatei übergeben wurde. Zur Erfüllung dieser Aufgabe ist es nicht notwendig, dass die Überprüfung auf dem Mikrocontroller erfolgt.

Die Funktion zur Überprüfung der Prüfsumme in DirectC zeigt eine kuriose Ineffektivität: Vor der Berechnung der Prüfsumme wird die erwartete Prüfsumme aus den Konfigurationsdaten gelesen und in einer Variable abgelegt. Nach der Berechnung wird die berechnete Prüfsumme mit dem Wert der Variable verglichen. Falls die beiden Werte nicht übereinstimmen, wird für die Fehlermeldung die erwartete Prüfsumme erneut aus den Konfigurationsdaten gelesen. Zusätzlich ist die angezeigte Beschriftung vertauscht. DirectC bezeichnet die berechnete Prüfsumme als die erwartete Prüfsumme und umgekehrt. Nach der Verlagerung der Prüfsummenberechnung auf den PC sind die problematischen Stellen inaktiv.

### 3.8.2 Kompression

Der Zeitaufwand einer Übertragung mit begrenzter Bandbreite ist, unter Vernachlässigung von eventuell auftretendem Overhead durch Kopfdaten und anderer durch das verwendete Protokoll vorgegebener zusätzlicher Nachrichten, proportional zu der zu übertragenden Datenmenge. Für die Konfiguration des FPGAs mit segmentweiser Übertragung der Konfigurationsdaten bedeutet dies, dass die für den Konfigurationsvorgang benötigte Zeit verringert werden kann, indem das Volumen der übertragenen Daten verringert wird, solange die Laufzeit der dafür notwendigen zusätzlichen Berechnungen (vor allem bei der Dekompression) diese Einsparungen nicht überschreitet.

**Feste Rohdatenmenge:**

Zunächst wird die Kompression von Segmenten fester Größe betrachtet, auch wenn dabei u. U. die Nutzlast der resultierenden Funkpakete nicht vollständig ausgenutzt wird.

Die Komprimierung der Seiten auf dem PC wird analog zu der in Abschnitt 5.1 beschriebenen Strategie zur Wahl einer optimalen Kompression, die bei Voruntersuchungen als vielversprechend ermittelt wurde, für jedes Segment vorgenommen. Im einfachsten Fall wird mit einer festen Länge an Rohdaten gearbeitet. Jede der aktivierten Kompressionsmethoden wird verwendet, um die Rohdaten zu komprimieren. Danach wird die Länge der komprimierten Daten berechnet, dabei wird bei der Huffman-Kodierung das zusätzliche Byte, das die Anzahl der verwendeten Bits des ersten kodierten Bytes angibt, mitgezählt. Die Methoden werden in der Reihenfolge unkomprimiert, RLE, RLEM und Huffman-Kodierung mit dem bisherigen besten Ergebnis verglichen. Dabei ist eine Methode besser als die vorherige, wenn die Länge der komprimierten Daten echt kleiner als die der verglichenen Methode ist. Dadurch wird sichergestellt,

dass bei gleich guter Kompression die Methode mit geringerem Dekodierungsaufwand präferiert wird. Für die Huffman-Kodierung wird ein fester Baum verwendet, der aus der gemeinsamen Häufigkeitsverteilung von D1 und D2 (siehe Abschnitt 4.1) erstellt wurde und im Quelltext in Form von zwei Arrays hinterlegt ist und somit nicht separat übertragen wird.

Das Dekodieren der komprimierten Segmente auf dem Mikrocontroller erfolgt aus Speicherspargründen so, dass die im Empfangspuffer abgespeicherte Nachricht direkt gelesen und in dekodierter Form in den von DirectC zur Speicherung der Segmente verwendeten Puffer geschrieben wird. Dieser Empfangspuffer wird bei kabelloser Übertragung verwendet, um erfolgreich empfangene Pakete aus dem FIFO zwischenzuspeichern. Bei kabelgebundener Übertragung per UART werden die empfangenen Daten ebenso direkt in diesen Puffer geschrieben. Damit ist klar, dass die Dekodierung nur stattfinden kann, wenn DirectC den vorherigen Inhalt des Puffers nicht mehr benötigt. Gleichzeitig benötigen die komprimierten Daten Platz im Empfangspuffer, was dazu führen kann, dass keine weiteren Daten empfangen werden können. Die Wahl der korrekten Dekodierungsmethode richtet sich nach der Marke der empfangenen Nachricht statt. Die Implementierung der RLE- und RLEM-Dekodierung sowie auf die Behandlung unkomprimierter Segmente wird im Folgenden nicht näher beschrieben.

Zur Dekodierung der Huffman-kodierten Segmente wird der verwendete Baum benötigt. Um den verwendeten Huffman-Baum im laufenden Betrieb beispielsweise vor jedem Konfigurationsvorgang austauschen zu können, sollte er idealerweise in einem kompakten Format vorliegen, das somit effizient übertragen werden kann und auch ein effizientes Dekodieren durch Traversieren des Baumes ermöglicht. Theoretisch könnte ein effizientes Dekodieren auch durch eine Umstrukturierung nach dem Empfang ermöglicht werden, dies benötigt jedoch zusätzlichen Speicher und verursacht zusätzlichen Rechenaufwand. In diesem Fall also als Sequenz von Byte-Werten. Der benötigte Binärbaum hat für jeden möglichen Wert eines Byte ein Blatt, d. h. 256 Blätter. Alle inneren Knoten eines Huffman-Baums, d. h. solche, die einschließlich der Wurzel keine Blätter sind, haben genau zwei Kinder. Daraus ergibt sich, dass die möglichen Huffman-Bäume 255 innere Knoten haben. Dies ermöglicht es, den Baum als eine Folge von Tupeln, die den inneren Knoten des Baumes entsprechen, abzuspeichern. Die Tupel werden mit Null für die Wurzel beginnend aufsteigend nummeriert. Diese Nummern sind alle echt kleiner als 256 und können somit als ein Byte gespeichert werden. Die Einträge der einzelnen Tupel entsprechen jeweils den ausgehenden Kanten jedes einzelnen Knotens. Zeigt die Kante auf einen inneren Knoten, wird die Nummer des entsprechenden Tupels abgelegt. Zeigt sie auf ein Blatt, wird der dem Blatt zugeordnete Wert abgelegt. Für jeden Eintrag muss zudem ein Bit verwendet werden, um zu markieren, ob es sich um einen Verweis auf ein anderes Tupel oder um einen Ausgabewert handelt. In den inneren Knoten ist das jeweilige Tupel aufgetragen. Die Tupel werden in der durch die Nummerierung vorgegebenen Reihenfolge aneinandergehängt und bilden einen Array aus 510 vorzeichenlosen Byte-Werten. Die 510 Bitwerte zur Unterscheidung zwischen Blättern und inneren Knoten werden zu einem zweiten Array aus 64 vorzeichenlosen Byte-Werten zusammengefasst. Zur Speicherung des Huffman-Baums sind daher insgesamt 574 Byte notwendig.

In Abbildung 3.4 ist diese Umformung am Beispiel des in Abschnitt 2.7 verwendeten Beispiels dargestellt. Man erhält `1, 'D', 2, 'C', 3, 'A', 'B', 'E'` und der Übersichtlichkeit halber als Binärwerte `0, 1, 0, 1, 0, 1, 1, 1`.

Die durch diese Kodierung gegebene Möglichkeit, den verwendeten Huffman-Baum auszutauschen, wurde im Laufe dieser Arbeit nicht verwendet und es wurden auch keine Nachrichten zur Übertragung der Bäume implementiert, da sich herausgestellt hat, dass dies unnötig ist (siehe auch Abschnitt 5.1). Die für den Austausch des Baums notwendigen Änderungen sind mit geringem Aufwand möglich.

**Variable Rohdatenmenge:**

Wie in Abschnitt 2.3 berechnet, beträgt die maximale Nutzdatenlänge 116 Byte. Durch die Kodierung der Nachrichten werden für vorhergesagte Segmente 5 Byte Overhead und für nicht vorhergesagte Segmente 2 Byte Overhead erzeugt. Prinzipiell könnte davon ein Byte (der Längeneintrag) für die Funkübertragung entfernt werden, da diese Längeninformation aus den Kopfdaten des Funkpaketes berechnet

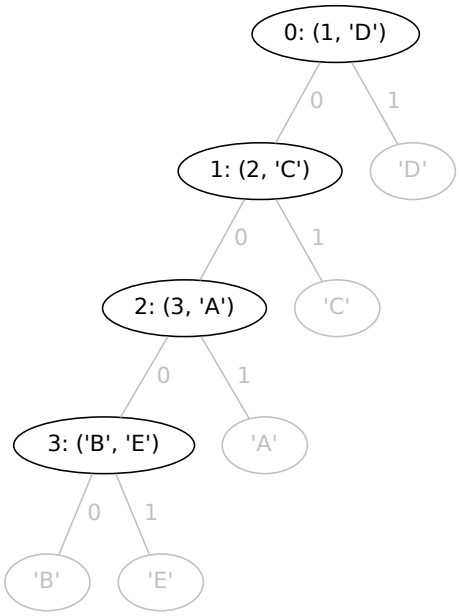


**Abbildung 3.4:** Umformung des Huffman-Baums aus Abbildung 2.5 in Tupel.

werden kann. Dazu wären weitere Fallunterscheidungen bei der Dekodierung notwendig, da dies für die zu Vergleichszwecken durchgeführt kabelgebundene und damit paketfreie Übertragung nicht möglich ist. Verzichtet man auf diese Unterscheidungen und nimmt den ungünstigsten Fall, also die Übertragung mit Vorhersage, an, bleiben 111 Byte für die (komprimierten) Segmentdaten übrig.

Das Verhältnis von Nutzdaten zu Overhead der Funkpakete ist am besten, wenn diese 111 Byte vollständig genutzt werden. Um dies zu erreichen, müssen die Kompressionsalgorithmen angepasst werden. Statt einer festen Datenmenge, die so gewählt ist, dass sie unkomprimiert in einem einzelnen Funkpaket übertragen werden könnte, wird eine variable Menge an Eingangsdaten verwendet. Beim Komprimieren wird die Größe der verwendeten Rohdaten und die Größe der komprimierten Daten in Byte erfasst. Vor dem Anhängen des jeweiligen Blocks wird kontrolliert, ob danach die angepeilte Größe der komprimierten Daten überschritten würde. Ist dies der Fall, werden die Daten nicht angehängt und der jeweilige Kompressionsvorgang wird abgebrochen. Bei der Huffman-Kodierung kann analog dazu die Anzahl der verwendeten Bits gezählt werden. Dies wurde in dieser Arbeit nicht implementiert, da sich zuvor gezeigt hat, dass die Nutzung der Huffman-Kodierung keine Vorteile hinsichtlich der Konfigurationszeit bringt. Nachdem die Resultate aller verwendeten Kompressionsverfahren vorliegen, wird anhand der Präferenzliste entschieden. Es wird das Kompressionsverfahren verwendet, bei dem die meisten Rohdaten komprimiert wurden. Falls keines der Verfahren mehr Rohdaten als das eingestellte Limit verarbeitet hat, wird wie im Fall mit fester Rohdatenmenge entschieden. Abhängig vom Kompressionsverfahren gibt es verschiedene Gründe, aus denen eine eingestellte Beschränkung (d. h. die maximale Nutzdatenmenge für das Funkpaket) nicht erreicht werden kann. Mit RLE komprimierte Daten bestehen aus z. B. Tupeln, daher ist die Größe eine gerade Anzahl an Byte und eine ungerade Beschränkung kann nicht erreicht werden. Im Gegensatz dazu können mit RLEM oder Huffman-Kodierung komprimierte Daten bei geeigneten Eingangsdaten beliebige Beschränkungen exakt erreichen.

Da die dekodierten Daten vollständig in den von DirectC verwendeten Puffer passen sollen und der Speicher des Mikrocontroller stark begrenzt ist, ist es sinnvoll, die maximale Menge der verwendeten Rohdaten zu begrenzen.

### 3.8.3 Verringerung der Anzahl an Anfragen

**Vermeidung überlappender Anfragen:**

Die Funktionalität von DirectC, die zur segmentweisen Übertragung der Konfigurationsdaten vorgesehen ist, sorgt dafür, dass bei jedem Zugriff auf eine Adresse der Konfigurationsdaten mindestens 16 Byte ab dieser Adresse im lokalen Segment-Puffer sind. Dies scheint auf den ersten Blick eine gute Strategie zu sein, wenn im Hintergrund übertragen würde und insbesondere die im Cache verbleibenden Daten genutzt würden. Da dies nicht der Fall ist, verursacht diese Strategie zwei Probleme: überlappende Anfragen am Anfang und in der Mitte der Konfigurationsdaten sowie überflüssige Anfragen am Ende der Konfigurationsdaten.

Sobald auf eines der letzten 16 Byte des Puffers zugegriffen werden soll, wird ein neues Segment angefordert. Der Zugriff erfolgt erst, wenn das neue Segment empfangen wurde und der Inhalt des Puffers neu geschrieben wurde. Somit ist der Puffer für DirectC effektiv um 16 Byte verkleinert und dieser Teil der Daten wird sinnlos übertragen.

Während des sequentiellen Zugriffs auf Adressen am Ende der Konfigurationsdaten führt dies zu vielen unnötigen Anfragen. Alle Zugriffe durch DirectC auf den Puffer finden blockweise auf 1 bis 4 Byte statt. Während des Zugriffs auf die letzten 16 Byte der Konfigurationsdaten wird zuerst festgestellt, dass nicht genug Daten im Puffer vorhanden sind, dann wird ein neuer Pufferinhalt angefordert. Nachdem dieser Pufferinhalt, der weniger als 16 Byte ausmacht, vorhanden ist, wird der Zugriff durchgeführt. Beim Zugriff auf folgende Adresse stellt DirectC fest, dass wieder zu wenig Inhalt im Puffer ist und stellt eine weitere Anfrage. Dies wiederholt sich, bis auf eine Adresse zugegriffen wird, die mehr als 16 Byte vom Dateiende entfernt ist. Ein Beispiel für dieses Verhalten ist in Tabelle 3.1 anhand sequentieller Zugriffe auf eine theoretische 32 Byte lange Konfigurationsdatei dargestellt. Die letzten vier Anforderungen sind überflüssig, da sich die angefragten Adressen jeweils bereits im Puffer befinden.

Dieses Verhalten kann durch Entfernung einer Bedingung in der für die Anforderung verantwortlichen if-Anweisung verhindert werden. Im in dieser Arbeit verwendeten Teil von DirectC besteht trotzdem keine Gefahr eines Zugriffs auf Daten außerhalb des Puffers, da jeder Aufruf der Funktion, die die Speicheradresse zu eine Dateiadresse berechnet, zusätzlich die Anzahl der im Puffer ab dieser Adresse verfügbaren Bytes liefert.

| angefragte Adresse | Pufferinhalt vor Anfrage | Anforderung |
|---|---|---|
| 0 | ? | 0...31 |
| 4 | 0...31 | |
| 8 | 0...31 | |
| 12 | 0...31 | |
| 16 | 0...31 | 16...31 |
| 20 | 16...31 | 20...31 |
| 24 | 20...31 | 24...31 |
| 28 | 24...31 | 28...31 |

**Tabelle 3.1:** Unnötige Anforderungen am Dateiende am Beispiel einer theoretischen 32 Byte lange Konfigurationsdatei.

**Zwischenspeicherung der Kopfdaten:**

Die von DirectC zur Konfiguration des FPGAs verwendete DAT-Datei ist in Blöcke unterteilt [24]. Die Startadresse der Blöcke ist in einer Lookup-Tabelle abgelegt, die ein Teil der Kopfdaten der DAT-Datei ist. Sobald DirectC auf einen anderen Block als zuvor zugreift, wird zunächst die Adresse der Lookup-Tabelle abgefragt, die an einer festen Adresse der Datei abgespeichert ist. Danach wird über diese Lookup-Tabelle die gesuchte Startadresse des Blockes bestimmt. Da alle Zugriffe auf die Kopfdaten

über eine einzige DirectC-Funktion (`dp_get_header_bytes`) stattfinden und diese nach Entfernung der Prüfsummenberechnung nicht mehr für den Zugriff auf die gesamte DAT-Datei verwendet wird, können die Kopfdaten mit geringem Aufwand auf dem Mikrocontroller während des ganzen Konfigurationsvorgangs zwischengespeichert werden. Dazu werden diese beim ersten Zugriff auf die Kopfdaten aus dem normalen Puffer in den Kopfdatenpuffer kopiert. Bei späteren Zugriffen kann stattdessen der Inhalt des Kopfdatenpuffers benutzt werden. Die Verwendung des Kopfdatenpuffers ermöglicht es, das erste Segment der Konfigurationsdaten schon vor der eigentlichen Initialisierung von DirectC anzufordern.

Für die Zwischenspeicherung wird angenommen, dass alle für das verwendete FPGA generierten DAT-Dateien 6 Einträge in der Lookup-Tabelle haben. Dies trifft für alle untersuchten Dateien zu. Weder die Microsemi bereit gestellte Dokumentation noch gezielte Anfragen konnten klären, ob für das verwendete AGL1000 FPGA prinzipiell Tabellen mit einer anderen Zahl an Einträgen erzeugt werden können. Für den Fall, dass diese Annahme unzutreffend sein sollte, wurde in der Gegenstelle eine Überprüfung der Anzahl an Lookup-Tabellen-Einträgen eingebaut. Sobald die Anzahl abweicht, wird eine Fehlermeldung ausgegeben und der Konfigurationsvorgang nicht gestartet.

Da die Länge der relevanten Kopfdaten mit 99 Byte kleiner als die maximal unkomprimiert in einem Funkpaket übertragbare Segmentlänge ist, wird die Übertragung vereinfacht als ein einzelnes Segment unabhängig von der eingestellten Segmentlänge vorgenommen. In Anbetracht der großen Zahl an Segmentanfragen (für eine Segmentlänge von 111 Byte erhält man ca. $2 \cdot 936524\,\text{Byte}/111\,\text{Byte} \approx 16875$ Anfragen), die während eines Konfigurationsvorgangs stattfinden, ist diese Zwischenspeicherung eine geringfügige Optimierung.

### 3.8.4 Ablauf des Konfigurationsvorgangs

In Abbildung 3.5 ist der Ablauf des Konfigurationsvorgangs nach den in den Abschnitten 3.8.1 bis 3.8.3 durchgeführten Optimierungen dargestellt. Zuerst überprüft der PC die Prüfsumme der Konfigurationsdatei. Falls diese Prüfung erfolgreich ist, sendet er den Startbefehl an den Mikrocontroller. Vor Durchführung seiner Initialisierung fragt dieser das erste Segment mit bekannter Adresse der Konfigurationsdaten an. Nach erhaltener Anfrage bereitet der PC das Segment vor, indem er es aus den Konfigurationsdaten ausschneidet und (falls gewünscht) mit dem für dieses Segment effektivsten der aktivierten Kompressionsalgorithmen komprimiert. Währenddessen führt DirectC auf dem Mikrocontroller (MCU) diverse Initialisierungen durch. Das Segment wird wie in Abschnitt 3.5 beschrieben kodiert und zum Mikrocontroller gesendet. Dieser dekomprimiert die Daten falls erforderlich und schreibt sie in den von DirectC erwarteten Puffer. Mit den erhaltenen Daten beginnt DirectC den Konfigurationsvorgang.

Sobald Daten benötigt werden, die nicht im Puffer abgelegt sind, wird ein weiteres Segment der Konfigurationsdaten angefordert. DirectC kann nicht weiterarbeiten, bis diese im Puffer abgelegt sind. Während der Übertragung der Anfrage an den PC, der Vorbereitung der Daten durch den PC und der Übertragung des angefragten Segments an den Mikrocontrollers wartet dieser auf den Empfang der Daten. Wie zuvor werden die Daten dekomprimiert und DirectC setzt die Arbeit fort. Dieser Vorgang wiederholt sich, bis alle Schritte des Konfigurationsvorgangs abgeschlossen sind. Zuletzt sendet der Mikrocontroller eine Nachricht mit dem ein Byte langen Rückgabewert von DirectC an den PC. Das von den DirectC-Entwicklern vorgegebene Wörterbuch mit den Bedeutungen der Meldungen ist in der Gegenstelle implementiert. Der Rückgabewert 0 steht, analog zu POSIX-kompatiblen Systemen, für den Erfolgsfall, also „kein Fehler aufgetreten“.

### 3.8.5 Segmentvorhersage

Eine anderer Ansatz, um die Laufzeit des Konfigurationsvorgangs zu verkürzen, ist die Vermeidung von Phasen, in denen der Mikrocontroller nicht arbeiten kann. Während DirectC auf die Übertragung von Konfigurationsdaten wartet, kommt es zu solchen Phasen. Eine Möglichkeit, diese Phasen zu verkürzen

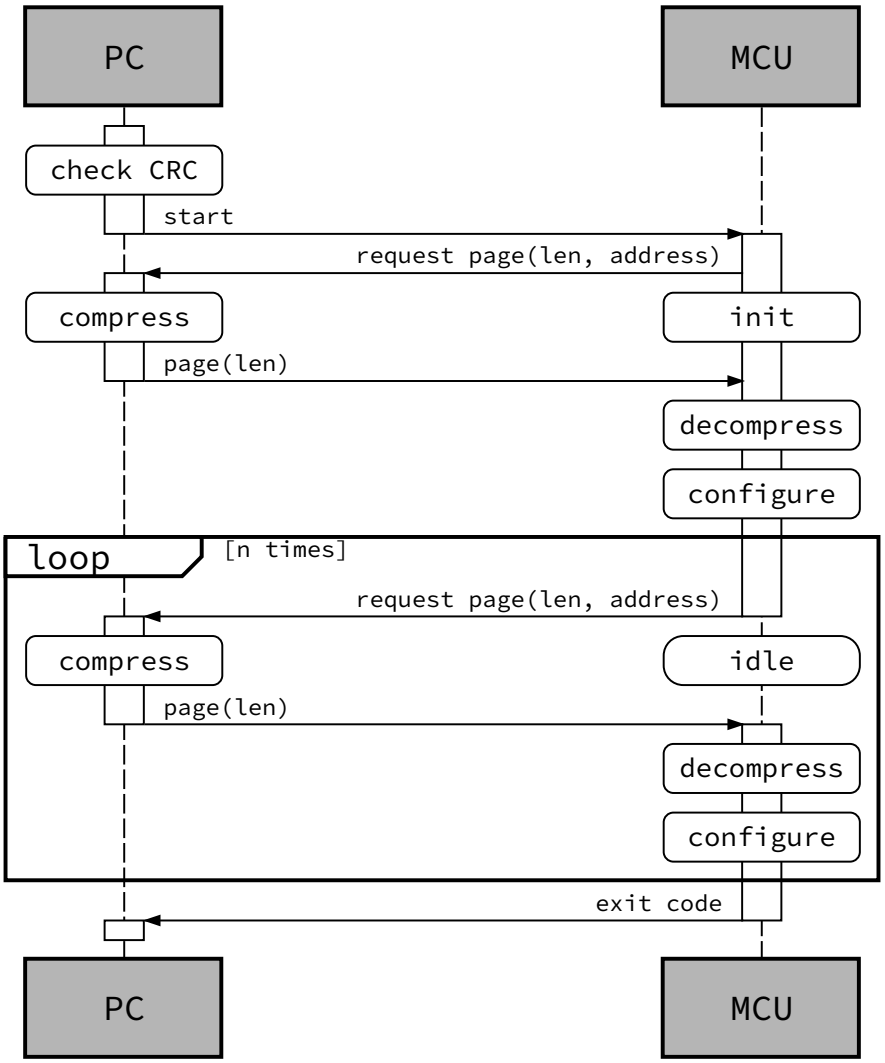


**Abbildung 3.5:** Ablauf des Konfigurationsvorgangs.

oder sogar ganz zu vermeiden, besteht darin, die Übertragung durch Verwendung des DMA-Controllers des Mikrocontrollers nebenbei zu erledigen. Um diesen Umbau durchzuführen, muss herausgefunden werden, welche Daten DirectC in Zukunft benötigen wird. Bei einer längeren Untersuchung des Quelltextes hat sich herausgestellt, dass ein Umbau von DirectC durch die verwendete Struktur (u. a. Arbeit auf Basis globaler Variablen) sehr aufwändig wäre. Glücklicherweise zeigt sich, dass bei fester Segmentlänge und anderen Parametern die Zugriffe auf die Konfigurationsdaten durch DirectC gleich sind und damit unabhängig von der verwendeten Konfigurationsdatei sind. Die auftretenden Zugriffsmuster (Vorhersagedaten) können von der Gegenstelle auf dem PC aufgezeichnet werden. Beim nächsten Konfigurationsvorgang können mit diesen Daten die jeweils nächsten benötigten Segmente vorhergesagt werden. Solange die verwendeten Optionen nicht geändert werden, können die Vorhersagedaten ohne Probleme wiederverwendet werden.

Die Abhängigkeit der Vorhersagedaten von der verwendeten Segmentlänge führt zu einem Problem in Kombination mit der im zweiten Teil von Abschnitt 3.8.2 beschriebenen Kompression mit variabler Rohdatenmenge (d. h. Segmentlänge). Da die Segmentlänge für jedes Segment aus der Wahl des Kompressionsverfahrens folgt, sind die Vorhersagedaten in diesem Modus nicht unabhängig von der verwendeten DAT-Datei. Da die Wahl des Kompressionsverfahren deterministisch ist, ist es zumindest möglich für jede DAT-Datei einen Satz an Vorhersagedaten zu generieren. Dies ist in der Praxis ein großer Nachteil,

da die Vorhersage erst bei einem zweiten Konfigurationsvorgang mit den gleichen Konfigurationsdaten verwendet werden kann.

Die durch diese Optimierung verursachten Änderungen im Ablauf des Konfigurationsvorgangs werden im folgenden Abschnitt erläutert.

### 3.8.6 Ablauf des Konfigurationsvorgangs mit Vorhersage

In Abbildung 3.6 ist der Ablauf des Konfigurationsvorgangs mit Segmentvorhersage in idealisierter Form dargestellt. Im Gegensatz zum Konfigurationsvorgang ohne Vorhersage werden jetzt Segmentanfragen ohne Adresse verwendet, dafür wird die Adresse des vorhergesagten Segments bei der Antwort mitgesendet, damit überprüft werden kann, ob das richtige Segment vorhergesagt wurde. Der Hauptunterschied ist, dass bei dieser Variante die Anfrage für ein neues Segment direkt nach dem Dekomprimieren des vorherigen Segments übertragen wird. Im günstigsten Fall bedeutet dies, dass DirectC nicht auf das Eintreffen von neuen Daten warten muss und der Mikrocontroller ohne Unterbrechung arbeiten kann. Da auf Seiten des Mikrocontrollers bei der Anforderung weiterer vorhergesagter Segmente nicht bekannt ist, ob das letzte erforderliche Segment bereits erhalten wurde, wird immer eine weiteres Segment angefordert. Dies führt dazu, dass gegen Ende des Konfigurationsvorgangs eine unnötige Anfrage abgeschickt wird. Die Gegenstelle antwortet auf diese laut den Vorhersagedaten unnötige Anfrage mit einer Dummy-Antwort der Adresse 0 und der minimal möglichen Länge 1.

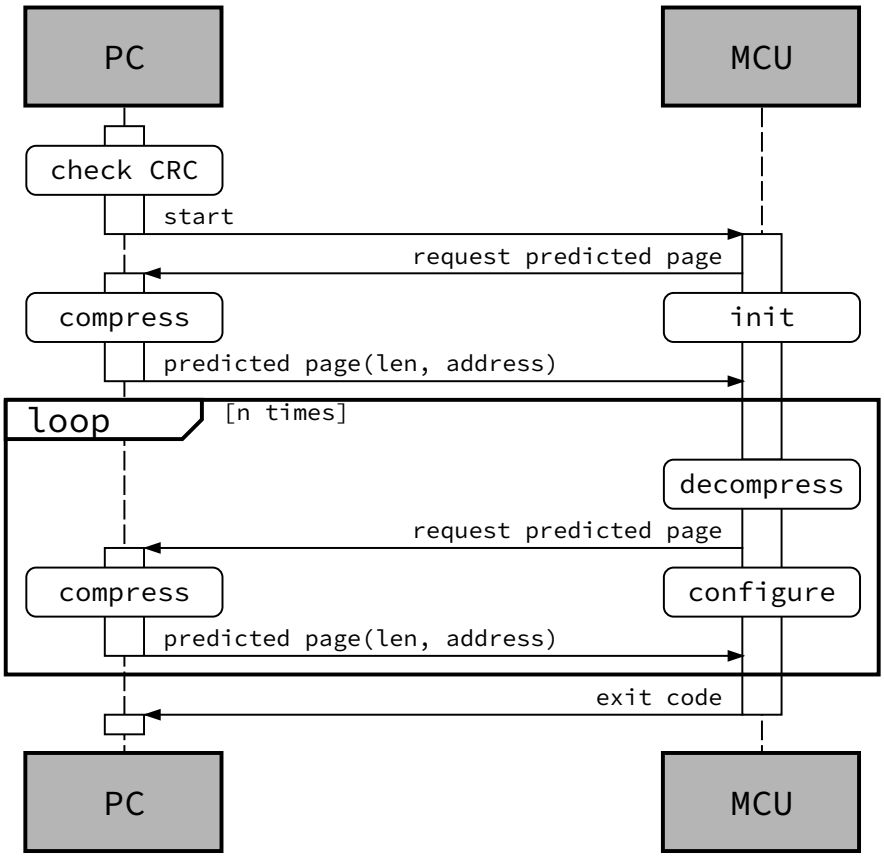


**Abbildung 3.6:** Ablauf des Konfigurationsvorgangs mit Vorhersage.

Um sicherzustellen, dass der Puffer mit den von DirectC erwarteten Daten gefüllt ist, wird vor dem (eventuellen) Entpacken überprüft, ob die mit den empfangenen Daten übertragene Adresse mit der benötigten Adresse übereinstimmt. Ist dies nicht der Fall, wird von einem Fehler der Vorhersagedaten ausgegangen. Nach Detektion eines solchen Vorhersagefehlers wird eine Anfrage (ohne Vorhersage) an

den PC geschickt und der Konfigurationsvorgang wird ohne Vorhersage fortgesetzt. Dieser Ablauf ist in Abbildung 3.7 dargestellt. Prinzipiell sind verschiedene Strategien zur Wiederaufnahme der Vorhersage denkbar. Beispielsweise könnte nach einer festgelegten Anzahl von Anfragen nicht vorhergesagter Segmente die Gegenstelle auf dem PC versuchen, die korrekte Stelle innerhalb der Vorhersagedaten zu finden. Jede dieser Wiederaufnahmestrategien hat den Nachteil, dass sie, falls die Vorhersagedaten tatsächlich falsch sind, zu Verzögerungen führt. Die verwendete Strategie zum Abfangen von Vorhersagefehlern versucht keine Wiederaufnahme der Vorhersage, sondern korrigiert die Vorhersagedaten automatisch, sodass sie beim nächsten Konfigurationsvorgang verwendet werden können.

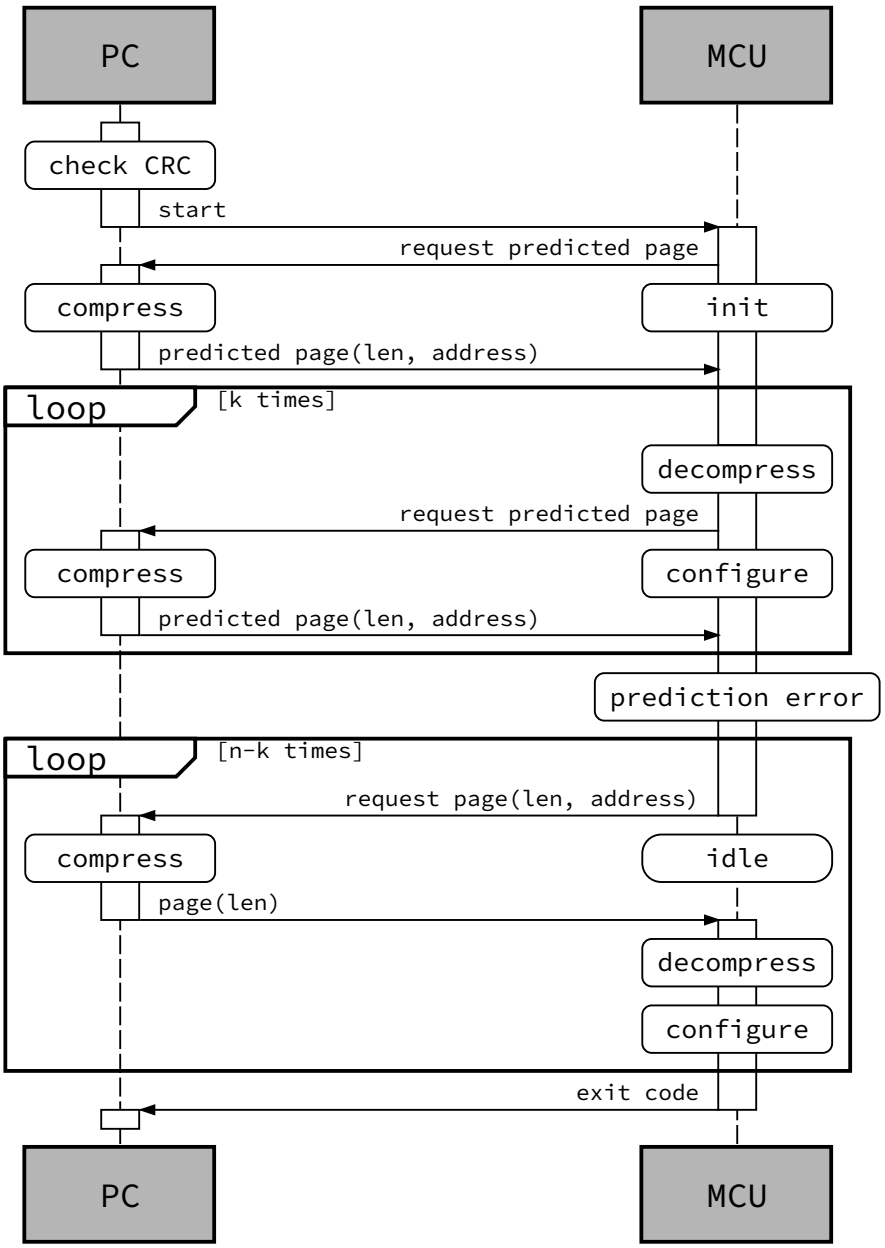


**Abbildung 3.7:** Ablauf des Konfigurationsvorgangs mit Vorhersage bei Detektion eines Vorhersagefehlers.

In der Realität kann es passieren, dass die Komprimierung bzw. Vorbereitung der Daten und die Übertragung der nächsten Segmente länger als die Arbeiten von DirectC mit dem aktuellen Pufferinhalt dauert. In diesem Fall kommt es trotz Vorhersage zu Wartephasen des Mikrocontrollers. Da DirectC die Daten angefordert hat, werden sie zumindest in irgendeiner Form verarbeitet, deshalb ist die War-

tezeit des Mikrocontrollers geringer als im Fall ohne Vorhersage. Dieser nicht idealisierte Ablauf des Konfigurationsvorgangs ist in Abbildung 3.8 dargestellt.

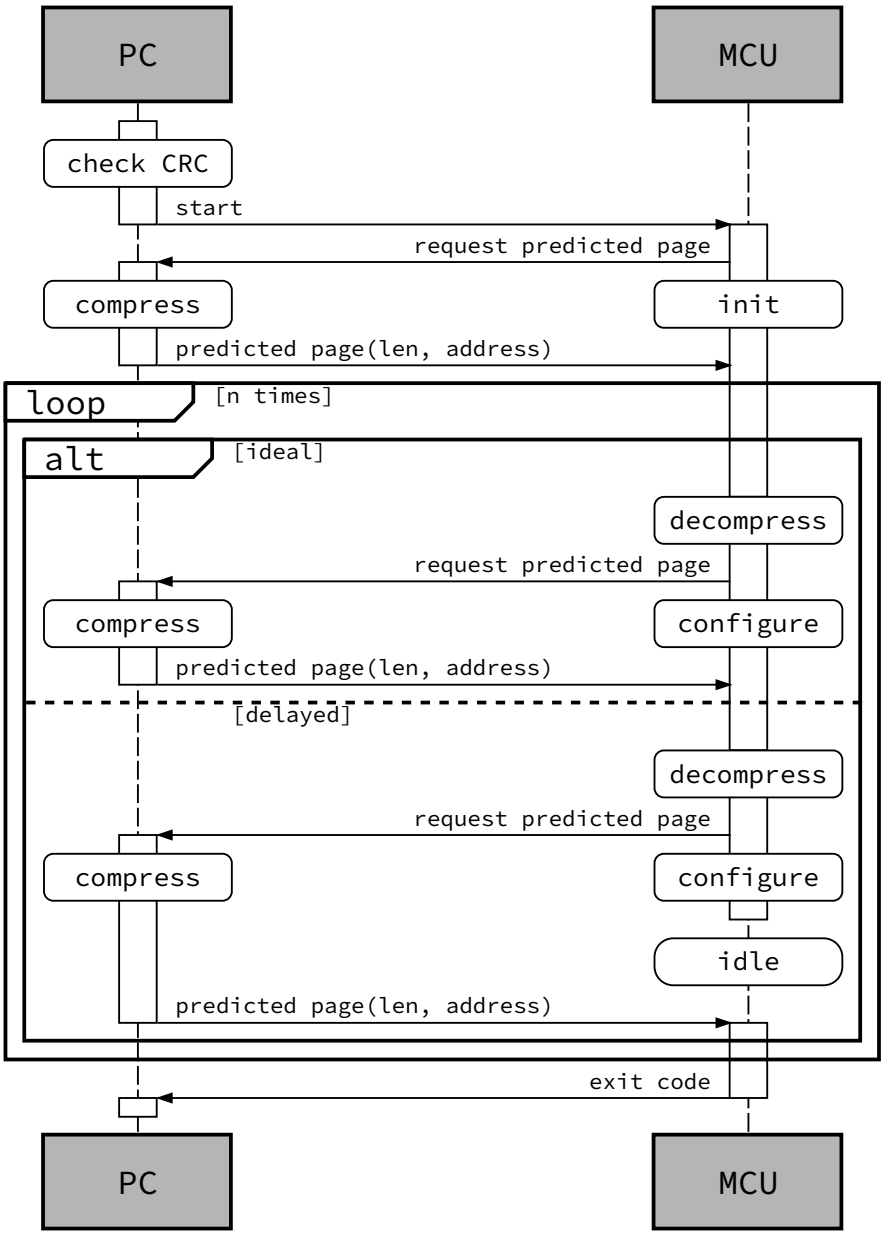


**Abbildung 3.8:** Ablauf des Konfigurationsvorgangs mit Vorhersage mit möglichen Verzögerungen.

### 3.8.7 Kommunikation vom Mikrocontroller zum PC

Die kabellose IEEE 802.15.4 konforme Übertragung ist im Gegensatz zur kabelgebundenen Übertragung nicht duplexfähig. Jeder Knoten kann zu jedem Zeitpunkt entweder nur senden oder empfangen, die Kommunikation findet im Wechselbetrieb (Halbduplex) statt. Befinden sich zwei Knoten gleichzeitig im Sendemodus, kann keiner der beiden die gesendeten Daten empfangen und somit auch keine Empfangsbestätigung senden. Empfängt ein Knoten in einem festgelegten Zeitintervall keine Empfangsbestätigung, sendet er das Paket erneut. Dies kann zu erheblichen Verzögerungen führen. Bei ungünstig gesetzten Parametern kann es passieren, dass beide Knoten nach mehreren Wiederholungen aufgeben und die Kommunikation abbricht.

Die vom Mikrocontroller zum PC gesendeten Nachrichten lassen sich in zwei Kategorien einteilen: Die Anforderungen von Segmenten der FPGA-Konfigurationsdaten ist notwendig und zudem zeitkritisch, während die Statusmeldungen lediglich nützlich und nicht zeitkritisch sind. Wenn man sie nutzen will, müssen sie natürlich auch per Funk übertragen werden. Für die korrekte Durchführung der Messung sind jedoch die Statusmeldungen, die den Beginn und das Ende des Konfigurationsvorgangs auf dem FPGA anzeigen, zeitkritisch. Prinzipiell könnten die Statusmeldungen bei fehlerfreiem Ablauf des Konfigurationsvorgangs mittels vorhandener Präprozessordirektiven vollständig eingespart werden. Da der (kabellose) Konfigurationsvorgang mehrere Minuten (siehe Abschnitt 5.3) dauert und die Statusmeldungen eine Doppelfunktion als Fortschrittsanzeige und Kontrollmechanismus haben, ist es von Vorteil, sie zu verwenden.

In DirectC ist vorgesehen, dass alle Meldungen (auch Zahlen) als ASCII-Zeichenkette auf einer lokal angeschlossenen Anzeige dargestellt werden. Die Anzahl der möglichen Texte ist gering, da der in der Arbeit verwendete Teil von DirectC 85 verschiedene Meldungen enthält. Diese Meldungen sind statisch, da alle variablen Anteile als separate Zahlenmeldungen vorgesehen sind. Deshalb bietet sich die Verwendung einer Wörterbuchkompression an, bei der ein Byte zur Spezifizierung des Textes ausreicht. Analog dazu können die Zahlen als 4 Byte Werte statt als Folge von ASCII-codierten Ziffern übertragen werden. Durch diese Maßnahmen wird die Länge der gesendeten Pakete stark reduziert.

Um die Anzahl der gesendeten Pakete zu verringern, können die zeitunkritischen Statusmeldungen im Sendepuffer zwischengespeichert werden. Der gesamte Inhalt des Sendepuffers wird nur gesendet, sobald eine zeitkritische Statusmeldung oder Segmentanforderung übertragen werden soll oder falls nach der Pufferung eine kritische Füllmenge überschritten wird, weil dann nicht mehr sichergestellt ist, dass die längstmögliche kodierte Nachricht an den Pufferinhalt angehängt werden kann.

Untersuchungen der Funkübertragung mit einem Paket-Sniffer [31] haben gezeigt, dass nur eine einstellige Anzahl von zusätzlichen Paketen durch Statusmeldungen erzeugt wird. Anhand des vom PC aufgezeichneten Protokolls eines typischen erfolgreichen Konfigurationsvorgangs kann berechnet werden, dass durch die Wörterbuchkompression 683 Byte und durch die geänderte und kompaktere Übertragung der Zahlen 191 Byte eingespart werden können. Durch die Zwischenspeicherung können ca. 130 Funkpakete eingespart werden. Eine exakte Angabe der Anzahl der Funkpakete ist nicht möglich, da die Funkpakete nicht beim PC ankommen, sondern beim Gateway (siehe Abschnitt 3.1), der die empfangenen Daten per UART weiterleitet.

### 3.8.8 Kleinere Programmoptimierungen

Zur Kompilierung des Projektes wird der frei verfügbare SDCC (Small Device C Compiler) [32] in der Version 3.5.0 verwendet. Dies hat gegenüber kommerziellen Compilern unter anderem den Vorteil, dass die Ergebnisse der Arbeit ohne Einschränkungen durch benötigte Lizenzen überprüft werden können. Ein bekannter Nachteil von SDCC ist, dass der erzeugte Code in vielen Fällen, vor allem im Vergleich zum IAR Embedded Workbench von IAR Systems, groß ist. SDCC kann unbenutzte statische Funktionen überhaupt nicht entfernen. Eine Entfernung von nicht-statischen Funktionen durch den Linker wird nur unterstützt, wenn jede Funktion in einem separaten Objekt bzw. in einer eigenen Quelldatei angelegt ist. Beim Versuch, den Speicherbedarf des Projektes in verschiedenen Konfigurationen zu vergleichen, ist eine weitere Eigenart von SDCC zu beobachten: Die Größe der verschiedenen Segmente (`CODE`, `DATA`, `XDATA`) ändert sich bei erneutem Kompilieren des ungeänderten Projektes.

Bei den Untersuchungen zur Implementierung einer Segmentvorhersage in DirectC wurde ungenutzter Code der JTAG-Funktionalität gefunden und mittels Präprozessordirektiven entfernt.

### 3.8.9 „Optimale“ Strategie

Wie in Abschnitt 3.8.6 beschrieben, findet das Entpacken vorhergesagter Segmente zu dem Zeitpunkt statt, an dem das neue Segment benötigt wird, d. h. das Entpacken verursacht eine Verzögerung. Unter

der Annahme, dass ein neues unkomprimiertes Segment vollständig übertragen werden kann, während DirectC auf den Daten des aktuellen Segments arbeitet, also keine Wartezeit auftritt, kann durch eine unkomprimierte Übertragung Zeit gespart werden. Nimmt man zudem an, dass die Übertragung und das Entpacken eines komprimierten Segments schneller ist als die Übertragung eines unkomprimierten Segments, bietet sich die folgende Strategie an: Alle vorhergesagten Segmente werden unkomprimiert übertragen. Sobald die Übertragung ohne Vorhersage (z. B. durch einen Vorhersagefehler) stattfindet, werden die Segmente komprimiert übertragen.

Diese Strategie kann komplett von der Gegenstelle auf dem PC gesteuert werden und erfordert keine Änderungen am Programm des Mikrocontrollers, solange alle benötigten Funktionalitäten aktiviert sind.

# 4 Durchführung der Messungen

In diesem Kapitel wird die Durchführung der Messungen dargestellt. Zunächst werden die drei für die Messungen verwendeten Konfigurationsdateien vorgestellt. Dann wird der Ablauf der Messungen unter Verwendung von DirectC und des von Microsemi vertriebenen Programmiergeräts erläutert. Zuletzt werden die für die Messung der Konfigurationszeit und des Energiebedarfs verwendete Hardware vorgestellt.

## 4.1 Verwendete DAT-Dateien

Für die Durchführung der Messungen wurden drei verschiedene FPGA-Konfigurationsdateien D1, D2 und D3 verwendet. D2 nutzt ungefähr die achtfache Anzahl an FPGA-Zellen wie D1, während D3 ungefähr die vierfache Anzahl an FPGA-Zellen verwendet wie D2. D1 und D2 realisieren synthetische Testanwendungen verschiedener Größe, die aus zwei parallel ausgeführten Teilen bestehen: Eine LED blinkt mit einer parametrisierten Rate (`RATE`). Gleichzeitig wird $Y = X^{\texttt{SIZE}}$ mit dem Parameter `SIZE` über eine Verkettung von 8-Bit-Multiplikatoren berechnet. Der zur Erzeugung der Dateien verwendete Verilog-Code ist in Algorithmus 4.1 aufgeführt. D3 ist für den tatsächlichen Einsatz der HaLOEWEn auf dem in Abschnitt 2.2 erwähnten Gebiet der Strukturüberwachung erstellt worden. Eine Übersicht inklusive genauer Zahlenwerte zur Anzahl der verwendeten Zellen und der gesetzten Parameter bei den künstlich erzeugten Testdateien ist in Tabelle 4.1 aufgeführt.

| Datei | Verwendete Zellen (*Core Cells*) | Anwendung | Parameter | |
|---|---|---|---|---|
| | | | `RATE` | `SIZE` |
| D1 | 716 (3 %) | künstlich | 1 | 5 |
| D2 | 5677 (23 %) | künstlich | 10 | 50 |
| D3 | 20714 (84 %) | SHM | N/A | N/A |

**Tabelle 4.1:** Ressourcenauslastung der drei DAT-Dateien.

## 4.2 Ablauf der Messungen

Da, wie in Abschnitt 2.2.1 erwähnt, das zur Programmierung des CC2530 benötigte Programm nur für Windows vorliegt, sind die Messungen während der Programmierung des Mikrocontrollers (d. h. der Arbeiten an DirectC, der Funkanbindung, der Dekompression, etc.) größtenteils unter Windows 8.1 durchgeführt worden. Dabei wurde jede Einzelmessung per Hand gestartet.

Die im folgenden Kapitel 5 vorgestellten Messergebnisse stammen, außer wenn gegenteilig vermerkt, von halb-automatischen Messungen, die als Messreihen unter Arch Linux durchgeführt wurden. Für feste Parameter erfolgten jeweils automatisch 10 Messungen. Einige Parameter können ohne Änderungen der Programmierung des Mikrocontrollers geändert werden. Somit können die Auswirkungen der Änderungen automatisch getestet werden. Dies beinhaltet die Deaktivierung von RLE/RLEM-Kompression und die Deaktivierung der Vorhersage. Für die Änderung einiger anderer Parameter, die eine große Auswirkung auf die Programmgröße und -komplexität haben, wie die zusätzliche Aktivierung der Huffman-Kompression oder die Änderung der Segmentgröße, ist eine manuelle Übertragung der Programmierung auf den Mikrocontroller unter Windows erforderlich. Aus diesem Grund wird der Begriff halb-automatisch verwendet.

Das zur Messung des Energiebedarfs verwendete Programm PowerScale liegt ebenfalls nur in einer Version für Windows vor. Zudem lässt es sich nicht automatisieren, sodass jede Messung per Hand gestartet

**Algorithmus 4.1** Verilog-Code zur Erzeugung von D1 und D2.

```
1  `default_nettype none
2  module top
3   (input  wire OSC_CLK,      // external oscillator
4    input  wire PO_RSTN,      // asynchronous power on reset (active low)
5    input  wire PB_RSTN,      // asynch. push button reset (active low)
6    output wire LED,          // indicator for SPEED-configuration
7    input  wire [7:0] X,      // input for exponentiation
8    output wire [7:0] Y);     // result of exponentiation (X^SIZE),
9                              // overflows are ignored
10
11 // global settings (shared with testbench)
12 `include "settings.v"
13
14 // combine reset sources
15 wire rstN = PO_RSTN && PB_RSTN;
16
17 // generate LED toggle rate by appropriate counter
18 localparam PERIOD = (OSC_RATE * 1000000) / LED_RATE;
19 reg [$clog2(PERIOD)-1:0] counter;
20 always @(posedge OSC_CLK) begin
21   if (!rstN || counter == PERIOD-1) counter <= 0;
22   else                              counter <= counter + 1;
23 end
24 assign LED = (counter >= PERIOD/2) ? 1 : 0;
25
26 // pipelined exponentiation (Y = X^SIZE)
27 reg  [ 7:0] x_pipe [0:SIZE-1];
28 reg  [15:0] y_pipe [0:SIZE-1];
29 genvar k;
30 generate
31   for (k=0; k<SIZE; k=k+1) begin
32     always @(posedge LED or negedge rstN) begin
33       if (!rstN) begin
34         x_pipe[k] <= 0;
35         y_pipe[k] <= 0;
36       end else if (k == 0) begin
37         x_pipe[k] <= X;
38         y_pipe[k] <= X;
39       end else if (k > 0) begin
40         x_pipe[k] <= x_pipe[k-1];
41         y_pipe[k] <= y_pipe[k-1][7:0] * x_pipe[k-1];
42       end
43     end
44   end
45 endgenerate
46 assign Y = y_pipe[SIZE-1][7:0];
47
48 endmodule
```

und auch gestoppt werden muss. Aus diesem Grund und dem großen Platzbedarf der aufgezeichneten Messdaten wurden für jede untersuchte Parameterkombination zwei Messungen durchgeführt.

## 4.3 Vergleichsgrundlage für die Konfigurationszeit

Zur Konfiguration des FPGAs vertreibt der Hersteller ein mittels USB an den PC anschließbares Programmiergerät namens FlashPro 4. Das dazugehörige Programm, das ebenfalls FlashPro heißt, kann über diesen Adapter das FPGA über seine JTAG-Schnittstelle konfigurieren. FlashPro verwendet ein ein drittes, anderes Dateiformat als DirectC und STAPL Player. Über den internen Aufbau des Programmiergeräts werden vom Hersteller keine Angaben gemacht, aber das abweichende Dateiformat lässt darauf schließen, dass er weder auf DirectC noch auf STAPL Player basiert. Um eine Grundlage für die Dauer des Konfigurationsvorgangs zu erhalten, wurden unter Windows mit FlashPro 11.7.0.119 mittels Python jeweils zehn Konfigurationsvorgänge mit jeder der drei den DAT-Dateien entsprechenden Konfigurationsdateien automatisiert durchgeführt.

## 4.4 Verwendete Hardware

Wie bereits in Abschnitt 2.2 erwähnt, wurden für die Messungen zwei unterschiedliche Aufbauten verwendet.

### 4.4.1 Messung der Konfigurationszeit

Der größte Teil der Messungen, wie zum Beispiel die Messung der Konfigurationszeit, werden mit der SmartRF05EB-Evaluationsplatine und dem auf dem HaLOEWEn verbauten FPGA durchgeführt. Die Verbindungen zum JTAG-Sockel des FPGAs auf der HaLOEWEn-Platine sind in Tabelle 4.2 angegeben. `VPUMP` und `GND` werden von Pins des `USB Debug`-Sockels des SmartRF05EB bezogen. Die HaLOEWEn-Platine wird über ihren `VBT`-Sockel mittels eines generischen Steckernetzteils versorgt. Die Spannung des Steckernetzteils schwankt bei der Einstellung für die gewünschten 3,0 V zwischen 3,9 V und 4,6 V. Dies hat keine negativen Auswirkungen, da die an `VBT` anliegende Spannung nicht direkt verwendet wird, sondern alle Bauteile über vorgeschaltete Spannungsregler versorgt werden. Die USB-Schnittstelle des SmartRF05EB kann zur Programmierung des CC2530 verwendet werden und dient gleichzeitig als seine Spannungsquelle.

| JTAG-Verbindung | SmartRF05EB CC2530 | SmartRF05EB |
|---|---|---|
| TCK | P0_0 | P20.12 |
| TDI | P0_6 | P20.17 |
| TDO | P1_3 | P18.4 |
| TMS | P0_7 | P18.17 |
| TRST | P1_2 | P20.18 |
| VPUMP | | P2.2 (USB Debug 3.3 V) |
| GND | | P2.1 (USB Debug GND) |

**Tabelle 4.2:** Pinbelegung für allgemeine Messungen.

### 4.4.2 Messung der benötigten Energie

Für die Konfigurationsvorgänge zur Messung der benötigten Energie werden das FPGA und der CC2531 der HaLOEWEn-Platine verwendet. Das SmartRF05EB wird lediglich als Quelle für `VPUMP` (und somit

| JTAG-Verbindung | HaLOEWEn CC2531 | HaLOEWEn | SmartRF05EB |
|---|---|---|---|
| TCK | P1_0 | J7.2 | |
| TDI | P0_7 | J6.2 | |
| TDO | P0_6 | JS3.1 | |
| TMS | P0_0 | JS1.1 | |
| TRST | P0_1 | JS2.1 | |
| VPUMP | | | P2.2 (USB Debug 3.3 V) |
| GND | | | P2.1 (USB Debug GND) |

**Tabelle 4.3:** Pinbelegung für Energiemessungen.

auch GND) verwendet. Um eine mögliche Störung (u. a. durch Funksignale auf dem gleichen Kanal) zu vermeiden, wird die Platine, auf der der CC2530 verbaut ist, vom SmartRF05EB abgezogen. Die in diesem Aufbau verwendeten Verbindungen sind in Tabelle 4.3 angegeben. Während Voruntersuchungen mit einem Rigol DP832A als Spannungsquelle durchgeführt wurden, ist für die ausführlichen Messungen der Spannungsausgangsmodus eines Nosram Stealth Evolution Batterieladegerätes aus logistischen Gründen als Quelle der 3,0 V für VBT verwendet worden.

Die Spannungs- und Strommessungen werden mit einem vier-kanaligen Messgerät namens PowerScale der Firma hitex durchgeführt. Mit ihm ist es möglich, über USB Spannungs- und Strommesswerte mit (je nach Anzahl der verwendeten Kanäle) bis zu 100 kHz am Computer aufzuzeichnen. Die Messdaten werden von der mitgelieferten Software in einem proprietären Format gespeichert. Nach freundlicher Anfrage war es möglich, eine Spezifikation zu erhalten und damit einen Interpreter in Python zu implementieren. Des Weiteren erlaubt das Messgerät die Verwendung eines globalen und pro Kanal eines zusätzlichen lokalen Triggereingangs. Um akkurate Zeitinformationen über den Beginn und das Ende des Konfigurationsvorgangs zu erhalten, wurde ein Kabel an ein Bein des auf dem HaLOEWEn verbauten Tasters (fest mit Pin P2_0 des CC2531 verbunden) gelötet, da alle anderen freien Pins für die JTAG-Schnittstelle benötigt wurden. Damit ist es möglich, vom Mikrocontroller beim Start des Konfigurationsvorgangs eine positive Kante und beim Ende des Konfigurationsvorgangs eine negative Kante auf den Triggereingang des PowerScale zu erzeugen. In Abbildung 4.1 ist der Anschluss des Messgerätes abgebildet. An Kanal 1 werden Spannung und Strom an VBT und an Kanal 2 Spannung und Strom an VPUMP gemessen. Positiv zu vermerken ist, dass die Spannung auf Seiten des Verbrauchers gemessen wird und dadurch der Spannungsabfall durch die Strommessung nicht vernachlässigt wird. Laut Anleitung des Messgerätes beträgt dieser Spannungsabfall ca. 100 mV [33]. Die Messungen wurden mit einer Samplerate von 75 kHz durchgeführt.

Die nach dem durch die Strommessung verursachten Spannungsabfall gemessene Spannung an VBT liegt zwischen 2.78 V und 3.17 V, während die Spannung an VPUMP zwischen 3.22 V und 3.45 V liegt.

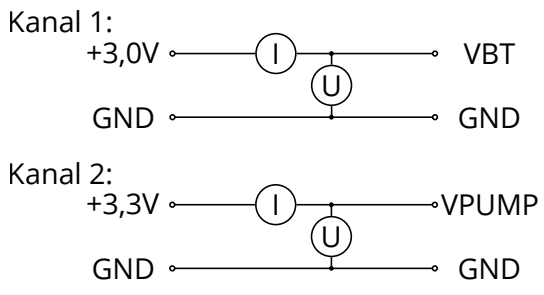


**Abbildung 4.1:** Anschluss des hitex PowerScale.

# 5 Auswertung

In diesem Kapitel werden die Messergebnisse vorgestellt und interpretiert. Zunächst wird in einer Simulation die Komprimierbarkeit der Konfigurationsdateien durch verschiedene Kompressionsalgorithmen untersucht. Im Anschluss daran wird die Größe der Firmware mit und ohne Huffman-Kodierung verglichen und die Absolutgröße betrachtet.

Danach werden die Ergebnisse der Konfigurationszeitmessungen für verschiedene Parameter (wie Übertragungsmedium, Kompression, Segmentvorhersage und Segmentlänge) vorgestellt und die Auswirkungen der Änderung der Parameter auf die Konfigurationszeit untersucht. Im Anschluss wird eine beispielhafte Kurve des Stromflusses über die beiden Versorgungsleitungen `VBT` und `VPUMP` erläutert und der Energiebedarf eine Konfigurationsvorgangs für verschiedene Parameter vorgestellt und verglichen.

## 5.1 Voruntersuchung zur Kompression

Um die zu Beginn von Abschnitt 2.7 geäußerte Vermutung, dass die von DirectC verwendeten DAT-Dateien gut komprimierbar seien sollten, zu überprüfen, bieten sich mehrere simulierte Untersuchungen ohne Einbezug von Mikrocontroller und FPGA an. Zunächst kann man die vorgestellten Algorithmen komplett auf die zur Messung verwendeten DAT-Dateien (siehe Abschnitt 4.1) anwenden. Das Ergebnis ist in Abbildung 5.1 aufgetragen: Die Balkengruppen stehen hier für die verschiedenen Methoden. Die drei Balken innerhalb der Gruppen stehen für die verschiedenen DAT-Dateien; von links nach rechts D1, D2 und D3. Ihre Höhe ist proportional zur Größe in Kilobyte. Wie in Abschnitt 2.5 erwähnt, sind alle Dateien im unkomprimierten Zustand (linke Balkengruppe) exakt gleich groß. Bei allen Kompressionsmethoden lässt sich D1 besser komprimieren als D2 und D2 besser komprimieren als D3. Dies entspricht der Erwartung, da die Nummerierung der Dateien nach steigender Komplexität der modellierten Netze vorgenommen wurde. Betrachtet man die Größe der einzelnen Dateien mit den verschiedenen Methoden, fällt auf, dass sie von links nach rechts immer weiter abnimmt. Bis auf die hier sehr eindeutige

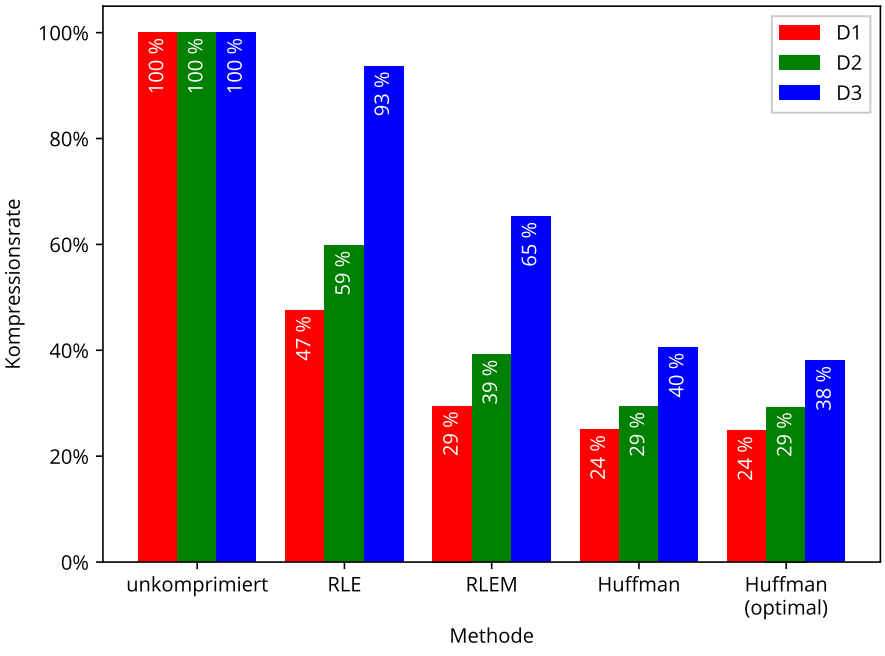


**Abbildung 5.1:** Theoretischer Vergleich der Kompressionsmethoden.

Verbesserung vom Wechsel von RLE auf RLEM, die darauf hinweist, dass die Lauflängenkodierung für manche Teile der DAT-Dateien ungeeignet ist, entspricht dies den Erwartungen, da die rechts angeordneten Huffman-Kodierungen komplexere Verfahren sind. Die zweite von rechts und mit „Huffman“ beschriftete Methode verwendet einen Huffman-Baum, der aus einer gemeinsamen Häufigkeitsverteilung von D1 und D2 berechnet wurde. Die ganz rechts aufgetragene und mit „Huffman (optimal)“ beschriftete Methode verwendet für jede DAT-Datei einen optimalen Huffman-Baum, der aus der Häufigkeitsverteilung der jeweiligen Datei erstellt wurde. Bei der Berechnung der Größe der Huffman-kodierten Daten ist der Speicherbedarf des Huffman-Baums nicht eingerechnet.

Die wichtigsten Erkenntnisse dieser groben Untersuchung sind, dass D1 und D2 durch RLEM ordentlich (ca. 40 %) komprimiert werden und dass die Huffman-Kodierung mit einem optimalen Huffman-Baum für die untersuchten Dateien keinen großen Vorteil erreicht gegenüber einem „gemittelten“, nicht optimalen Baum.

Unter Verwendung einer bei späteren Messungen erstellten Datei von tatsächlich auftretenden Zugriffsmustern auf Segmente der DAT-Datei kann man genauere Untersuchungen durchführen. DirectC greift mehrfach auf die Segmente zu, deshalb sind die simulierten übertragenen Datenmengen höher als die Größe der ursprünglichen Dateien. Bei der Simulation wurde der zusätzliche Overhead durch die Kopf- und Fußdaten der Funkpakete nicht berücksichtigt, deshalb hat die eingestellte Größe der übertragenen Segmente von 111 Byte keine Auswirkung auf das Ergebnis.

Bei der Untersuchung der Struktur der DAT-Dateien fällt auf, dass sich, je nach modellierter logischer Schaltung bzw. Anwendung, größere Bereich mit vielen Wiederholungen und Bereiche ohne Wiederholungen abwechseln. Da sowieso segmentweise auf die Datei zugegriffen werden muss und verschiedene Kompressionsmethoden zur Verfügung stehen, bietet sich die folgende Strategie an: Beim Komprimieren der Segmente wird für jedes Segment untersucht, welche Methode die größte Kompression ermöglicht. Bei gleicher Größe wird anhand einer Präferenzliste entschieden. Mit abnehmender Präferenz wird unkomprimiert, RLE, RLEM, Huffman-Kodierung und optimale Huffman-Kodierung verwendet. Hierbei wird sichergestellt, dass kein kodiertes Segment die Länge der unkodierten Form überschreitet.

In Abbildung 5.2 ist das Ergebnis analog zur vorherigen Darstellung aufgetragen. Die beiden rechten zusätzlichen Gruppen zeigen die Resultate für die im letzten Absatz beschriebene Strategie. Die beiden

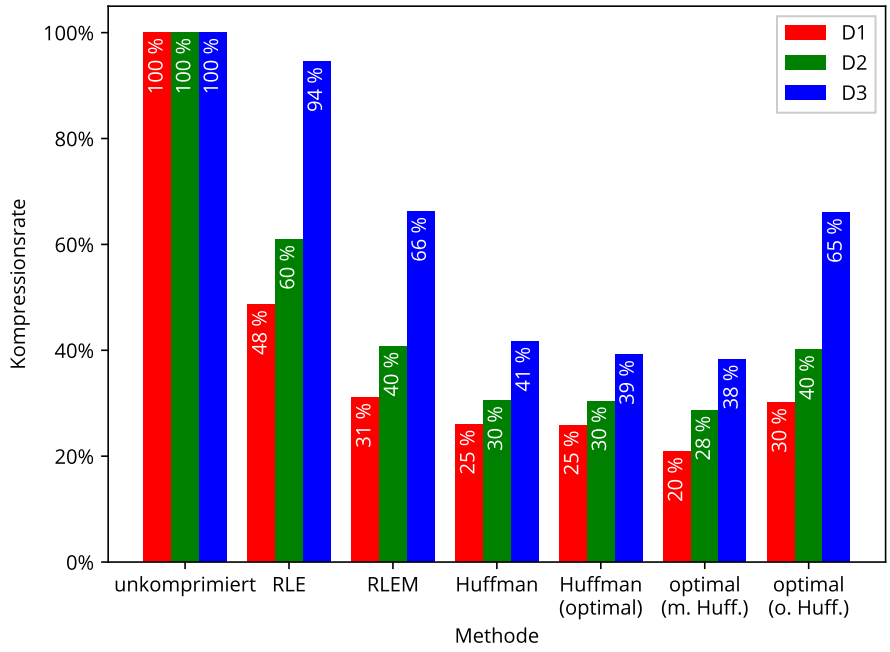


**Abbildung 5.2:** Vergleich der Kompressionsmethoden (simulierte Übertragung).

Gruppen unterscheiden sich nur dadurch, dass für die linke mit „optimal (m. Huff.)“ beschriftete die Verwendung der Huffman-Kodierung erlaubt ist im Gegensatz zur rechten, die mit „optimal (o. Huff.)“ beschriftet ist. Wie angenommen zeigt die „optimale“ Strategie eine Verbesserung zur jeweiligen vormals besten Methode. Die sechste Balkengruppe (optimale Kompressionmethode mit Huffman-Kodierung) ist niedriger als die fünfte (nur Huffman-Kodierung mit optimalem Baum) und die letzte Balkengruppe (optimale Kompressionsmethode ohne Huffman-Kodierung) ist niedriger als die dritte (nur RLEM). Wie zuvor fallen die Unterschiede abhängig von der Komplexität der DAT-Dateien unterschiedlich stark aus.

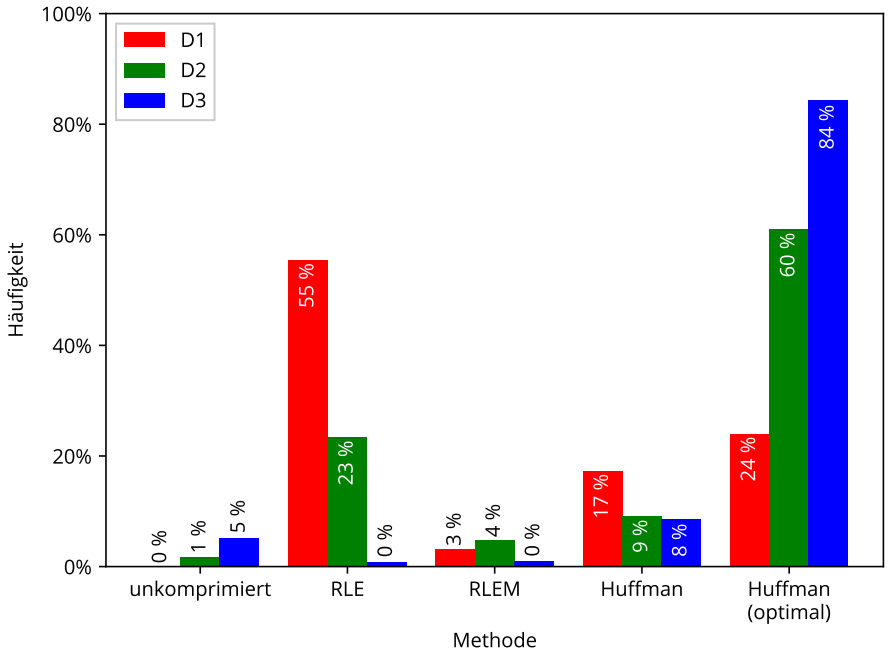


**Abbildung 5.3:** Verteilung der automatisch bestimmten Kompressionsmethoden bei simulierter Übertragung mit Huffman-Kodierung.

In den Abbildungen 5.3 (mit Huffman-Kodierung) und 5.4 (ohne Huffman-Kodierung) ist die Häufigkeit der Verwendung der Kompressionsmethoden bei segmentspezifischer Wahl der optimalen Kompressionsmethode aufgetragen. Bei beiden Abbildungen kann man erkennen, dass die Verwendung von RLE mit steigender Komplexität der DAT-Dateien abnimmt. Erlaubt man die Verwendung der Huffman-Kodierung, spielen unkomprimierte und RLEM-komprimierte Segmente eine geringe Rolle; der größte Teil wird mittels RLE und Huffman-Kodierung komprimiert. Erlaubt man die Verwendung der Huffman-Kodierung nicht, nimmt die Anzahl der unkomprimierten Segmente stark zu und RLEM ersetzt zwangsweise die Huffman-Kodierung.

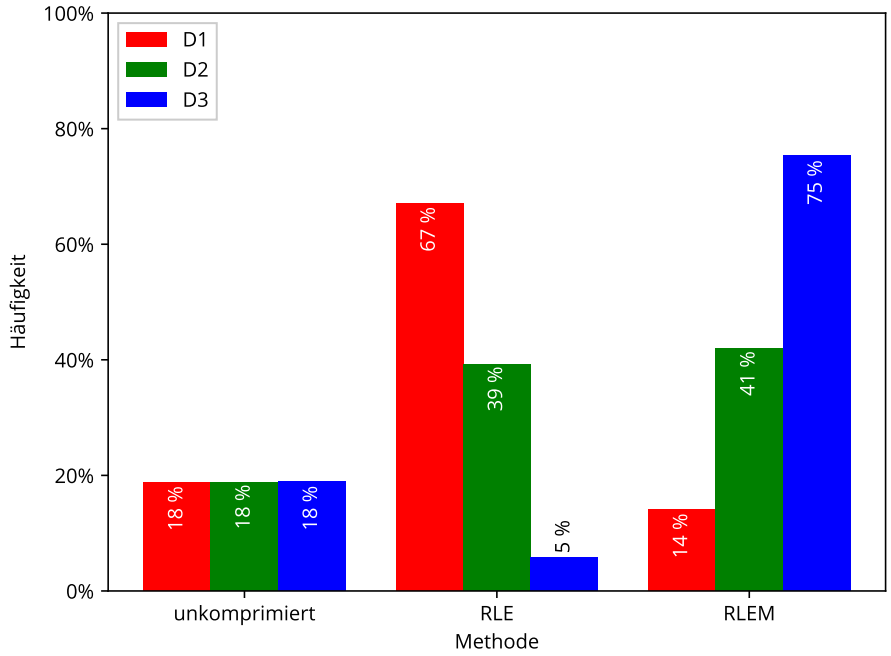


**Abbildung 5.4:** Verteilung der automatisch bestimmten Kompressionsmethoden bei simulierter Übertragung ohne Huffman-Kodierung.

## 5.2 Speicherbedarf der Firmware

In Tabelle 5.1 ist der Speicherbedarf der OTAP-Firmware auf dem Mikrocontroller, jeweils unterteilt in die verschiedenen Speicherbereiche, angegeben. Die Firmware wurde mit der für die Energiemessung benutzten Version zur Verwendung auf dem HaLOEWEn erstellt. Wegen der in Abschnitt 3.8.8 beschriebenen Abweichungen der Größe bei verschiedenen Durchläufen von SDCC wurde die Firmware fünfmal neu erstellt. Der Speicherbedarf wurde aus den von SDCC generierten MEM-Datein ermittelt. Im normalen Einsatz als Teil einer eigentlichen Anwendung in einem drahtlosen Sensornetwerk wäre die tatsächliche durch die OTAP-Funktionalität verursachte Größe kleiner, da hier der Funk-Stack und die dazugehörigen Puffer (in `XDATA`) enthalten sind. Im `DATA`-Segment werden 117 Byte, im `XDATA`-Segment 528 Byte und (20275,6 ± 8,6) Byte im `CODE`-Segment verwendet. In Abbildung 5.2 sind die entsprechenden Werte mit aktivierter Huffman-Kodierung aufgetragen. Im Vergleich zu den Werten ohne Huffman-Kodierung werden 2 Byte im `XDATA`-Segment zusätzlich benötigt. Der größte Unterschied ist im `CODE`-Segment zu finden: Mit Huffman-Kodierung werden 21928,6 ± 6,2 Byte benötigt, das sind ca. 1653 Byte zusätzlich.

| Vorgang | `DATA` | `XDATA` | `CODE` |
|---|---|---|---|
| 1 | 117 | 528 | 20277 |
| 2 | 117 | 528 | 20286 |
| 3 | 117 | 528 | 20262 |
| 4 | 117 | 528 | 20277 |
| 5 | 117 | 528 | 20276 |
| | 117 | 528 | 20275,6 ± 8,6 |

**Tabelle 5.1:** Speicherbedarf der Firmware in der für die Energiemessung verwendeten Version ohne Huffman-Kodierung in Byte.

| Vorgang | DATA | XDATA | CODE |
|---|---|---|---|
| 1 | 117 | 530 | 21936 |
| 2 | 117 | 530 | 21929 |
| 3 | 117 | 530 | 21928 |
| 4 | 117 | 530 | 21931 |
| 5 | 117 | 530 | 21919 |
| | 117 | 530 | 21928,6 ± 6,2 |

**Tabelle 5.2:** Speicherbedarf der Firmware in der für die Energiemessung verwendeten Version mit Huffman-Kodierung in Byte.

## 5.3 Konfigurationszeiten

In allen folgenden Diagrammen (Abbildungen 5.5 bis 5.14) steht jeder Balken i. a. für den Mittelwert von zehn Messungen. Die Standardabweichungen der gemessenen Zeiten liegen zwischen 0.01 s und 0.50 s und sind deshalb bei einem Wertebereich der Y-Achse von 0 s bis 750 s nicht zu erkennen. Die Messungen erfolgen mit dem in Abschnitt 4.4.2 beschriebenen Aufbau.

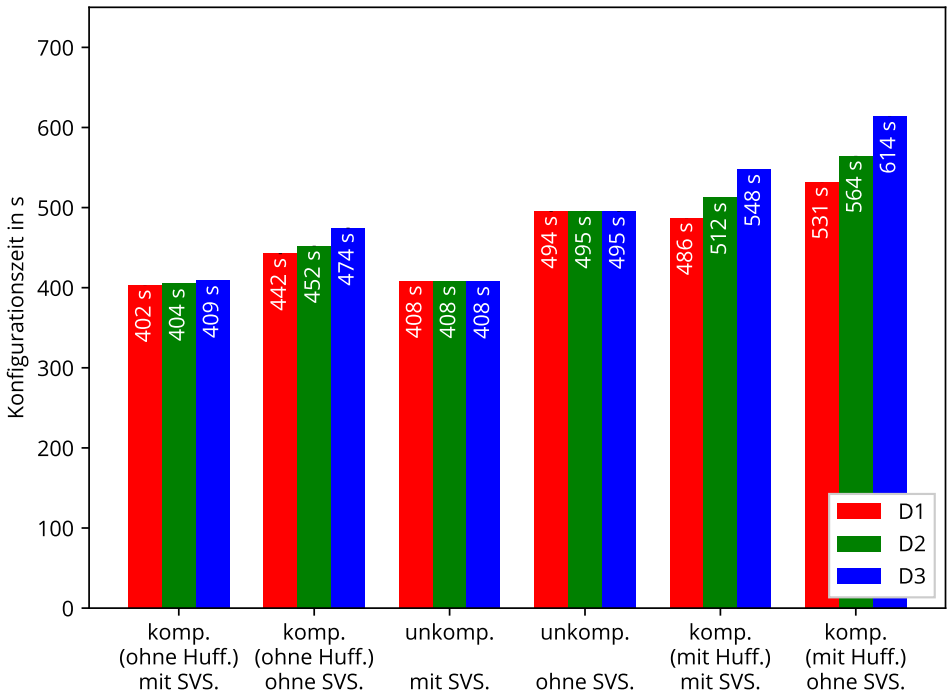


**Abbildung 5.5:** Konfigurationszeit bei Segmentlänge von 100 Byte und kabelgebundener Übertragung.

### 5.3.1 Kabelgebundene und drahtlose Übertragungen ohne Vorhersagefehler

In Abbildungen 5.5 und 5.6 ist die für einen Konfigurationsvorgang benötigte Zeit (d. h. die Konfigurationszeit) inklusive Löschen, Schreiben und Verifizieren des Flash-Speichers (siehe Abschnitt 3.8) für kabelgebundene bzw. kabellose Übertragung bei einer angefragten Segmentlänge von 100 Byte aufgetragen. Wie bei den vorherigen Diagrammen stehen die unterschiedlichen Balkengruppen auf der x-Achse für verschiedene Übertragungsparameter. Die Balken innerhalb der Gruppen stehen für die verschiedenen Konfigurationsdateien D1 bis D3. Von links nach rechts ist die komprimierte Übertragung

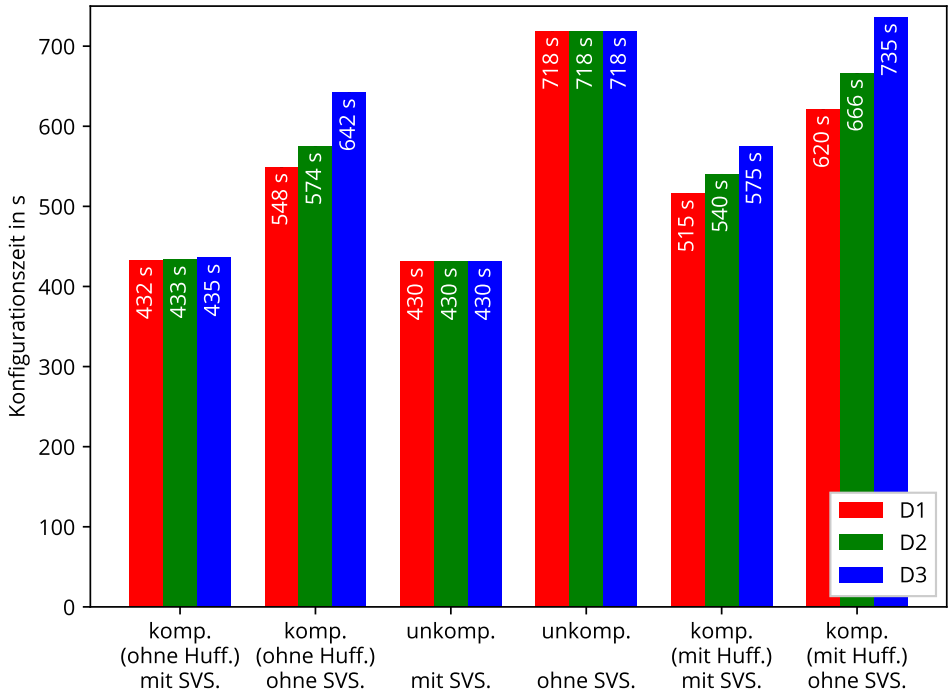


**Abbildung 5.6:** Konfigurationszeit bei Segmentlänge von 100 Byte und kabelloser Übertragung.

(ohne Huffman-Kodierung) zunächst mit und dann ohne Vorhersage angegeben, darauf folgt die unkomprimierte Übertragung mit und ohne Vorhersage, zuletzt folgt die komprimierte Übertragung (mit Huffman-Kodierung) mit und ohne Vorhersage.

Betrachtet man die ersten vier Gruppen in Abbildung 5.5, fällt auf, dass wie erwartet die unkomprimierte Übertragung ohne Vorhersage (vierte Gruppe) mit ca. 500 s am längsten dauert. Aktiviert man eine komprimierte Übertragung ohne die Verwendung der Huffman-Kodierung (zweite Gruppe) bei kabelgebundener Übertragung, wird eine Zeitersparnis von ca. 20 s bis 50 s (je nach verwendeter Konfigurationsdatei) ermöglicht. Die Verwendung der Segmentvorhersage (Gruppe 1 und Gruppe 3) bringt eine deutliche Verbesserung. Im unkomprimierten Fall wird die Konfigurationszeit durch die Segmentvorhersage um ca. 85 s reduziert. Im Vergleich der ersten vier Gruppen wird deutlich, dass nur bei komprimierter Übertragung ohne Vorhersage eine deutliche Änderung der Programmierzeit beim Wechsel der Konfigurationsdatei auftritt. Wie erwartet ist in diesem Fall der Konfigurationsvorgang für die einfachste Konfigurationsdatei am schnellsten. Mit zunehmender Komplexität der Konfigurationsdatei steigt die benötigte Zeit an. Beim Vergleich der komprimierten Übertragung ohne Huffman-Kodierung (Gruppe 1 und 2) mit der komprimierten Übertragung mit Huffman-Kodierung (Gruppe 5 und 6) sieht man, dass die Verwendung der Huffman-Kodierung eine deutlich längere Dauer des Konfigurationsvorgangs von bis zu 140 s (D3 ohne Vorhersage) verursacht. Bei kabelgebundener Übertragung dauert der Konfigurationsvorgang mit Huffman-Kodierung in den meisten Fällen sogar länger als der komplett unoptimierte Vorgang ohne Kompression und ohne Vorhersage. Bei aktivierter Vorhersage unterscheiden sich die Zeiten für unkomprimierte und komprimierte Übertragung (ohne Huffman-Kodierung) nur minimal.

Bei Funkübertragung (Abbildung 5.6) ändert sich im Vergleich zur kabelgebundenen Übertragung wenig am relativen Verhältnis der verschiedenen Konfigurationszeiten. Bei den ersten vier Gruppen ist wie gehabt die unkomprimierte Übertragung ohne Vorhersage (Gruppe 4) mit ca. 720 s am langsamsten. Durch Verwendung von Kompression ohne Huffman-Kodierung (Gruppe 2) können ca. 76 s bis 170 s eingespart werden. Wie zuvor bringt die Verwendung der Segmentvorhersage (Gruppe 1 und 3) die größte Zeitersparnis von ca. 285 s. Die Verwendung der Huffman-Kodierung (Gruppe 5 und 6) bringt wieder eine deutliche Verschlechterung von ca. 80 s bis 140 s ohne Vorhersage und von ca. 70 s bis

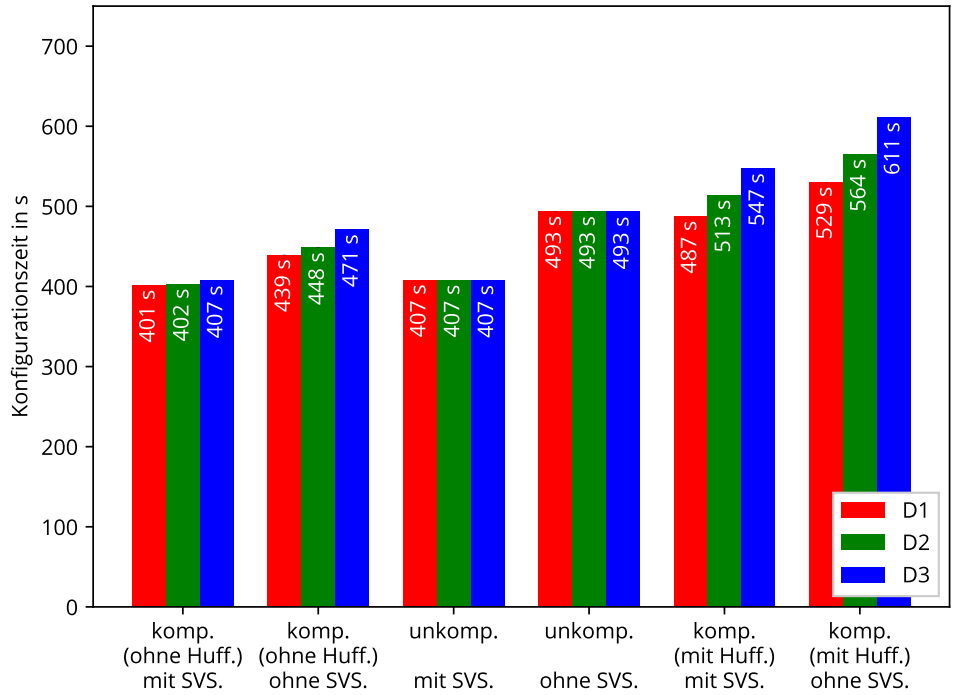


**Abbildung 5.7:** Konfigurationszeit bei Segmentlänge von 111 Byte und kabelgebundener Übertragung.

90 s mit Vorhersage. Immerhin ist die Funkübertragung so viel langsamer als die kabelgebundene Übertragung, dass nur die komprimierte Übertragung mit Huffman-Kodierung von D3 ohne Vorhersage mit einer Konfigurationszeit von 735 s langsamer ist als die unkomprimierte Übertragung ohne Vorhersage.

In Abbildungen 5.7 und 5.8 sind analog dazu die Konfigurationszeiten bei kabelgebundener bzw. kabelloser Übertragung mit einer Segmentlänge von 111 Byte aufgetragen. Beim Vergleich der Konfigurationszeiten der kabelgebundenen Übertragung (Abbildung 5.7) mit denen für eine Segmentlänge von 100 Byte kann keine relevante Abweichung festgestellt werden. Die meisten Konfigurationszeiten sind lediglich um ca. 1 s kürzer geworden. Beim Vergleich der Konfigurationszeiten bei Funkübertragung (Abbildung 5.8) fällt auf, dass durch Erhöhung der Segmentgröße nur in den Fällen ohne Vorhersage eine geringe, aber nicht vernachlässigbare Verringerung um ca. 5 s bis 7 s eintritt. Bei aktivierter Vorhersage ist die unkomprimierte Übertragung um ca. 4 s bis 7 s schneller als die komprimierte Übertragung ohne Huffman-Kodierung.

**Interpretation:**

Die Verwendung der Huffman-Kodierung bringt in keinem der untersuchten Fälle eine Verbesserung, da der zusätzliche Zeitaufwand für die Dekodierung den Vorteil durch das geringere Datenvolumen aufbraucht. Aus diesem Grund wird sie im Folgenden nicht mehr betrachtet. Bei kabelgebundener Übertragung zeigt eine Erhöhung der Segmentlänge um ca. 10 % nur eine geringfügige Auswirkung. Bei der Funkübertragung ohne Vorhersage kommt es zu einer Änderung, die vermutlich durch den geringeren Anteil an Overhead durch die niedrigere Gesamtzahl an Funkpaketen verursacht wird. Bei aktivierter Vorhersage wird dieser Unterschied durch die Übertragung im Hintergrund verdeckt. Die Tatsache, dass bei aktivierter Vorhersage die unkomprimierte Funkübertragung am schnellsten ist, widerlegt nicht die Hypothese des optimalen Übertragungsmodus (siehe Abschnitt 3.8.9), die im folgenden Abschnitt genauer untersucht wird. Durch die Verwendung der Segmentvorhersage kann der Nachteil der deutlich langsameren Übertragungsrate der Funkverbindung fast vollständig kompensiert werden, es kommt im Vergleich zur kabelgebundenen Übertragung lediglich zu einer Verlängerung der Konfigurationszeit um ca. 20 s bis 30 s, während der Konfigurationsvorgang ohne Vorhersage bis zu 220 s länger dauert.

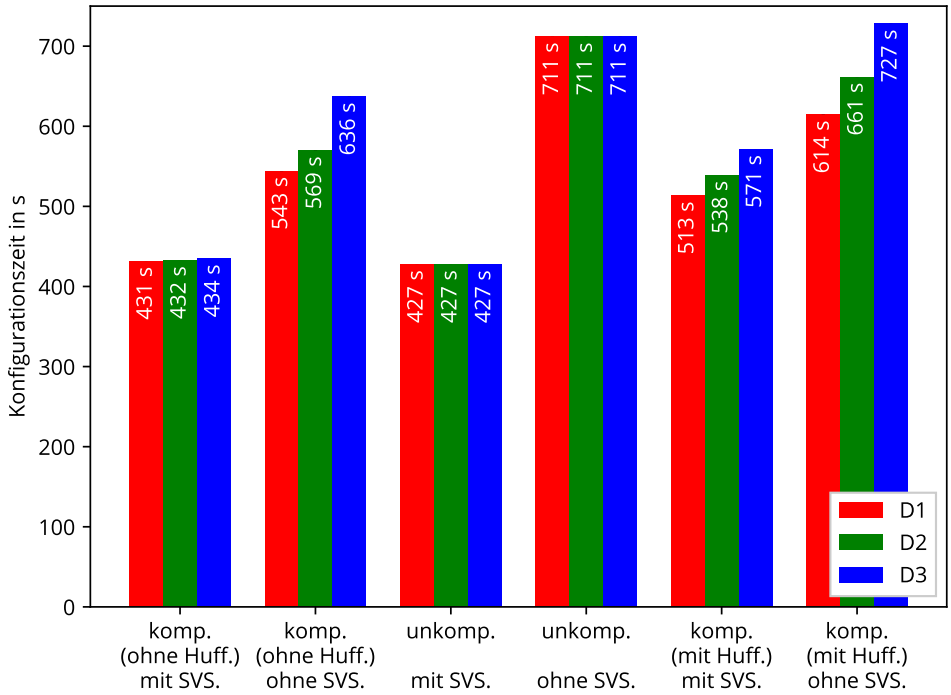


**Abbildung 5.8:** Konfigurationszeit bei Segmentlänge von 111 Byte und kabelloser Übertragung.

### 5.3.2 Kabelgebundene und drahtlose Übertragungen mit Vorhersagefehler

In diesem Abschnitt wird die Hypothese des optimalen Übertragungsmodus (siehe Abschnitt 3.8.9) untersucht. In den Abbildungen 5.9 und 5.10 ist die Konfigurationszeit für Konfigurationsvorgänge mit einem absichtlich durch einen Fehler in der Mitte der Vorhersagedaten herbeigeführten Vorhersagefehler aufgetragen. Wie gehabt zeigt Abbildung 5.9 die Daten für die kabelgebundene Übertragung und 5.10 die Daten für die kabellose Übertragung. Die erste Balkengruppe („optimaler Modus“) zeigt jeweils die Messwerte für eine unkomprimierte Übertragung mit Vorhersage bis zum Auftreten des Vorhersagefehlers. Danach wird zu komprimierter Übertragung ohne Vorhersage gewechselt. Die zweite Balkengruppe zeigt die Konfigurationszeiten für eine dauerhaft komprimierte Übertragung, während die dritte Balkengruppe die Zeiten für eine dauerhaft unkomprimierte Übertragung zeigt. Wie erwartet ist die komplett unkomprimierte Übertragung die langsamste Option. Die Messwerte der ersten und zweiten Balkengruppe zeigen keine signifikanten Unterschiede. Die erhoffte Verbesserung durch die Deaktivierung der Kompression, während die Übertragung mit Segmentvorhersage durchgeführt wird, was theoretisch zum Einsparen der für die Dekompression benötigten Zeit führt, kann mit diesem Experiment weder bestätigt noch widerlegt werden.

**Interpretation:**

Es kann keine schlüssige Aussage getroffen werden, ob es einen Vorteil hat, die Übertragung unkomprimiert durchzuführen, solange die Segmentvorhersage genutzt werden kann. Auf jeden Fall ist klar, dass es nicht sinnvoll ist, auf eine komprimierte Übertragung zu verzichten, falls keine Vorhersage verwendet werden kann.

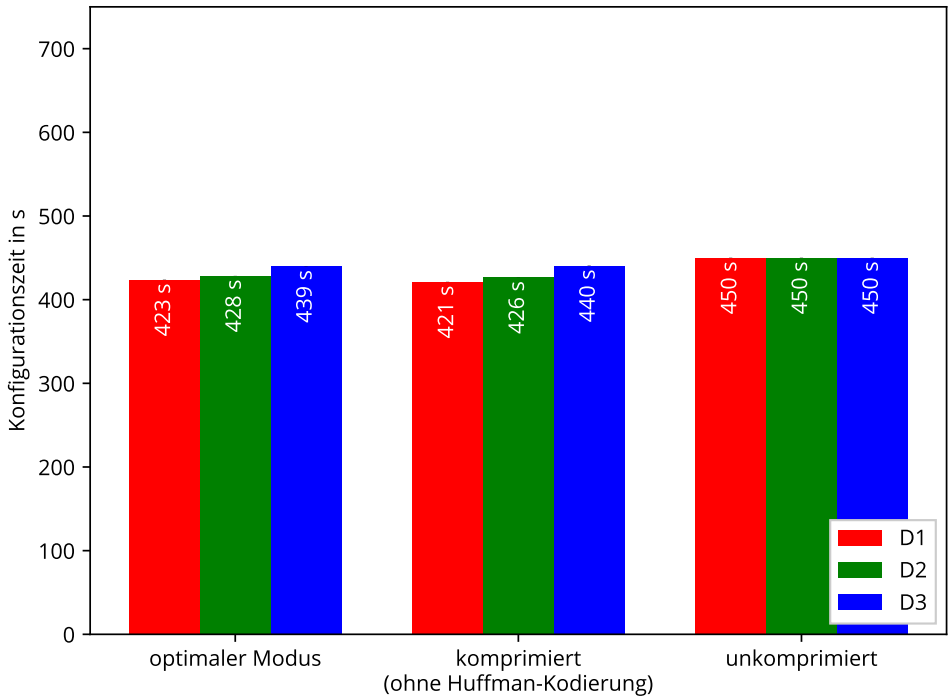


**Abbildung 5.9:** Konfigurationszeit mit absichtlichen Vorhersagefehler bei Segmentlänge von 111 Byte und kabelgebundener Übertragung.

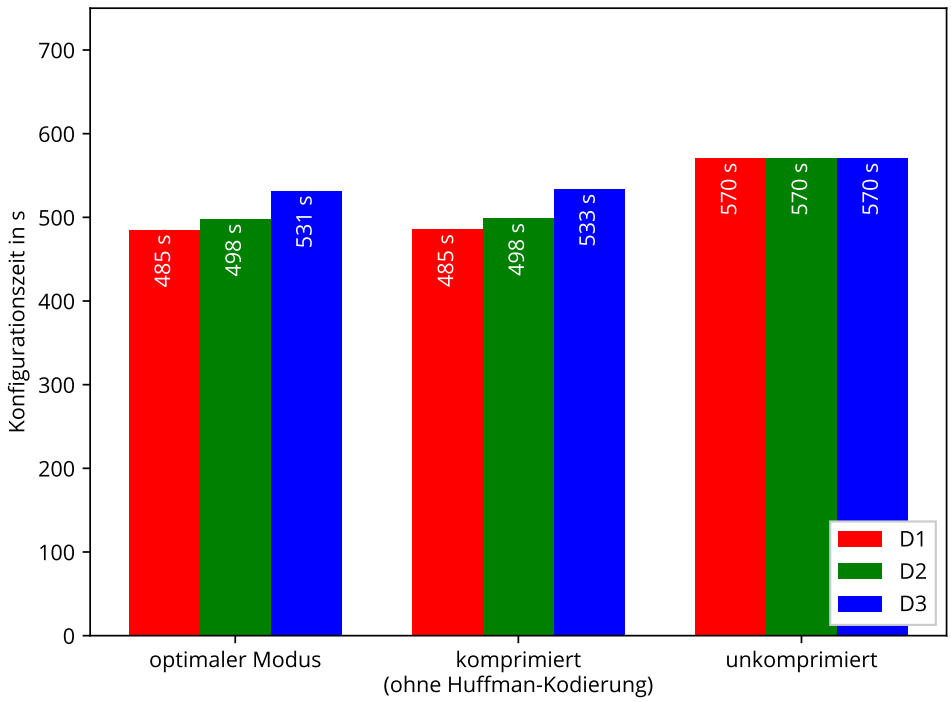


**Abbildung 5.10:** Konfigurationszeit mit absichtlichen Vorhersagefehler bei Segmentlänge von 111 Byte und kabelloser Übertragung.

### 5.3.3 Einfluss der Segmentlänge bei kabelgebundener und drahtloser Übertragung

In diesem Abschnitt wird der Einfluss der eingestellten Segmentlänge auf die Konfigurationszeit bei komprimierter Übertragung (ohne Huffman-Kodierung) untersucht. Alle in diesem Abschnitt untersuchten Ergebnisse stammen von Konfigurationsvorgängen mit dieser Übertragungsart. Aus Gründen der Übersichtlichkeit wird dies nicht jedes Mal erwähnt.

Wie in Abschnitt 3.8.2 erwähnt, werden zwei verschiedene Kompressionsstrategien verwendet. Bei einer Segmentlänge von maximal 111 Byte wird mit einer festen Rohdatenmenge gearbeitet. Die übertragenen (komprimierten) Segmente können also, falls sie komprimiert sind, kürzer als die jeweils angegebene Segmentlänge sein. Bei größeren Segmentlängen wird mit einer variablen Rohdatenmenge gearbeitet. In diesem Fall dient die eingestellte Segmentlänge als Obergrenze der unkomprimierten Rohdatenmenge. Somit wird die Länge der (komprimierten) Segmente auf 111 Byte begrenzt.

In Abbildung 5.11 sind die Konfigurationszeiten für verschiedene eingestellte Segmentlängen bei Übertragung per Kabel aufgetragen. Von der kürzesten Segmentlänge (64 Byte) bis zu längsten (255 Byte) nimmt die Gesamtzeit um ca. 22 s ab. Vergleicht man die Zeiten der dritten Balkengruppe (111 Byte) mit den Zeiten der beiden extremen Segmentlängen, fällt auf, dass die Vergrößerung zur kleinsten Segmentlänge mit ca. 16 s deutlich größer ausfällt als die Verkleinerung zur größten Segmentlänge mit ca. 6 s.

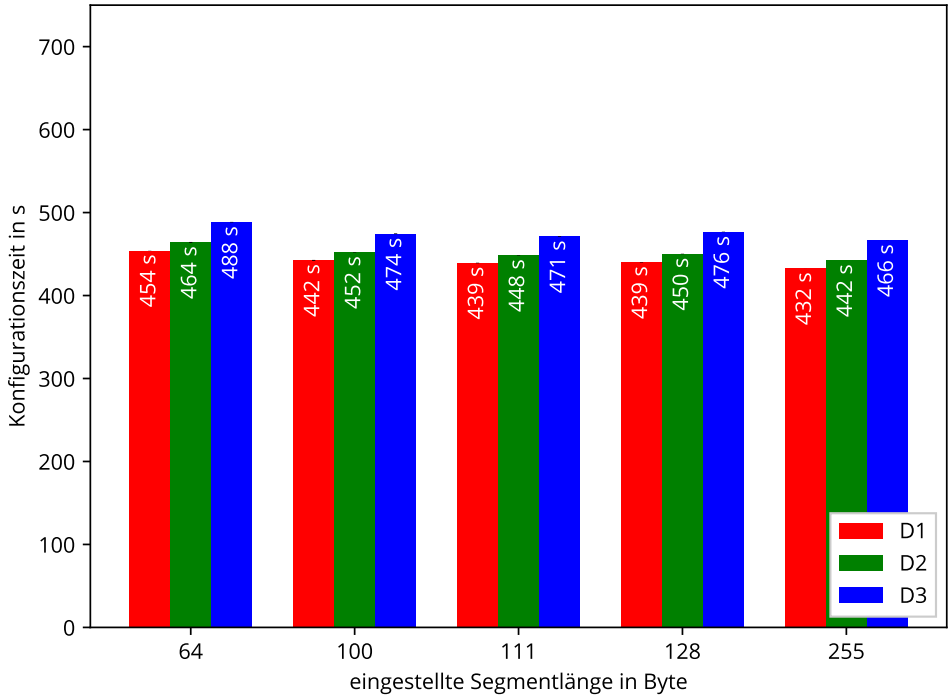


**Abbildung 5.11:** Konfigurationszeit nach eingestellter Segmentlänge bei kabelgebundener Übertragung mit Kompression (ohne Huffman-Kodierung) ohne Vorhersage.

Bei den Konfigurationszeiten bei kabelgebundener Übertragung mit Vorhersage, die in Abbildung 5.12 dargestellt sind, gibt es nur geringe Unterschiede. Zwischen der kürzesten Segmentlänge und der längsten Segmentlänge liegt eine Zeitersparnis von lediglich 5 s. Der größte Änderungsanteil tritt schon beim Wechsel von 64 Byte zu 100 Byte auf, durch den ca. 3 s weniger benötigt werden.

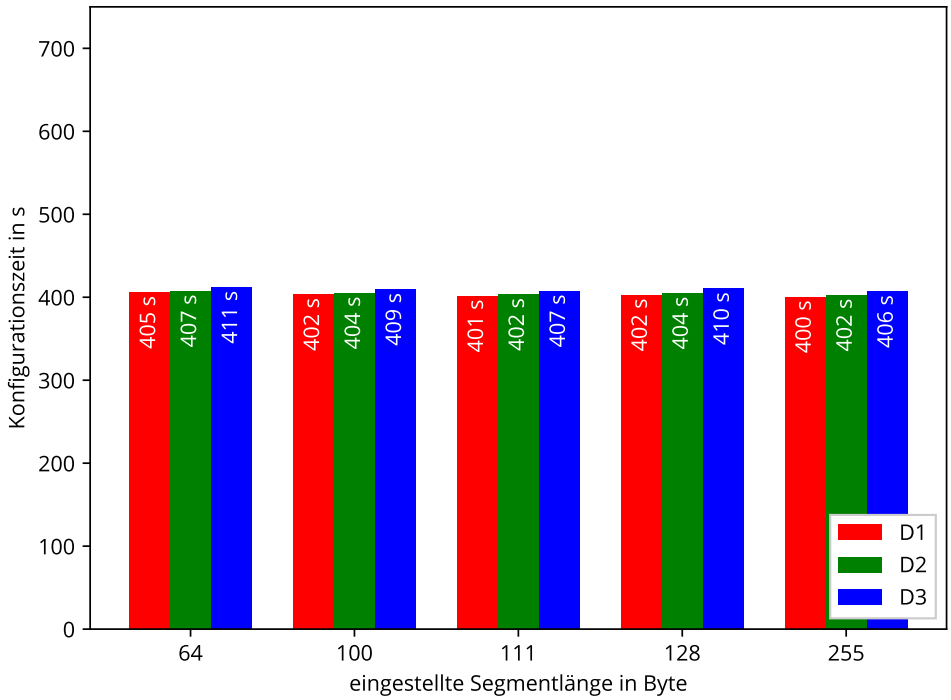


**Abbildung 5.12:** Konfigurationszeit nach eingestellter Segmentlänge bei kabelgebundener Übertragung mit Kompression (ohne Huffman-Kodierung) mit Vorhersage.

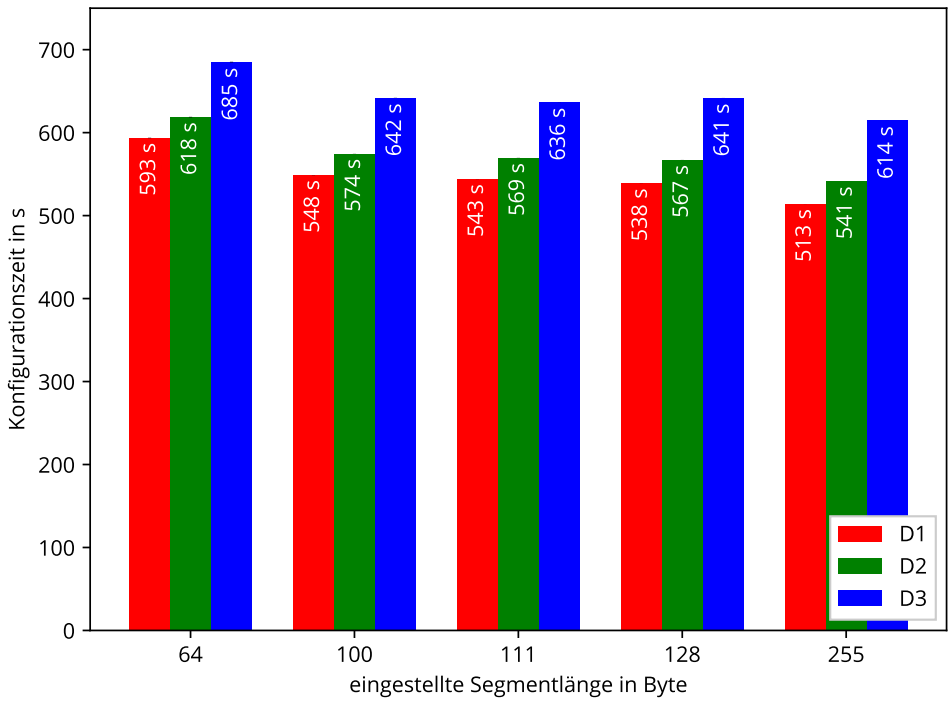


**Abbildung 5.13:** Konfigurationszeit nach eingestellter Segmentlänge bei kabelloser Übertragung mit Kompression (ohne Huffman-Kodierung) ohne Vorhersage.

In Abbildung 5.13 sind die Konfigurationszeiten für die kabellose Übertragung ohne Vorhersage aufgetragen. Wie gehabt haben bei der Übertragung per Funk die unterschiedlichen Komprimierbarkeiten der untersuchten Konfigurationsdateien stärkere Auswirkungen auf die Konfigurationszeiten als bei der kabelgebundenen Übertragung. Beim Wechsel von der kleinsten Segmentlänge von 64 Byte (erste Gruppe) zur größten Segmentlänge von 255 Byte (fünfte Gruppe) können ca. 75 s eingespart werden. Wie bei der kabelgebundenen Übertragung entfällt der größte Teil der Zeitersparnis auf den Wechsel von 64 Byte (erste Gruppe) zu 111 Byte (dritte Gruppe). Bei der Funkübertragung sind dies ca. 50 s, während beim Wechsel von 111 Byte zu 255 Byte lediglich ca. 25 s eingespart werden können.

In Abbildung 5.14 sind die Konfigurationszeiten für die kabellose Übertragung mit Vorhersage aufgetragen. Wie im kabelgebundenen Fall ist die Auswirkung einer Änderung der Segmentlänge bei der Übertragung mit Vorhersage deutlich geringer als bei der Übertragung ohne Vorhersage. Trotzdem sind die Unterschiede zwischen den einzelnen Segmentlängen bei der Funkübertragung deutlich größer als bei der kabelgebundenen Übertragung. Zwischen der kürzesten Segmentlänge von 64 Byte (erste Balkengruppe) und der größten Segmentlänge von 255 Byte (fünfte Gruppe) liegt eine Verringerung der Konfigurationszeit um ca. 40 s. Wie gehabt tritt der größte Teil dieser Verringerung schon beim Wechsel von 64 Byte zu 111 Byte (dritte Gruppe) auf. In diesem Fall gibt es eine Zeitersparnis von ca. 25 s. Auf den Wechsel von 111 Byte zu 255 Byte (fünfte Gruppe) entfallen die restlichen ca. 15 s.

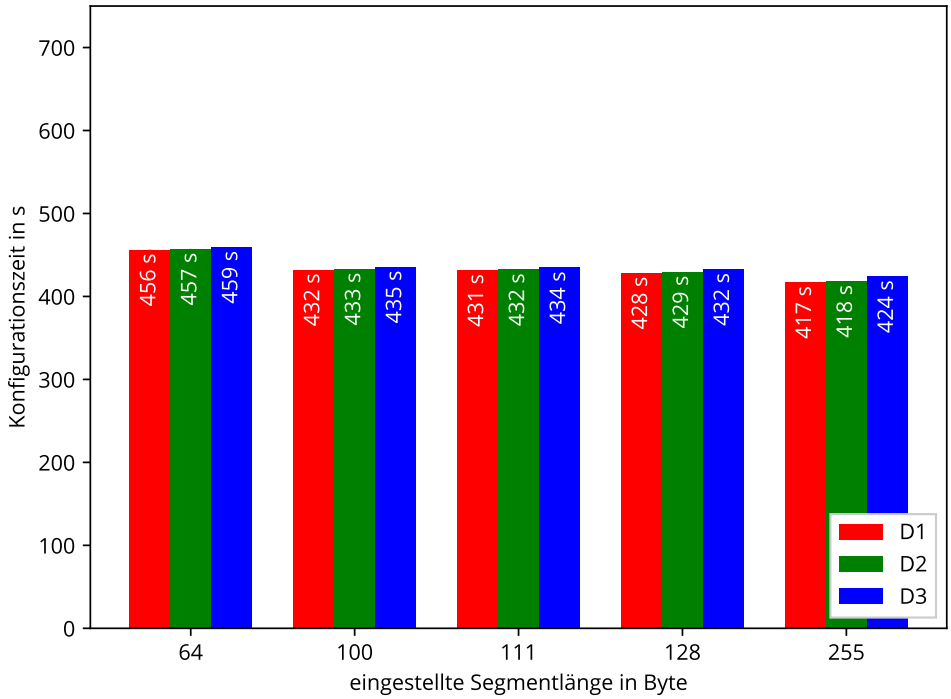


**Abbildung 5.14:** Konfigurationszeit nach eingestellter Segmentlänge bei kabelloser Übertragung mit Kompression (ohne Huffman-Kodierung) mit Vorhersage.

In Abbildung 5.15 ist die mittlere unkomprimierte Segmentlänge für die verschiedenen verwendeten eingestellten Segmentlängen aufgetragen. Die mittleren unkomprimierten Segmentlängen sind, bis auf die der letzte Balkengruppe, innerhalb der Gruppen gleich. Für die ersten drei Gruppen (64 Byte, 100 Byte und 111 Byte) kann dies nicht anders sein, da das Segment vollständig unkomprimiert in einem Funkpaket übertragen werden kann und es somit zu keiner Begrenzung während des Kompressionsvorgangs kommen kann. Bei eingestellter Segmentlänge von 128 Byte hat die verwendete Konfigurationsdatei ebenfalls keine Auswirkung auf die mittlere unkomprimierte Segmentlänge. Bei einer eingestellten Segmentlänge von 255 Byte ist die mittlere unkomprimierte Segmentlänge bei der Konfigurationsdatei D3 um ca. 35 Byte kleiner als die der anderen beiden Konfigurationsdateien. Dies liegt vermutlich an der

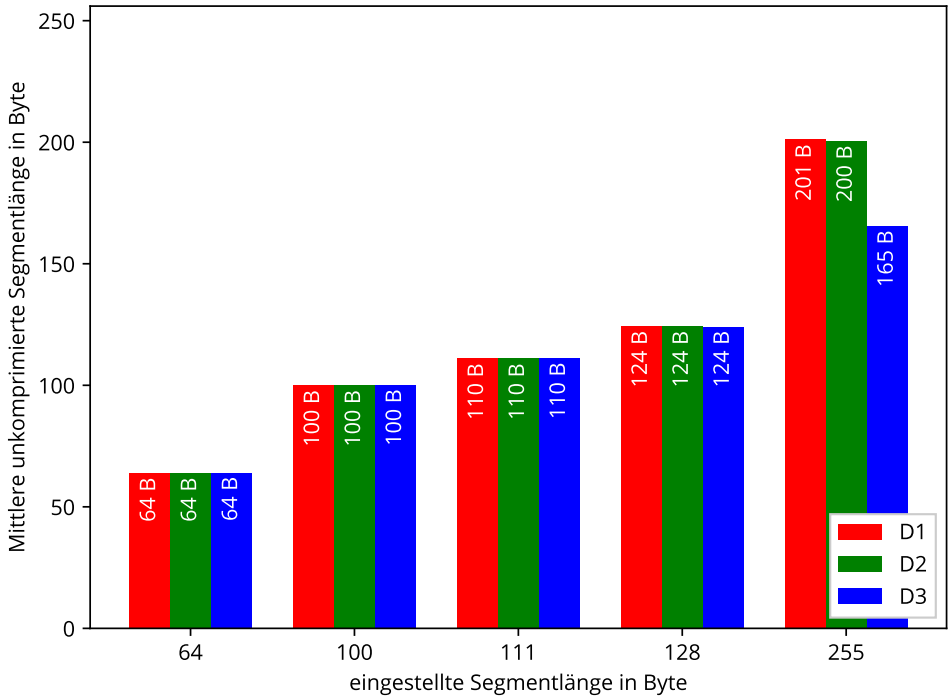


**Abbildung 5.15:** Mittlere unkomprimierte Segmentlänge nach eingestellter Segmentlänge bei Übertragung mit Kompression (ohne Huffman-Kodierung).

schlechteren Komprimierbarkeit dieser Datei, die dazu führt, dass weniger Rohdaten in komprimierter Form in ein Funkpaket passen.

In Abbildung 5.16 ist die mittlere komprimierte Segmentlänge für die verschiedenen eingestellten Segmentlängen aufgetragen. Man kann deutlich die erwartete unterschiedlich gute Komprimierbarkeit der verschiedenen Konfigurationsdateien erkennen (siehe auch Abschnitt 5.1). Der Balken für die mittlere komprimierte Segmentlänge für die Konfigurationsdatei D3 bei einer eingestellten Segmentlänge von 255 Byte ist nahe an den maximal möglichen 111 Byte. Dies stützt die bei der Interpretation des vorherigen Diagramms genannte Vermutung, dass durch die schlechtere Komprimierbarkeit von D3 bei der Kompression der Segmente weniger Rohdaten verwendet werden können.

**Interpretation:**

Die möglichen Zeiteinsparungen bei der Verwendung von Segmentlängen, die größer als 111 Byte sind, sind bei der Übertragung mit Vorhersage zu gering. Da die Menge an Rohdaten pro Segment in diesem Fall abhängig von der Komprimierbarkeit der Segmente und damit von der der Konfigurationsdateien ist, sind die benötigten Vorhersagedaten nicht mehr unabhängig von der verwendeten Konfigurationsdatei. Dies ist ein großer Nachteil, der für einen kleinen Vorteil in Kauf genommen werden muss. Ein weiterer Nachteil ist die Tatsache, dass DirectC einen größeren Puffer benötigt, um die Daten auf dem Mikrocontroller zu speichern, was bei den knappen Ressourcen des CC2530 nicht wünschenswert ist. Wegen der unterschiedlich starken Komprimierbarkeit der Segmente ist nicht klar, ob dieser vergrößerte Puffer während des Konfigurationsvorgangs vollständig genutzt wird.

Wird der Konfigurationsvorgang hingegen ohne Vorhersage durchgeführt, kann die Verwendung von Segmentlängen größer als 111 Byte sinnvoll sein, da in diesem Fall die Zeitersparnis, je nach verwendeter Konfigurationsdatei, größer sein kann. Der Nachteil des größeren Puffers und des damit erhöhten Speicherbedarfs auf dem Mikrocontroller bleibt aber weiterhin bestehen.

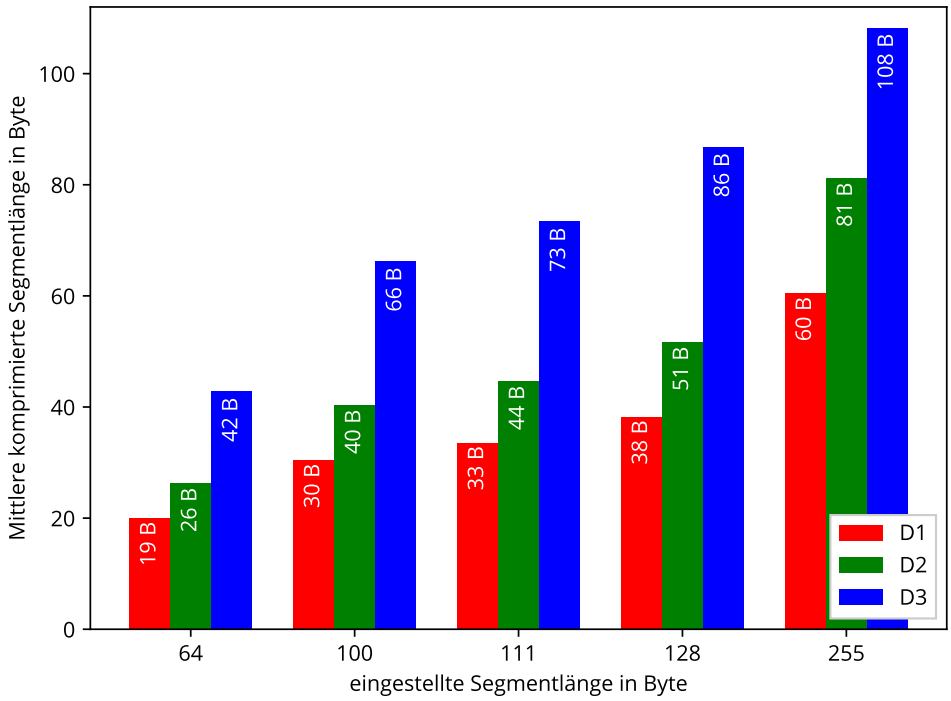


**Abbildung 5.16:** Mittlere komprimierte Segmentlänge nach eingestellter Segmentlänge bei Übertragung mit Kompression (ohne Huffman-Kodierung).

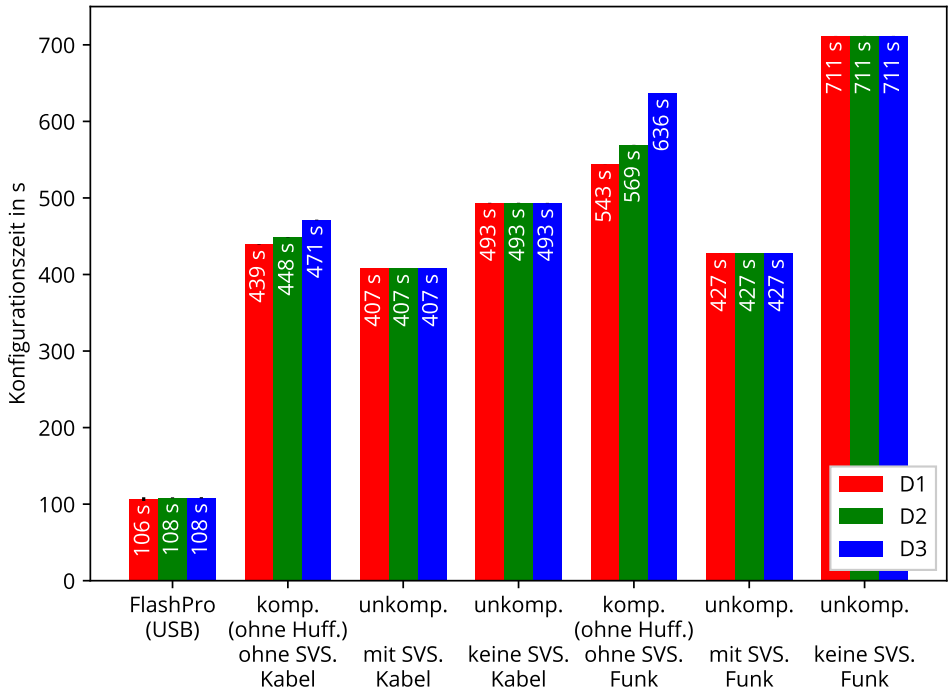


**Abbildung 5.17:** Vergleich der Konfigurationszeiten für 111 Byte Segmentlänge mit denen von FlashPro.

### 5.3.4 Vergleich mit FlashPro

In Abbildung 5.17 ist die Konfigurationszeit bei der Verwendung von FlashPro, dem offiziellen Programmiergerät für Microsemi FPGAs, zusammen mit einigen ausgewählten Konfigurationszeiten bei kabelgebundener und kabelloser Übertragung mit einer Segmentlänge von 111 Byte aufgetragen. Man kann erkennen, dass die Konfiguration mittels DirectC unabhängig vom verwendeten Übertragungskanal (d. h. Kabel oder Funk) sogar im bestmöglichen Fall der Konfiguration mit Vorhersage (Gruppe 3 und 6) ca. um den Faktor 4 langsamer ist als FlashPro (Gruppe 1). Trotzdem ist durch die möglichen Optimierungen eine deutliche Verbesserung eingetreten, da die unkomprimierte Übertragung ohne Vorhersage ca. um den Faktor 5 (mit Kabel, Gruppe 4) bzw. um den Faktor 7 (per Funk, Gruppe 7) schlechter als FlashPro ist. Im Ursprungszustand von DirectC, d. h. ohne die zusätzlichen Optimierungen wie die Verlagerung der Prüfsummenberechnung auf den PC, die wie in Abschnitt 3.8.1 erwähnt alleine eine Einsparung von ca. 90 s bringt, würde die Konfiguration per CC2530 im Vergleich zu FlashPro noch schlechter ausfallen.

## 5.4 Energiebedarf eines Konfigurationsvorgangs

Zunächst wird der Stromfluss über die beiden Versorgungsleitungen für einen typischen Konfigurationsvorgang untersucht und dabei eine Abschätzung des Energiebedarfs vorgenommen. Danach folgt eine genauere Untersuchung des Energiebedarfs in Abhängigkeit von verschiedenen Übertragungsparametern. Die Messungen erfolgen mit dem in Abschnitt 4.4.2 beschriebenen Aufbau.

### 5.4.1 Stromfluss über die beiden Versorgungsleitungen

In Abbildung 5.18 ist der zeitliche Verlauf des Stromflusses auf beiden Versorgungsleitungen für einen beispielhaften Konfigurationsvorgang aufgetragen. Dabei sind die unterschiedlichen Phasen des Konfi-

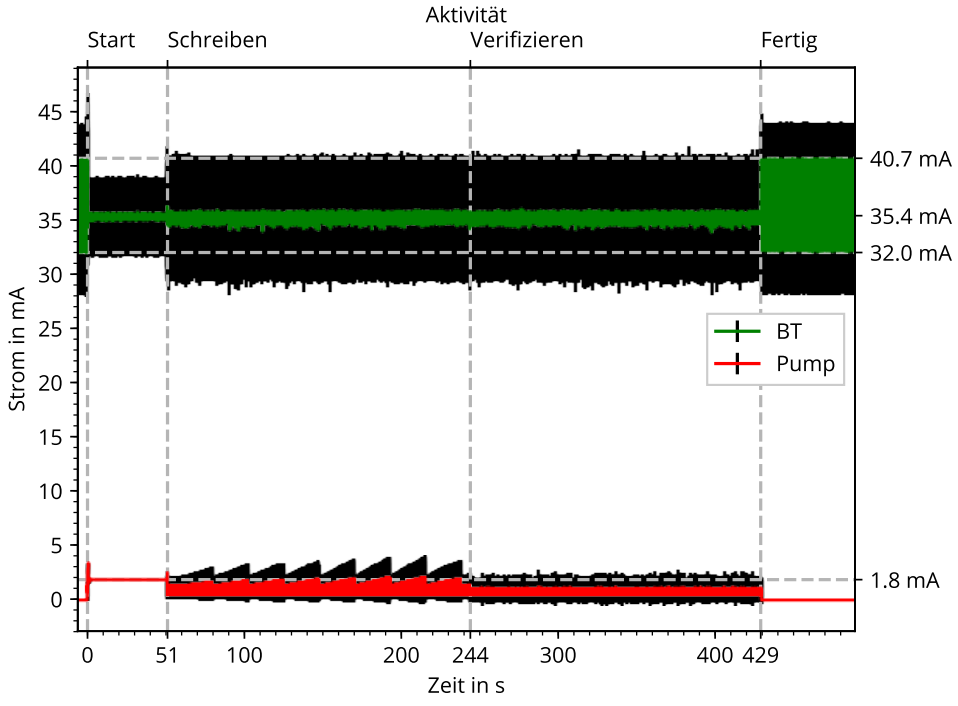


**Abbildung 5.18:** Verlauf von $I_{BT}$ und $I_{Pump}$ während eines Konfigurationsvorgangs von D1 mit Kompression (ohne Huffman-Kodierung) und Vorhersage.

gurationsvorgangs markiert. Die Informationen über die jeweiligen Zeitpunkte für Schreiben und Verifizieren stammen aus dem Ereignisprotokoll der Gegenstelle auf dem PC und sind wegen der gepufferten Übertragung von nicht-kritischen Nachrichten möglicherweise minimal verzögert. Der Abgleich der unterschiedlichen Zeiten des Ereignisprotokolls zu den Messwerten wurde anhand der positiven Kante auf dem Triggereingang, der den Start des Konfigurationsvorgangs auf dem Mikrocontroller markiert, durchgeführt. Zur Kontrolle wurden die Zeitpunkte der negativen Kante auf dem Triggereingang, der das Ende des Konfigurationsvorgangs markiert, mit dem Protokoll der Gegenstelle erfolgreich verglichen. Unter der Annahme, dass die Zeitverzögerung zwischen dem Start einer Phase von DirectC und dem Eintreffen der Nachricht am PC für alle Nachrichten gleich groß ist, wird diese Verzögerung durch die angewendete Vorgehensweise herausgerechnet.

Da die Messwerte mit einer Samplingrate von 75 kHz aufgezeichnet wurden und die dadurch verursachte Datenmenge von ca. 2 GByte pro Messung ungünstig für die Erstellung von Plots ist, wurden in einem Vorverarbeitungsschritt der Mittelwert und die Standardabweichung von jeweils 750 Messwerten gebildet. Die obere grüne Kurve zeigt $I_{BT}$, also den Strom, der durch den `VBT`-Pin der HaLOEWEn-Platine fließt und die Hauptspannungsquelle des gesamten Aufbaus darstellt. Die untere rote Kurve zeigt $I_{Pump}$, also den Strom, der über die `VPUMP`-Leitung der JTAG-Schnittstelle fließt. Die schwarzen Bereiche um die Kurven stehen für die Standardabweichung.

Vor Beginn des ersten Abschnitts, also bis zum Zeitpunkt t = 0 s und nach Ende des dritten Abschnitts, also zum Zeitpunkt t = 429 s und mit „Fertig“ beschriftet, kann man starke Wechsel von $I_{BT}$ zwischen ca. 32,0 mA und 40,7 mA mit einer Standardabweichung von ca. ±4 mA erkennen, die durch das Blinken der LED bei der Ausführung der FPGA-Konfiguration D1 verursacht werden. Zu diesem Zeitpunkt fließt über $I_{Pump}$ ein zu vernachlässigender Strom mit einer zu vernachlässigenden Standardabweichung.

Der erste Abschnitt „Start“ von t = 0 s bis t = 51 s entspricht den Schritten 1, 3 und 4 des Konfigurationsvorgangs (siehe Abschnitt 3.8), und beinhaltet die Überprüfung des Typs des angeschlossenen FPGAs, das Auslesen weiterer FPGA-Daten und das Löschen des Flash-Speichers. Während dieses Zeitraums hat $I_{BT}$einen mittleren Wert von ca. 35,4 mA mit geringen Abweichungen nach oben und unten und einer Standardabweichung von ca. ±3,5 mA. Zu Beginn der Phase steigt $I_{Pump}$ sprunghaft an, um danach auf einen Wert von ca. 1,8 mA zu fallen. Der sprunghafte Anstieg wird möglicherweise durch Kapazitäten im FPGA verursacht. Die Standardabweichung von $I_{Pump}$ ist immer noch zu vernachlässigen.

Der zweite Abschnitt „Schreiben“ beginnt bei t = 51 s und dauert bis t = 244 s. Er entspricht Schritt 5 des Konfigurationsvorgangs, also dem Schreiben der Konfigurationsdaten über die JTAG-Schnittstelle in den Konfigurationsspeicher des FPGAs. Dazu wird die Konfigurationsdatei per Funk segmentweise vom PC angefordert. Der Mittelwert von $I_{BT}$ liegt weiterhin bei ca. 35,4 mA, aber die Stärke des Rauschens ist größer als im ersten Abschnitt. Analog dazu beträgt die Standardabweichung ca. ±5 mA. Zu Beginn dieses Abschnitts hat $I_{Pump}$einen weiteren sprunghaften Anstieg mit anschließendem Abfall auf ca. 0,45 mA, um danach eine Art Sägezahnschwingung mit einem Mittelwert von ca. 0,7 mA zu beginnen. Ab diesem Zeitpunkt beträgt die Standardabweichung von $I_{Pump}$ ca. ±1,5 mA.

Der dritte Abschnitt „Verifizieren“ beginnt bei t = 244 s und endet bei t = 429 s. Er entspricht Schritt 6 des Konfigurationsvorgangs, bei dem DirectC die geschriebenen Konfigurationsdaten verifiziert. Dazu wird die Konfigurationsdatei erneut segmentweise vom PC angefordert. $I_{BT}$ ist unverändert. Die Sägezahnschwingung von $I_{Pump}$ tritt nicht mehr auf, stattdessen hat $I_{Pump}$ einen Mittelwert von ca. 0,7 mA mit einigem Rauschen nach oben und unten. Die Standardabweichung beträgt ca. ±1,5 mA.

Mit den aus dem Diagramm abgelesenen Werten kann man den Energiebedarf grob abschätzen. Dabei erhält man $E_{BT} \approx \bar{V}_{BT} \cdot \bar{I}_{BT} \cdot \Delta t = 3{,}0\,\text{V} \cdot 35{,}4\,\text{mA} \cdot 429\,\text{s} \approx 45{,}6\,\text{J}$ und $E_{Pump} \approx \bar{V}_{Pump} \cdot (\bar{I}_{Pump,1} \cdot \Delta t_1 + \bar{I}_{Pump,2} \cdot \Delta t_2) = 3{,}3\,\text{V} \cdot (1{,}8\,\text{mA} \cdot 51\,\text{s} + 0{,}7\,\text{mA} \cdot 378\,\text{s}) \approx 1{,}2\,\text{J}$.

### 5.4.2 Vergleich des Energiebedarfs

In Abbildung 5.19 ist der Energiebedarf für verschiedene Konfigurationsvorgänge mit einer Segmentlänge von 111 Byte und Übertragung per Funk aufgetragen. Jeder der Balken ist der Mittelwert zweier

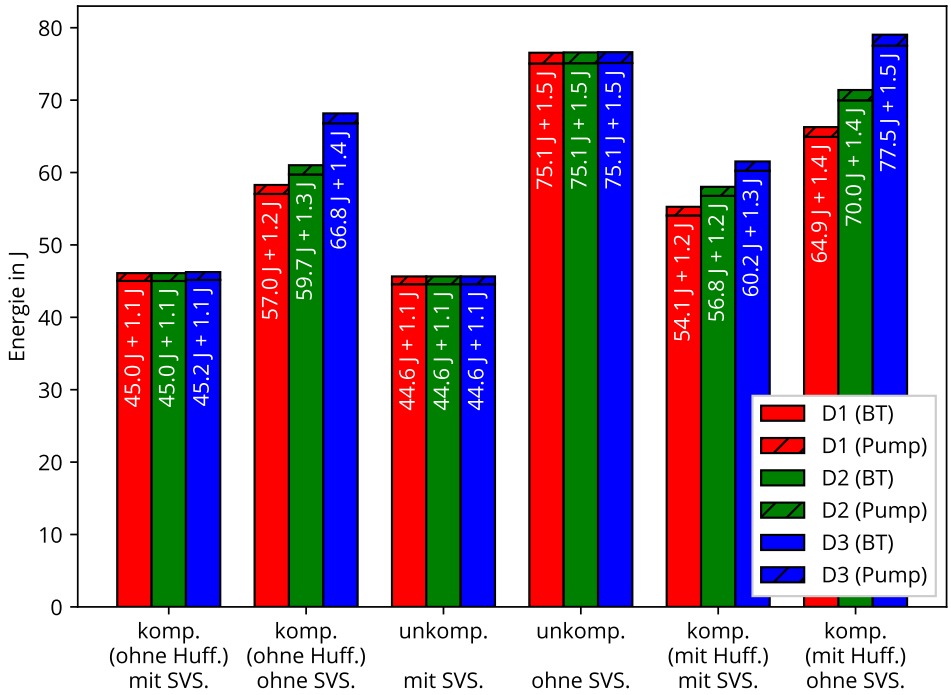


**Abbildung 5.19:** Vergleich des Energiebedarfs eines Konfigurationsvorgangs in Abhängigkeit der Kompressionsmethode, der DAT-Datei und der Aktivierung/Deaktivierung der Segmentvorhersage.

Messungen. Die relative Standardabweichung liegt jeweils unter 8 ‰ und ist deshalb nicht erkennbar. Wie gehabt stehen die Balken innerhalb der Gruppen für die verschiedenen Konfigurationsdateien. Der obere, abgesetzte und schraffierte Teil jedes Balkens steht für den `VPUMP`-Anteil und der untere Teil steht für den `VBT`-Anteil der Gesamtmenge. Die Anordnung der Balkengruppen ist wie gehabt: von links nach rechts ist zunächst die komprimierte Übertragung (ohne Huffman-Kodierung) jeweils mit (Gruppe 1) und ohne Segmentvorhersage (Gruppe 2), dann die unkomprimierte Übertragung jeweils mit (Gruppe 3) und ohne (Gruppe 4) Vorhersage und zuletzt die komprimierte Übertragung (mit Huffman-Kodierung) mit (Gruppe 5) und ohne (Gruppe 6) Vorhersage dargestellt. Die Energiebedarfswerte wurden durch Integrieren der Produkte aus gemessener Spannung und Stromstärke berechnet, d. h. $E = \frac{\sum_i U_i \cdot I_i}{f_s}$ mit der Samplerate $f_s$ und dem für jedes Sample gemessenen Strom $I_i$ und der Spannung $U_i$. Die im vorherigen Abschnitt untersuchte Stromkurve ist eine der beiden Messungen, die die Grundlage für die Werte des ersten Balkens der ersten Gruppe bilden. Die abgeschätzten Werte von $E_{BT} \approx 45{,}6$ J und $E_{Pump} \approx 1{,}2$ J stimmen mit den genaueren Werten von $E_{BT} = 45$ J und $E_{Pump} = 1.1$ J recht gut überein.

Die generelle Struktur des Diagramms entspricht denen der Konfigurationszeit bei kabelloser Übertragung, was bei Betrachtung des beispielhaften Verlaufs des Stromflusses und der daraus erfolgreichen Abschätzung des Energiebedarfs nicht verwunderlich ist, da der Energiebedarf weitestgehend proportional zur Konfigurationszeit ist. Betrachtet man die ersten vier Balkengruppen, braucht der Konfigurationsvorgang mit unkomprimierte Übertragung ohne Vorhersage (Gruppe 4) unabhängig von der gewählten Konfigurationsdatei mit 75,1 J über `VBT` und 1,5 J über `VPUMP` die meiste Energie. Die unkomprimierte Übertragung mit Vorhersage (Gruppe 3) benötigt mit 44,6 J und 1,1 J die geringste Energie. Die Werte für die komprimierte Übertragung (ohne Huffman-Kodierung) mit Vorhersage benötigt mit 45,0 J und 1,1 J für D1 und D2 bzw. 45,2 J und 1,1 J für D3 etwas mehr Energie. Wird keine Vorhersage verwendet und komprimiert (ohne Huffman-Kodierung) übertragen, ist der Energiebedarf von der gewählten Konfigurationsdatei abhängig und benötigt beispielsweise 57,0 J und 1,2 J für D1 und bis zu 66,8 J

und 1,4 J für D3. Wie bei Betrachtung der Konfigurationszeiten bringt die Verwendung der Huffman-Kodierung keine Verbesserung. Die ermittelten Werte in Gruppe 5 und 6 liegen alle deutlich über den entsprechenden Werten ohne Huffman-Kodierung in Gruppe 1 und 2.

**Interpretation:**

Auch aus energetischer Sicht ist die Verwendung der Huffman-Kodierung nicht empfehlenswert. Weitaus besser ist die Verwendung von Segmentvorhersage, mit der der Energiebedarf um ca. 40 % verringert werden kann. Ohne die Verwendung der Segmentvorhersage kann stattdessen durch Kompression je nach verwendeten Konfigurationsdaten der Energiebedarf um ca. 11 % bis 24 % gesenkt werden.

# 6 Diskussion

In diesem Abschnitt werden zunächst verwandte Arbeiten vorgestellt. Danach werden in Abschnitt 6.2 die Ergebnisse dieser Arbeit zusammengefasst und, soweit möglich, mit den Ergebnissen der anderen Arbeiten verglichen. Zuletzt folgt ein Ausblick auf weitere an diese Arbeit anschließende Forschungsansätze.

## 6.1 Verwandte Arbeiten

Adly et al. verwenden einen PSoC (CY8C29466) der Firma Cypress auf einer Erweiterungsplatine eines modular aufgebauten Sensorknotens, um eine flexible Anschlussmöglichkeit für analoge Sensoren bereitzustellen. Mit einem zwischengeschalteten CPLD (EPM2210F324, einem Bauteil der Reihe Max II, hergestellt von der Firma Altera), kann der PSoC über Funk rekonfiguriert werden. Am CPLD ist ein Flash-Speicher zum Speichern von mehreren unterschiedlichen Konfigurationen des PSoC angeschlossen. Es sind zwei unterschiedliche Modi implementiert. Wenn der Flash-Speicher leer ist, werden die Konfigurationsdaten per Funk übertragen und auf dem Flash-Speicher abgelegt. Erst im Anschluss an die erfolgreiche Übertragung wird die Rekonfiguration durchgeführt. Alternativ dazu können bereits abgespeicherte Konfigurationsdaten benutzt werden. Die verwendeten PSoC-Konfigurationsdaten sind jeweils 32 kB groß, ihre Übertragung dauert ca. 1 s und ein Konfigurationsvorgang dauert 15 s [34].

In einem anderen Artikel stellen Adly et al. eine Methode zur Wahl der Paketlänge, die eine bestmögliche Übertragungsqualität erlaubt, für die kabellose Rekonfiguration in drahtlosen Sensornetzwerken vor. Vor der Übertragung der Konfigurationsdaten wird die verwendete Länge durch Versenden eines oder mehrerer Testpakete zur Bestimmung der *Received Signal Strength Indicator* (*RSSI*), eines Indikators für die Empfangsfeldstärke, ermittelt [35].

Wee et al. schlagen ein modifiziertes Kompressionsverfahren vor, das für die Aktualisierung von Firmware von Mobiltelefonen genutzt werden kann. Ihr Ansatz sieht vor, dass bei der Aktualisierung die aktuelle Firmware, die zur Beschleunigung des Startvorgangs komprimiert im verwendeten NAND-Flash-Speicher abgelegt ist, komplett dekomprimiert wird. Danach werden diese alten Daten mit den empfangenen Aktualisierungsdaten modifiziert und zuletzt im komprimierten Zustand auf den NAND-Flash-Speicher zurück geschrieben. Dieses Vorgehen ist notwendig, da mit einer Art der Delta-Kodierung gearbeitet wird und diese für unkomprimierte Daten effektiver ist. Zur Kompression der übertragenen Daten wird ein modifizierter Lempel-Ziv-Algorithmus verwendet [36].

Biedermann et al. beschreiben einen Sensorknoten, der auf einem Xilinx Spartan-6 FPGA basiert, der ebenfalls Möglichkeiten zur Rekonfiguration hat. Die Speicherung der Konfigurationsdaten kann entweder auf den einzelnen Knoten oder auf einem dezidierten Server erfolgen. Die Übertragung der Daten erfolgt per HTTP über TCP mit einer maximalen Übertragungsrate von 1 MB/s. Die Knoten können sich selbst partiell rekonfigurieren. Dazu wird die Fähigkeit der FPGAs der Spartan-Reihe zur dynamischen Rekonfiguration genutzt. Eine komplette Rekonfiguration der FPGAs kann durch Verwendung eines zweiten Sensorknotens, der mit dem zu rekonfigurierenden Knoten per JTAG verbunden ist, durchgeführt werden [37].

Krasteva et al. verwenden bei ihrem Sensorknoten namens Cookie ein separates ZigBee-Modul (ETRX2 von Telegesis), mit dem der verwendete ADuC841 Mikrocontroller der Firma Analog Devices neu programmiert werden kann. Mit dem Mikrocontroller kann per JTAG eine partielle Rekonfiguration des Xilinx XC3S200 Spartan-3 FPGAs vorgenommen werden. Durch die Möglichkeit, den Mikrocontroller kabellos neu zu programmieren, kann die für die Rekonfiguration des FPGA benötigte Funktionalität bei Bedarf übertragen werden. Somit stehen im normalen Betrieb mehr Ressourcen des Mikrocontrollers zur Verfügung. Im Experiment wurden zwei verschiedene Übertragungsformate verwendet, die sich durch die Anzahl der verwendeten Bytes unterscheiden. Die beiden Formate wurden jeweils bei Übertragung

per Kabel und per Funk untersucht. Eine kabelgebundene partielle Rekonfiguration benötigt ca. 87 s bei Verwendung des 8-Byte-Formats bzw. ca. 67 s bei Verwendung des 16-Byte-Formats. Bei kabelloser Übertragung werden ca. 219 s bzw. ca. 144 s für die partielle Rekonfiguration benötigt. Eine vollständige Rekonfiguration ist wegen der Größe der Konfigurationsdateien von 131 kB nicht vorgesehen [38, 39]. Peter et al. haben die drahtlose partielle Rekonfiguration des Cookie-Sensorknotens am Beispiel von Elliptischer-Kurven-Kryptografie (*Elliptic curve cryptography*, *ECC*) untersucht [40].

Yamaguchi et al. haben einen Sensorknoten entwickelt, der aus einem MSP430 Mikrocontroller, einem 920 MHz Funkmodul der Firma Markhor und einem Lattice iCE40LP1K FPGA besteht. Das FPGA kann per Funk innerhalb von 200 s rekonfiguriert werden [41].

Moiş et al. verwenden ein Tag4M, ein WiFi-RFID-Modul basierend auf einem 32-Bit SPARC V8 Mikroprozessor, zur Rekonfiguration eines Xilinx Spartan-3 FPGAs [42].

Rubert et al. benutzen ein separates Bluetooth-Modul (Adeunis ARF7678AA) zur drahtlosen Rekonfiguration eines XESS XuLA FPGA [43].

## 6.2 Zusammenfassung

In dieser Arbeit wurde die Rekonfiguration eines FPGAs per Funk durch einen Mikrocontroller untersucht. Die verwendete Hardware entspricht der mittlerweile veralteten Version 3 der HaLOEWEn-Platine. Die Rekonfiguration wird über die JTAG-Schnittstelle des FPGAs durchgeführt. Der Mikrocontroller verwendet eine modifizierte Version des vom Hersteller des FPGAs zur Verfügung gestellten Projektes DirectC. Da die Konfiguration, auch wegen der geringen Zahl an verfügbaren Pins des TI CC2531 Mikrocontrollers ohne zusätzliche Bauteile, d. h. auch ohne zusätzlichen Speicher, durchgeführt wird, kann die im Vergleich zum verfügbaren Speicher sehr große Konfigurationsdatei nur partiell auf dem Mikrocontroller gespeichert werden. Dies hat den Nachteil, dass Daten die zu verschiedenen Zeiten des Konfigurationsvorgangs benötigt werden, auch mehrfach übertragen werden müssen. Eine segmentweise durchgeführte vollständige Übertragung der Konfigurationsdatei konnte durch die Verlagerung der Prüfsummenberechnung auf den PC vermieden werden. Durch das Einsparen der Übertragung und die größere Rechenleistung des PCs kann damit die für den Konfigurationsvorgang benötigte Zeit um ca. 90 s gesenkt werden.

Um die übertragene Datenmenge weiter zu verringern und damit die für die Übertragung verwendete Zeit zu senken, wurden verschiedene Kompressionsverfahren untersucht. Für jedes übertragene Segment wird die bestmögliche Kompression vom PC automatisch gewählt. Im Experiment hat sich herausgestellt, dass dieses Verfahren am besten funktioniert, wenn der PC zwischen unkomprimierter Übertragung und zwei verschiedener Methoden der Lauflängenkodierung wählen kann. Die zusätzliche Verwendung der ebenfalls implementierten Huffman-Kodierung führt bei der begrenzten Leistungsfähigkeit des verwendeten Mikrocontrollers zu keiner Verbesserung hinsichtlich der Konfigurationszeit bzw. des Energiebedarfs, da die Dekodierung zu rechenaufwändig ist. Deaktiviert man die Huffman-Kodierung, kann zudem der Programmspeicherbedarf (`CODE`-Segment) auf dem Mikrocontroller um ca. 1653 Byte reduziert werden.

Die größte Einsparung hinsichtlich der Konfigurationszeit bzw. des Energiebedarfs (jeweils ca. 40 %) erhält man durch die entwickelte Strategie der Segmentvorhersage, bei der Aufzeichnungen aus vorherigen Konfigurationsvorgängen verwendet werden, um die Übertragung des nächsten benötigten Segments weitestgehend im Hintergrund durchzuführen, während DirectC mit den Daten des aktuellen Segments arbeitet.

Bei den Messungen des Energiebedarfs hat sich herausgestellt, dass der Stromfluss über die `VPUMP`-Leitung der JTAG-Schnittstelle deutlich geringer als erwartet ist. Bei unkomprimierter Übertragung mit Segmentvorhersage beträgt der Energiebedarf über `VPUMP` 1,1 J, während über `VBT`, also die Spannungsversorgung der HaLOEWEn-Platine, ein Strom fließt, der einen Energiebedarf von 44,6 J verursacht. Damit liegt der `VPUMP`-Anteil lediglich bei ca. 2,5 % des gesamten Bedarfs.

Ein Vergleich der Messergebnisse mit denen im vorherigen Abschnitt genannten verwandten Arbeiten ist schwierig. Einerseits sind oft benötigte Daten hinsichtlich der Größe der Konfigurationsdateien, der Konfigurationszeit und des Energiebedarfs nicht angegeben, andererseits hat deren jeweils verwendete Hardware eine größtenteils deutlich höhere Rechenleistung und maximale Übertragungsgeschwindigkeit. Zudem sind die untersuchten Konfigurationsdateien gleichzeitig wegen Verwendung partieller Rekonfiguration oder des Einsatzes kleinerer FPGAs/PSoCs um Größenordnungen kleiner. Einzig der Vergleich der Konfigurationszeit mit Krasteva et al. ist möglich, aber wegen der unterschiedlichen Konfigurationsmodi nur bedingt aussagekräftig: In ihrem Experiment benötigen sie im besten Fall für eine drahtlose partielle Rekonfiguration ca. 144 s, während in dieser Arbeit mindestens (bei unkomprimierter Übertragung mit Segmentvorhersage und Segmentlänge von 255 Byte) ca. 417 s erforderlich sind, da hier eine drahtlose vollständige Rekonfiguration durchgeführt wird.

## 6.3 Ausblick

Grundsätzlich wäre eine Untersuchung mit einer aktuellen Version der HaLOEWEn-Platine interessant. Ab der neueren Version vier der Platine ist statt des TI CC2531 ein leistungsfähigerer ATmega256RFR2 Mikrocontroller der Firma Atmel verbaut. Dieser verfügt zudem über deutlich mehr nutzbare Pins als der CC2531.

Da der Konfigurationsvorgang mit Segmentvorhersage aus zeitlicher und energetischer Sicht am sinnvollsten ist, ist der implementierte Modus, im Fall eines Vorhersagefehlers die Vorhersage ohne Korrekturversuche zu deaktivieren, ungünstig. Sowohl bei unveränderter Hardware als auch bei Nutzung der neueren Version der HaLOEWEn-Platine, empfiehlt sich die Untersuchung alternativer Strategien. Beispielsweise ist es denkbar, je nach Anzahl der bisher erfolgreich vorhergesagten Pakete, eine mögliche Anzahl an Wiederholungen für das gleiche vorhergesagte Paket zu erlauben. Dieser Ansatz hat zum einen den Vorteil, dass bei falschen Vorhersagedaten zu Beginn des Konfigurationsvorgangs nicht zu viel Zeit bzw. Energie durch unnötige Wiederholungen verschwendet wird, aber bei korrekten Vorhersagedaten im Fall eines Übertragungsfehlers versucht wird, den schnelleren bzw. sparsameren Modus beizubehalten.

Ein großer Nachteil der aktuellen Implementierung ist, dass während des ganzen Konfigurationsvorgangs der Mikrocontroller entweder auf eingehende Funk-Pakete wartet oder er selbst Daten per Funk sendet. Somit wird der Stromsparmodus des Mikrocontrollers nicht genutzt. Prinzipiell könnte der Funk-Empfang immer ausgeschaltet werden, wenn nicht auf Segmente der Konfigurationsdatei gewartet wird. Dies ist eine einfache Möglichkeit, die Leistungsaufnahme des Mikrocontrollers zu senken, die aus Zeitgründen nicht untersucht wurde.

Bei Verwendung eines zusätzlichen Speichers, der groß genug ist, eine Konfigurationsdatei vollständig zu speichern, gibt es eine große Zahl vielversprechender Ansätze, um den Konfigurationsvorgang zu beschleunigen. Wird ein volatiles Speichermedium, also z. B. SRAM verwendet, muss die Konfigurationsdatei höchstens einmal vollständig, statt wie bisher ca. zweimal partiell, übertragen werden. Zudem gibt es die Möglichkeit bei der Konfiguration mehrerer Sensorknoten die vollständig übertragenen gleichen Daten für mehrere Knoten zu verwenden und individuelle Anpassungen für die einzelnen Knoten vor Beginn des Konfigurationsprozesses beispielsweise per Delta-Kodierung zu übertragen. Mit dieser Methode könnte die übertragene Gesamtdatenmenge stark gesenkt werden. Falls sich die verwendeten Konfigurationsdaten nicht stark von den zuvor verwendeten unterscheiden, ist es auch denkbar, die auf dem FPGA gespeicherte Konfiguration auszulesen, im SRAM zu speichern und die neuen Konfigurationsdaten Delta-kodiert zu übertragen. Auch auf diese Weise könnte prinzipiell die übertragene Datenmenge gesenkt werden. Natürlich müsste überprüft werden, ob das Auslesen der Konfiguration des FPGAs nicht mehr Zeit als die Übertragung der Daten per Funk benötigt. Auch wenn dies nicht der Fall ist, könnte trotzdem weniger Energie benötigt werden, da während des Auslesens keine Daten per Funk empfangen werden müssen und durch Abschalten des Funk-Empfangs der Energiebedarf des Mikrocontrollers stark gesenkt werden kann. Wird statt eines volatilen Speichermediums ein nicht-volatiles Medium wie z. B.

ein Flash-Speicher verwendet, könnte der Auslesevorgang des FPGAs eingespart und stattdessen direkt die auf dem Speicher abgelegten Daten für die Delta-Kodierung verwendet werden.

Alle für den Konfigurationsvorgang verwenden Funk-Nachrichten werden unverschlüsselt übertragen und verwenden als einzige Absicherung die Prüfsumme des Funkpaketes. Aus diesem Grund ist es sehr einfach, falsche Konfigurationsdaten während der Übertragung der Daten einzuschleusen. Dies stellt ein großes Sicherheitsrisiko im Praxiseinsatz dar. Deshalb ist die vorliegende Implementierung nur für Untersuchungen im Labor geeignet. Alle zusätzlichen Maßnahmen, die ein Mindestmaß an notwendiger Sicherheit gewährleisten, würden eine Steigerung der Konfigurationszeit und des Energiebedarfs verursachen.

# Anhang

## Nachrichtenformate zur Kommunikation zwischen PC und Mikrocontroller

### Nachrichten vom Mikrocontroller zum PC

**Statusmeldung in Textform:**

| Beschreibung: | Marke | ASCII-Daten | Marke |
|---|---|---|---|
| Wert: | \x01 | ··· | \x02 |
| Länge in Byte: | 1 | var. | 1 |
| Beispiel: | \x01 | Test | \x02 |

**Rückgabewert:**

| Beschreibung: | Marke | Rückgabewert |
|---|---|---|
| Wert: | \x03 | ··· |
| Länge in Byte: | 1 | 1 |
| Beispiel: | \x03 | \x00 |

**Anfrage eines Segments:**

| Beschreibung: | Marke | Adresse | Länge |
|---|---|---|---|
| Wert: | \x04 | ··· | ··· |
| Länge in Byte: | 1 | 3 | 1 |
| Beispiel: | \x03 | \xAB\xCD\xEF | \xF4 |

Die Adresse ist im Big-Endian-Format angegeben, d. h. 11259375=0xABCDEF wird als 0xAB, 0xCD, 0xEF angegeben.

**Anfrage eines vorhergesagten Segments:**

| Beschreibung: | Marke |
|---|---|
| Wert: | \x14 |
| Länge in Byte: | 1 |
| Beispiel: | \x14 |

**Statusmeldung (wörterbuchkomprimiert):**

| Beschreibung: | Marke | Code |
|---|---|---|
| Wert: | \x10 | ··· |
| Länge in Byte: | 1 | 1 |
| Beispiel: | \x10 | \01 |

**Statusmeldung (Zahl, Anzeige im Hexadezimal-Format):**

| Beschreibung: | Marke | Zahl |
|---|---|---|
| Wert: | \x11 | ··· |
| Länge in Byte: | 1 | 4 |
| Beispiel: | \x11 | \xAB\xCD\xEF\x01 |

Die Zahl ist im Big-Endian-Format kodiert.

**Statusmeldung (Zahl, Anzeige im Dezimal-Format):**

| Beschreibung: | Marke | Zahl |
|---|---|---|
| Wert: | \x12 | ··· |
| Länge in Byte: | 1 | 4 |
| Beispiel: | \x12 | \xAB\xCD\xEF\x01 |

Die Zahl ist im Big-Endian-Format kodiert.

## Nachrichten vom PC zum Mikrocontroller

**Segment (unkomprimiert):**

| Beschreibung: | Marke | Länge | Daten |
|---|---|---|---|
| Wert: | \x05 | ··· | ··· |
| Länge in Byte: | 1 | 1 | var. |
| Beispiel: | \x05 | \x05 | \x01\x23\x45\x67\x89 |

**Segment (RLE-codiert):**

| Beschreibung: | Marke | Länge | RLE-codierte Daten |
|---|---|---|---|
| Wert: | \x06 | ··· | ··· |
| Länge in Byte: | 1 | 1 | var. |
| Beispiel: | \x06 | \x06 | \x01\x23\x45\x67\x89\xAB |

**Segment (RLEM-codiert):**

| Beschreibung: | Marke | Länge | RLEM-codierte Daten |
|---|---|---|---|
| Wert: | \x07 | ··· | ··· |
| Länge in Byte: | 1 | 1 | var. |
| Beispiel: | \x07 | \x06 | \x01\x23\x45\x67\x89\xAB |

**Segment (Huffman-codiert):**

| Beschreibung: | Marke | Länge | Bit-Anzahl | Huffman-codierte Daten |
|---|---|---|---|---|
| Wert: | \x08 | ··· | ··· | ··· |
| Länge in Byte: | 1 | 1 | 1 | var. |
| Beispiel: | \x08 | \x07 | \x02 | \x01\x23\x45\x67\x89\xAB |

Mit „Bit-Anzahl“ ist die Anzahl an Bits, die das erste Byte der kodierten Nachrichten enthält gemeint. Das dafür verwendete Byte wird bei der Berechnung der Länge mitgerechnet.

**Segment (unkomprimiert mit Vorhersage):**

| Beschreibung: | Marke | Länge | Adresse | Daten |
|---|---|---|---|---|
| Wert: | \x15 | ··· | ··· | ··· |
| Länge in Byte: | 1 | 1 | 3 | var. |
| Beispiel: | \x15 | \x05 | \xAB\xCD\xEF | \x01\x23\x45\x67\x89 |

**Segment (RLE-codiert):**

| Beschreibung: | Marke | Länge | Adresse | RLE-codierte Daten |
|---|---|---|---|---|
| Wert: | \x16 | ··· | ··· | ··· |
| Länge in Byte: | 1 | 1 | 3 | var. |
| Beispiel: | \x16 | \x06 | \xAB\xCD\xEF | \x01\x23\x45\x67\x89\xAB |

**Segment (RLEM-codiert):**

| Beschreibung: | Marke | Länge | Adresse | RLEM-codierte Daten |
|---|---|---|---|---|
| Wert: | \x17 | ··· | ··· | ··· |
| Länge in Byte: | 1 | 1 | 3 | var. |
| Beispiel: | \x17 | \x06 | \xAB\xCD\xEF | \x01\x23\x45\x67\x89\xAB |

**Segment (Huffman-codiert):**

| Beschreibung: | Marke | Länge | Adresse | Bit-Anzahl | Huffman-codierte Daten |
|---|---|---|---|---|---|
| Wert: | \x18 | ··· | ··· | ··· | ··· |
| Länge in Byte: | 1 | 1 | 3 | 1 | var. |
| Beispiel: | \x18 | \x07 | \xAB\xCD\xEF | \x02 | \x01\x23\x45\x67\x89\xAB |

Mit „Bit-Anzahl“ ist die Anzahl an Bits, die das erste Byte der kodierten Nachrichten enthält gemeint. Das dafür verwendete Byte wird bei der Berechnung der Länge mitgerechnet.

**Start des Konfigurationsvorgangs:**

| Beschreibung: | Marke |
|---|---|
| Wert: | \x60\60 |
| Länge in Byte: | 2 |
| Beispiel: | \x60\60 |

**Start des Konfigurationsvorgangs (erzwingt Deaktivierung der Vorhersage):**

| Beschreibung: | Marke |
|---|---|
| Wert: | \x60\61 |
| Länge in Byte: | 2 |
| Beispiel: | \x60\61 |

## Tabellenverzeichnis

## Abbildungsverzeichnis

## Literaturverzeichnis


[1] Wei Liu, Xiaotian Fei, Tao Tang, Pengjun Wang, Hong Luo, Beixing Deng, and Huazhong Yang. Application specific sensor node architecture optimization —experiences from field deployments. In *17th Asia and South Pacific Design Automation Conference*, pages 389–394, Jan 2012.

[2] W. Z. Song, R. Huang, M. Xu, B. Shirazi, and R. LaHusen. Design and deployment of sensor network for real-time high-fidelity volcano monitoring. *IEEE Transactions on Parallel and Distributed Systems*, 21(11):1658–1674, Nov 2010.

[3] J. Burrell, T. Brooke, and R. Beckwith. Vineyard computing: sensor networks in agricultural production. *IEEE Pervasive Computing*, 3(1):38–45, Jan 2004.

[4] Octav Chipara, Chenyang Lu, Thomas C. Bailey, and Gruia-Catalin Roman. Reliable clinical monitoring using wireless sensor networks: Experiences in a step-down hospital unit. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 155–168, New York, NY, USA, 2010. ACM.

[5] D. Boyle, M. Magno, B. O'Flynn, D. Brunelli, E. Popovici, and L. Benini. Towards persistent structural health monitoring through sustainable wireless sensor networks. In *2011 Seventh International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pages 323–328, Dec 2011.

[6] D. Abruzzese, M. Angelaccio, R. Giuliano, L. Miccoli, and A. Vari. Monitoring and vibration risk assessment in cultural heritage via wireless sensors network. In *2009 2nd Conference on Human System Interactions*, pages 568–573, May 2009.

[7] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *2007 6th International Symposium on Information Processing in Sensor Networks*, pages 254–263, April 2007.

[8] V. Potdar, A. Sharif, and E. Chang. Wireless sensor networks: A survey. In *2009 International Conference on Advanced Information Networking and Applications Workshops*, pages 636–641, May 2009.

[9] M. Z. A. Bhuiyan, J. Cao, and G. Wang. Deploying wireless sensor networks with fault tolerance for structural health monitoring. In *2012 IEEE 8th International Conference on Distributed Computing in Sensor Systems*, pages 194–202, May 2012.

[10] M. Z. A. Bhuiyan, J. Cao, G. Wang, and X. Liu. Energy-efficient and fault-tolerant structural health monitoring in wireless sensor networks. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 301–310, Oct 2012.

[11] J. Wu and G. Zhou. A new ultra-low power wireless sensor network with integrated energy harvesting, data sensing, and wireless communication. In *2011 IEEE International Conference on Communications (ICC)*, pages 1–5, June 2011.

[12] T. Galchev, J. McCullagh, R. L. Peterson, and K. Najafi. Harvesting traffic-induced bridge vibrations. In *2011 16th International Solid-State Sensors, Actuators and Microsystems Conference*, pages 1661–1664, June 2011.

[13] IEEE Computer Society. IEEE standard for Low-Rate Wireless Networks. *IEEE Std 802.15.4-2015*.

[14] D. Dondi, A. Di Pompeo, C. Tenti, and T. Simunic Rosing. Shimmer: A wireless harvesting embedded system for active ultrasonic structural health monitoring. In *2010 IEEE Sensors*, pages 2325–2328, Nov 2010.

[15] A. Engel, B. Liebig, and A. Koch. HaLOEWEn: A heterogeneous reconfigurable sensor node for distributed structural health monitoring. In *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*, pages 1–2, Oct 2012.

[16] F. Philipp, F. A. Samman, and M. Glesner. Design of an autonomous platform for distributed sensing-actuating systems. In *2011 22nd IEEE International Symposium on Rapid System Prototyping*, pages 85–90, May 2011.

[17] Texas Instruments Incorporated. *A True System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee Applications*, February 2011. SWRS081B.

[18] Intel Corporation. *MCS 51 Microcontroller Family User's Manual*, February 1994.

[19] Texas Instruments Incorporated. *CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee Applications*, February 2011. SWRU191F.

[20] Texas Instruments Incorporated. *CC2530 FAQ*, 2009. SWRA280A.

[21] Microsemi corporation: Acquisitions. `https://www.microsemi.com/company/acquisitions`. Abgerufen: 2017-04-06.

[22] Microsemi Corporation. *IGLOO Low Power Flash FPGAs*, revision 27 edition, May 2016. DS0095.

[23] Microsemi Corporation. *Performing Microprocessor Programming for Microsemi ProASICPLUS Devices*, October 2015. Application Note AC208.

[24] Microsemi Corporation. *DirectC v3.2 User's Guide*, 2015.

[25] C. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27, 1948.

[26] David Huffman. A method for the construction of minimum redundancy codes. 40(9):1098–1101, 1952.

[27] pySerial: Python serial port extension. `https://pypi.python.org/pypi/pyserial/`. Abgerufen: 2017-05-30.

[28] Microsemi Corporation. *FlashPro5 Device Programmer Quickstart Card*, 2014.

[29] Microsemi Corporation. *FlashPro Express for Software v11.6 User's Guide*, 2015.

[30] Coan: The C Preprocessor Chainsaw. `http://coan2.sourceforge.net/`. Abgerufen: 2017-05-26.

[31] Texas Instruments Incorporated. *SmartRF Packet Sniffer User's Manual*, 2009. SWRU187G.

[32] *SDCC Compiler User Guide 3.5.0*, 2015. `http://sdcc.sourceforge.net/`.

[33] Hitex Development Tools GmbH. *PowerScale User Manual*, 006 edition, March 2011.

[34] I Adly, HF Ragai, A El-Hennawy, and KA Shehata. Over-the-air programming of PSoC sensor interface in wireless sensor networks. In *MELECON 2010-2010 15th IEEE Mediterranean Electrotechnical Conference*, pages 997–1002. IEEE, April 2010.

[35] I. Adly, H. F. Ragai, A. E. Elhennawy, and K. A. Shehata. Adaptive packet sizing for OTAP of PSoC based interface board in WSN. In *2010 International Conference on Microelectronics*, pages 148–151, Dec 2010.

[36] Y. Wee and T. Kim. A new code compression method for FOTA. *IEEE Transactions on Consumer Electronics*, 56(4):2350–2354, November 2010.

[37] A. Biedermann, B. Dreyer, and S. Huss. A generic, scalable reconfiguration infrastructure for sensor networks functionality adaption. In *2013 IEEE International SOC Conference*, pages 301–306, Sept 2013.

[38] Y. E. Krasteva, J. Portilla, J. M. Carnicer, E. de la Torre, and T. Riesgo. Remote HW-SW reconfigurable wireless sensor nodes. In *2008 34th Annual Conference of IEEE Industrial Electronics*, pages 2483–2488, Nov 2008.

[39] Y. E. Krasteva, J. Portilla, E. de la Torre, and T. Riesgo. Embedded runtime reconfigurable nodes for wireless sensor networks applications. *IEEE Sensors Journal*, 11(9):1800–1810, Sept 2011.

[40] S. Peter, O. Stecklina, J. Portilla, E. de la Torre, P. Langendoerfer, and T. Riesgo. Reconfiguring crypto hardware accelerators on wireless sensor nodes. In *2009 6th IEEE Annual Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks Workshops*, pages 1–3, June 2009.

[41] S. Yamaguchi, T. Miyazaki, J. Kitamichi, S. Guo, T. Tsukahara, and T. Hayashi. Programmable wireless sensor node featuring low-power FPGA and microcontroller. In *2013 International Joint Conference on Awareness Science and Technology Ubi-Media Computing (iCAST 2013 UMEDIA 2013)*, pages 596–601, Nov 2013.

[42] G. D. Mois, M. Hulea, S. Folea, and L. Miclea. Self-healing capabilities through wireless reconfiguration of FPGAs. In *2011 9th East-West Design Test Symposium (EWDTS)*, pages 22–27, Sept 2011.

[43] P. Ruberg, A. Guitar, and P. Ellervee. Flexible controller for educational robot kit. In *2015 IEEE International Conference on Microelectronics Systems Education (MSE)*, pages 17–20, May 2015.