Universität Stuttgart

Fakultät Informatik, Elektrotechnik
und Informationstechnik

Diplomarbeit Nr. 2274

# Dynamischer Austausch von Komponenten in TinyOS

Matthias Gauger

**Studiengang:** Informatik

**Prüfer:** Prof. Dr. Kurt Rothermel
**Betreuer:** Dr. Pedro José Marrón
Dipl.-Inf. Andreas Lachenmann
Dipl.-Inf. Daniel Minder

**begonnen am:** 28.10.2004
**beendet am:** 29.04.2005

**CR-Klassifikation:** C.3, D.3.4, D.4.9

Institut für Parallele und
Verteilte Systeme
Abteilung Verteilte Systeme
Universitätsstr. 38
D–70569 Stuttgart

**Zusammenfassung**

Sensornetze und Sensornetzanwendungen sind ein sehr aktives Forschungsgebiet, das von zahlreichen Forschergruppen in verschiedenen Arbeitsbereichen vorangetrieben wird. Mit der fortschreitenden Entwicklung der verfügbaren Hard- und Softwarelösungen wird eine wachsende Anzahl unterschiedlichster Anwendungen möglich. Die zunehmende Komplexität der Anwendungen und gleichzeitig steigende Erwartungen an die Laufzeit der Systeme sorgen dabei für einen steigenden Bedarf an Lösungen, die dynamische Aktualisierungen der Anwendungsprogramme während des Betriebs ermöglichen.

Ansätze zur Aktualisierung des Anwendungscodes müssen schonend mit den beschränkten Ressourcen der Sensorknoten umgehen. Gleichzeitig sollten die Update-Mechanismen eine möglichst einfache und flexible Umgestaltung der Funktionalität der Sensorknoten ermöglichen.

In dieser Arbeit wird mit `FlexCup` („**FLEX**ible **C**ode **UP**dates“) ein System vorgestellt, das als Teil des `TinyCubus`-Projektes den dynamischen Austausch von Komponenten in Sensornetzen ermöglicht, die auf dem TinyOS-Betriebssystem aufbauen. Anstatt im Falle von Aktualisierungen den vollständigen Anwendungscode neu zu übertragen, zerteilt `FlexCup` die Anwendungen in mehrere Komponenten und überträgt nur den Programmcode geänderter Komponenten. Die Integration geänderter Komponenten in die Anwendungen geschieht mit Hilfe von Metadaten, die auf den Sensorknoten gespeichert und zusammen mit neuen Komponenten übertragen werden.

Die Funktionalität von `FlexCup` wird von mehreren Einzelanwendungen realisiert, die in dieser Arbeit ausführlich vorgestellt werden. Der `FlexCup`-Analyzer analysiert die Komponenten der Anwendungen und generiert die benötigten Metadaten. Der `FlexCup`-Linker ist für die dynamische Integration der Komponenten in den Programmcode der Sensorknoten zuständig. Die Aufgabe des `FlexCup`-Bootloaders ist anschließend die Installation des geänderten Anwendungscodes.

Ein Vergleich von `FlexCup` mit zwei verwandten Ansätzen aus der Literatur zeigt, dass `FlexCup` identische Codeaktualisierungen bis zu achtmal schneller erledigt und dabei in manchen Fällen mit nur einem Achtel des Energieverbrauchs der anderen Ansätze auskommt.

Am Ende dieser Ausarbeitung werden die Ergebnisse der Arbeit zusammengefasst, und es erfolgt ein Ausblick auf mögliche Erweiterungen und zukünftige Entwicklungsmöglichkeiten des `FlexCup`-Systems.

# Inhaltsverzeichnis


**1 Einleitung** **1**
1.1 Überblick . . . 1
1.2 Problemstellung . . . 2
1.3 Zielsetzung . . . 3
1.4 Aufbau dieser Ausarbeitung . . . 4

**2 Grundlagen** **5**
2.1 Die Mica2-Sensorplattform . . . 5
2.1.1 Prozessor . . . 6
2.1.2 Externer Flashspeicher . . . 8
2.1.3 Funkschnittstelle . . . 9
2.1.4 Sensorbausteine . . . 10
2.2 TinyOS . . . 10
2.3 nesC . . . 11
2.3.1 nesC-Compiler . . . 12
2.3.2 Sprachaufbau von nesC . . . 12
2.3.3 Binäre Komponenten in nesC 1.2 . . . 16
2.4 Kompilieren für AVR-Prozessoren . . . 18
2.4.1 Speicherlayout . . . 18
2.4.2 Objektdateien . . . 22
2.4.3 Linker . . . 26
2.4.4 Übertragen und Installieren des Programms . . . 28

**3 Verwandte Arbeiten** **31**
3.1 Komplettaustausch . . . 31
3.1.1 Xnp . . . 31
3.1.2 Deluge . . . 32
3.2 Differenzansätze . . . 34
3.2.1 Algorithmus von Reijers und Langendoen . . . 34
3.2.2 Implementierung von Ye, Yamamoto und Stathopolous . . . 35
3.2.3 Ansatz von Jeong und Culler . . . 36
3.2.4 Inkrementelles Linken . . . 36
3.3 „Virtual Machine“-Ansätze . . . 38
3.3.1 Maté . . . 38
3.3.2 SensorWare . . . 39

3.4 Middleware- und Betriebssystemansätze . . . 40
3.4.1 Impala . . . 40
3.4.2 Contiki . . . 41

**4 TinyCubus** **43**
4.1 Konzept . . . 43
4.1.1 Tiny Data Management Framework . . . 43
4.1.2 Tiny Cross-Layer Framework . . . 44
4.1.3 Tiny Configuration Engine . . . 45
4.2 Diplomarbeit von Robert Sauter . . . 46
4.3 Anforderungen an diese Diplomarbeit . . . 47

**5 Entwurf** **49**
5.1 Vorgehen und Arbeitsschritte . . . 49
5.2 Annahmen . . . 51
5.3 Aufbau der Daten im Speicher der Sensorknoten . . . 53
5.3.1 Entwurfsprinzipien . . . 53
5.3.2 Bezeichner . . . 54
5.3.3 Struktur der Metadaten auf dem Sensorknoten . . . 54
5.3.4 Struktur der Programmdaten . . . 59
5.3.5 Struktur der Metadaten einer neuen Komponente . . . 59
5.3.6 Gesamtaufbau der Daten im externen Flash . . . 61

**6 Implementierung** **63**
6.1 Aufteilung von nesC-Anwendungen in Komponenten . . . 63
6.1.1 Aufteilen der Systemkomponenten . . . 63
6.1.2 Manuelle Anpassungen . . . 66
6.2 FlexCup-Analyzer . . . 67
6.2.1 Analyse des Programmcodes . . . 68
6.2.2 Speicherung der Informationen auf dem PC . . . 68
6.2.3 Erzeugen der Metadaten . . . 69
6.2.4 Auswertungsfunktionen . . . 69
6.3 FlexCup-Linker . . . 70
6.3.1 Einzelschritte . . . 70
6.3.2 Weitere Implementierungsdetails . . . 74
6.4 FlexCup-Bootloader . . . 78
6.5 Simulation des FlexCup-Linkers auf dem PC . . . 79
6.6 Optimierungen . . . 80
6.6.1 Pufferung der Symboltabelle . . . 80
6.6.2 Verwendung von Pufferbereichen . . . 83
6.6.3 Umgehen der Flashpuffer . . . 87
6.6.4 Spätes Rücksetzen der Änderungsmarkierungen . . . 89
6.6.5 Mögliche weitere Optimierungen . . . 90
6.7 Stand der Gesamtimplementierung . . . 92

**7 Evaluation** **95**
7.1 Theoretische Betrachtungen . . . . . 95
7.1.1 Bedeutung für TinyCubus . . . . . 95
7.1.2 Benötigte Ressourcen . . . . . 96
7.1.3 Systemabhängigkeit der Lösung . . . . . 97
7.1.4 Mangelnde Toleranz gegenüber Abstürzen . . . . . 99
7.2 Praktische Untersuchungen . . . . . 99
7.2.1 Untersuchte Anwendungen . . . . . 99
7.2.2 Untersuchte Codeänderungen . . . . . 101
7.2.3 Bewertungskriterien . . . . . 103
7.2.4 Vorgehen . . . . . 104
7.2.5 Performance-Ergebnisse . . . . . 106

**8 Zusammenfassung und Ausblick** **113**
8.1 Zusammenfassung . . . . . 113
8.2 Ausblick auf zukünftige Arbeiten . . . . . 115
8.2.1 Mögliche Ergänzungen und Erweiterungen . . . . . 115
8.2.2 FlexCup Diff . . . . . 117
8.3 Fazit . . . . . 120

**Literaturverzeichnis** **121**

# 1 Einleitung

## 1.1 Überblick

Sensoren unterschiedlicher Art sind bereits seit vielen Jahren ganz selbstverständlich in zahlreichen Dingen des Alltags enthalten: Thermometer geben Auskunft über die aktuelle Temperatur; Lampen schalten sich automatisch ein, sobald sie Bewegungen in ihrer Umgebung feststellen; Feuermelder schlagen bei übermäßiger Hitze Alarm; Alarmanlagen lösen aus, wenn ein gewisser Geräuschpegel überschritten wird.

Relativ neu ist dagegen noch die Entwicklung, Sensoren zusammen mit einem Mikrocontroller, eigenem Speicher sowie einer Kommunikationsschnittstelle in einem eingebetteten System zu integrieren. Diese Kombination aus Sensor, Rechen- und Kommunikationseinheit wird häufig **Sensorknoten** genannt, mehrere Sensorknoten gemeinsam bilden dann ein so genanntes **Sensornetz**. Der Aufbau und die möglichen Anwendungen solcher Sensornetze haben sich in den letzten Jahren zu einem aktiven Forschungsgebiet entwickelt.

Die fortschreitende technische Entwicklung der Sensorknoten verbunden mit einer zunehmenden Miniaturisierung vieler Bauteile lässt eine Vielzahl unterschiedlicher Anwendungsszenarien realistisch erscheinen. Schon vor mehreren Jahren wurde an der University of California in Berkeley der Begriff **„Smart Dust“** (= Intelligenter Staub) geprägt [HSW+00]. Dieser Ausdruck wird verwendet, um die Vision zu formulieren, dass es in absehbarer Zeit möglich sein könnte, hunderte oder gar tausende winzig kleiner Sensorknoten ähnlich wie Staub auszustreuen. Diese so genannten **„Motes“** (= Stäubchen) sollen sich dann selbstständig in einem Netzwerk organisieren, Merkmale ihrer Umgebung überwachen und ihre Beobachtungen an eine Basisstation melden, wo diese für die jeweilige Anwendung verarbeitet werden können. Auch wenn die in der Forschung aktuell gebräuchlichen Sensorplattformen von der Größe eines Staubkorns noch weit entfernt sind, so ist der Trend zu immer kleineren und bezüglich mancher Aspekte trotzdem mächtigeren Sensorknoten doch klar erkennbar.

Ein wichtiger Faktor bei der Entwicklung von Sensornetzanwendungen ist die **inhärente Ressourcenbeschränktheit** der Sensorknoten. Aufgrund ihrer beschränkten Größe (und in gewissem Maße auch aufgrund der zu begrenzenden Kosten) besitzen Sensorknoten immer nur stark eingeschränkte Rechen-, Speicher- und Batteriekapazitäten. Insbesondere die Einschränkungen bei der zur Verfügung stehenden Energie müssen bei der Entwicklung von Anwendungen in Betracht gezogen werden, da typische Sensornetzanwendungen üblicherweise über einen ausgedehnten Zeitraum betrieben werden sollen, ohne zwischendurch die Batterien aufladen zu können. Das Ziel vieler Entwicklungen ist

deswegen nicht primär, die Leistungsfähigkeit der Sensorknoten immer weiter zu steigern, sondern vielmehr die erforderliche Leistung mit dem kleinstmöglichen Energieaufwand zu erbringen.

Der besondere Nutzen eines Sensornetzes ergibt sich aus der **Zusammenarbeit aller Sensorknoten** innerhalb einer Anwendung. Während einzelne Sensorknoten häufig nicht über ausreichend Ressourcen verfügen, um die Anforderungen einer Anwendung erfüllen zu können, ist es durch eine geschickte Verteilung der Aufgaben auf die Knoten eines Sensornetzes und durch Kooperationen bei der Erfüllung dieser Aufgaben möglich, auch relativ komplexe Anwendungen zu unterstützen. Fällt einer der Sensorknoten während des Betriebs aus, so können eventuell die Nachbarsensorknoten seine Aufgaben mit übernehmen.

Für die Kooperation mit anderen Knoten sowie für die Auslieferung der Daten an eine Basisstation spielt die **Kommunikationsschnittstelle** der Sensorknoten eine wichtige Rolle. In den meisten Fällen kommunizieren die Sensorknoten über Funk, es gibt aber auch alternative Kommunikationsmöglichkeiten, wie beispielsweise die kabelgebundene Kommunikation oder die Kommunikation über Infrarotsignale. Die Kommunikation der Sensorknoten ist ein kritischer Faktor beim Betrieb eines Sensornetzes, da der Betrieb der Funkschnittstelle besonders viel Energie benötigt. Algorithmen für Sensornetzanwendungen müssen demnach immer um Einsparungen bei der Kommunikation der Sensorknoten bemüht sein.

Sensornetze können in einer Vielzahl verschiedener Anwendungsgebiete zum Einsatz kommen. Beispiele für mögliche Anwendungen liegen in der Überwachung natürlicher Lebensräume [MCP+02][CEH+01], im medizinischen Bereich [SGW01][BKM04] oder auch in der Landwirtschaft [BCPR04]. An der Universität Stuttgart gilt ein besonderes Interesse dem Einsatz von Sensornetzen für die Überwachung von Brücken [MSKG05]. Mit Hilfe der Sensoren sollen hier strukturelle Beschädigungen frühzeitig und gleichzeitig kostengünstig erkannt werden.

## 1.2 Problemstellung

In vielen Anwendungsgebieten müssen drahtlose Sensornetze über einen langen Zeitraum möglichst ohne Eingriffe von außen zuverlässig funktionieren. Mit steigender Leistungsfähigkeit und Komplexität der auf den Sensorknoten installierten Software muss man jedoch damit rechnen, dass während einer solchen längeren Betriebsphase in unregelmäßigen Abständen Aktualisierungen notwendig werden, um die Software der Sensorknoten anzupassen. Mit diesen Softwareupdates können entweder Fehler beseitigt oder auch die Funktionalität der Software an geänderte Anforderungen angepasst oder ergänzt werden.

Sind sehr viele Sensorknoten über ein großes, eventuell unzugängliches Gebiet verteilt, so ist es nicht praktikabel, alle Sensorknoten einzusammeln und die Software manuell zu aktualisieren. Es muss stattdessen eine Möglichkeit geben, Softwareupdates automatisch an

die Sensorknoten zu verteilen und dort ohne Eingriffe des Benutzers zu installieren. Für die Verteilung des neuen Programmcodes kommt nur die Funkschnittstelle der Sensorknoten in Frage, für die automatische Installation des Programmcodes wird entsprechende Hardwareunterstützung durch die Sensorknoten benötigt, die es erlaubt, während des Systembetriebs Daten in den Programmspeicher zu schreiben und so Software während des laufenden Betriebs neu zu installieren.

Aufgrund der Ressourcenbeschränkungen der Sensorknoten muss bei der Verteilung neuer Softwarekomponenten über die Funkschnittstelle darauf geachtet werden, möglichst geringe Datenvolumina zu übertragen. Aktuelle Sensorknoten können in vielen Fällen Programme mit einem Umfang von 100 Kilobyte und mehr aufnehmen. Muss jedoch der Programmcode einer solchen Anwendung zwecks Aktualisierungen mehrmals drahtlos an die Sensorknoten übertragen werden, so kann dies einen erheblichen Energieaufwand verursachen, der die Batterien der Sensorknoten stark belastet. Es sind also Lösungen notwendig, die den Programmcode der Sensorknoten auf intelligente Weise aktualisieren, ohne dabei vollständige Programme übertragen zu müssen.

Das automatische Aktualisieren der Software von Sensorknoten gestaltet sich noch deutlich schwieriger, wenn kein vollständig homogenes Netzwerk vorliegt. Wenn die Elemente eines Sensornetzes unterschiedliche Aufgaben besitzen und dafür mit unterschiedlicher Software (und Hardware) ausgestattet sind, so können Softwareupdates nicht ungefiltert an alle Sensorknoten verteilt werden. Stattdessen dürfen sie nur an die tatsächlich betroffenen Sensorknoten ausgeliefert werden. Wenn Sensorknoten jedoch in einem Teil ihrer Software übereinstimmen, so wäre es gleichzeitig ineffizient, Aktualisierungen dieser Teile für jeden Sensortyp getrennt auszuliefern.

## 1.3 Zielsetzung

Aufbauend auf einem bereits existierenden Algorithmus zur Verteilung von Programmcode soll in dieser Arbeit ein Verfahren für den Austausch von Programmkomponenten in Sensornetzen entwickelt und prototypisch implementiert werden. Als Arbeitsgrundlage dienen dabei Sensorknoten der Mica2-Sensorplattform [Crob] und das TinyOS-Betriebssystem [LLWC03] [HSW+00].

Die Grundidee des Ansatzes liegt im dynamischen Austausch einzelner Komponenten einer Sensornetzanwendung. Zu diesem Zweck soll in dieser Arbeit ein Verfahren entworfen werden, das die vorhandene Komponentenstruktur typischer TinyOS-Anwendungen im erzeugten Programmcode erhält und während des Codeupdates ausnutzt. Anstatt bei der Aktualisierung einer Anwendung ihren vollständigen Programmcode zu übertragen, sollen in dieser Lösung nur die geänderten Komponenten an die Sensorknoten übertragen und dort dynamisch in das installierte Programm integriert werden.

Auf Seiten der Sensorknoten erfordert dieser Lösungsansatz die Implementierung eines Programmteils, der – ähnlich wie ein Linker – die empfangenen Komponenten mit den

bereits installierten Komponenten des Programms zusammenfügt und die Verbindungen zwischen den Programmteilen herstellt.

Der in dieser Arbeit entwickelte Lösungsansatz soll zukünftig einen Teil des `TinyCubus`-Projektes [MLM+04][MLM+05] bilden. Ein besonderes Augenmerk muss deswegen bei der Entwicklung den Anforderungen von `TinyCubus` gelten. Diese Anforderungen implizieren unter anderem, dass Sensorknoten zukünftig selbstständig entscheiden können, welche Komponenten sie installieren und wann sie eine Komponente gegen eine andere Komponente austauschen.

Eine weitere wichtige Aufgabe liegt in der Evaluation der Ergebnisse. Der Lösungsansatz soll dabei sowohl qualitativ als auch quantitativ mit anderen Konzepten verglichen werden, die ebenfalls Lösungen für den Austausch von Programmcode in Sensornetzen anbieten.

## 1.4 Aufbau dieser Ausarbeitung

Der Rest dieser Ausarbeitung ist wie folgt aufgebaut: Das nächste Kapitel beschreibt zunächst die Grundlagen, auf denen diese Arbeit basiert. Kapitel 3 präsentiert dann einige verwandte Ansätze, die ähnliche Ziele wie diese Arbeit verfolgen. Anschließend wird im vierten Kapitel das `TinyCubus`-Projekt genauer vorgestellt und die Anforderungen an diese Arbeit im Rahmen des Projektes werden beschrieben. Kapitel 5 skizziert den verfolgten Lösungsansatz, dessen praktische Umsetzung anschließend in Kapitel 6 beschrieben wird. Kapitel 7 schildert Ergebnisse der theoretischen und praktischen Evaluation des Lösungsansatzes. Im letzten Kapitel werden schließlich die Ergebnisse der Arbeit zusammengefasst und es erfolgt ein Ausblick auf mögliche zukünftige Arbeiten in diesem Gebiet.

# 2 Grundlagen

In diesem Kapitel werden einige wichtige Grundlagen beschrieben, auf denen diese Arbeit aufbaut. Zunächst wird die in der Implementierung verwendete Mica2-Sensorplattform vorgestellt. Anschließend werden mit TinyOS und nesC das Betriebssystem und die Programmiersprache der Sensornetzanwendungen beschrieben. Der letzte Abschnitt geht dann detailliert auf den Kompiliervorgang des AVR-GCC-Compilers ein.

## 2.1 Die Mica2-Sensorplattform

Die Sensorknoten der Mica2-Familie (auch Mica2-Motes genannt) [Crob][Cro04] gehören zur dritten Generation der an der University of California in Berkeley entwickelten Mote-Sensorknoten [HC02]. Sie werden von der Firma Crossbow Technology [Croa] gefertigt und sind im Handel allgemein verfügbar. In Abbildung 2.1 ist ein Mica2-Mote abgebildet. Zu sehen sind die Platine des Motes, die auf der Batteriehalterung montiert ist, sowie die Antenne, die für die drahtlose Kommunikation benötigt wird.

Der Mica2-Mote beinhaltet einen **Atmel ATmega128L-Prozessor**, einen **externen Flashbaustein**, eine **Funkschnittstelle**, sowie einen **Anschluss für verschiedene Sensorbausteine**. Die Stromversorgung wird durch zwei Mignon(AA)-Batterien geliefert, die gemeinsam für über 60% des Gesamtgewichts von zirka 45 Gramm verantwortlich sind.

Abbildung 2.1: Mica2-Mote im Überblick

Als einzige direkte Schnittstelle zwischen dem Mica2-Sensorknoten und dem Benutzer stehen **drei Leuchtdioden** (in gelb, grün und rot) zur Verfügung. Diese Leuchtdioden bieten dem Programmierer eine einfache Möglichkeit, den Benutzer über grundlegende Ereignisse im System zu informieren. Der Benutzer wiederum kann nur durch das Aus- und Einschalten des Systems oder durch die Übertragung passender Nachrichten per Funk Einfluss auf den Systembetrieb nehmen. Weitergehende, direkte Interaktionsmöglichkeiten sind nicht vorgesehen, da in den meisten zu erwartenden Einsatzgebieten der Sensorknoten keine regelmäßige Interaktion zwischen dem Benutzer und dem Sensorknoten auftreten dürfte.

### 2.1.1 Prozessor

Der Atmel ATmega128L [Atm04b] ist ein 8-Bit Prozessor, basierend auf der AVR-RISC Architektur. Er besitzt 32 8-Bit-Register, vier Kilobyte internen SRAM sowie einen vier Kilobyte großen internen EEPROM-Speicher („Electrically Eraseable Programmable Read Only Memory“-Speicher). Der interne Programmspeicher besitzt eine Größe von 128 Kilobyte.

Beim Befehlssatz der AVR Prozessoren [Atm02] handelt es sich um eine Ausprägung des RISC („Reduced Instruction Set Computer“)-Ansatzes mit einem relativ kleinen, einfachen Satz an Befehlen. Der AVR-Befehlssatz ist speziell darauf ausgelegt, kompilierten C-Programmcode effizient auszuführen. Der ATmega128L unterstützt insgesamt 133 unterschiedliche Befehle.

Der ATmega128L verwendet, wie alle Prozessoren der AVR-Familie, die so genannte Harvard-Architektur. Dies bedeutet, dass der Prozessor getrennte Speicher und getrennte Bussysteme für Programme und Daten besitzt. Die Architektur erlaubt potentiell eine höhere Performance und mehr Parallelität, da der Prozessor gleichzeitig Befehle und Daten aus dem Speicher lesen kann. Ein Nachteil der Harvard-Architektur besteht in der Inflexibilität, Daten aus dem Datenspeicher nicht als Befehle interpretieren zu können.

#### Programmspeicher: Anwendungs- und Bootloader-Abschnitt

Der interne Programmspeicher des ATmega128L mit seiner Größe von 128 Kilobyte ist in 512 so genannte Pages aufgeteilt. Jede Page besitzt eine Größe von 256 Bytes. Zusätzlich ist der Programmspeicher logisch in zwei Abschnitte aufgeteilt, den Anwendungsabschnitt und den Bootloader-Abschnitt. Die Größe dieser beiden Abschnitte ist konfigurierbar: Der Bootloader-Abschnitt kann entweder vier, acht, 16 oder 32 Pages umfassen, der Anwendungsabschnitt nimmt jeweils den restlichen verfügbaren Platz im Programmspeicher ein. Im **Anwendungsabschnitt** des Programmspeichers sind die normalen Anwendungsdaten gespeichert, die während des laufenden Systembetriebs verwendet werden. Der **Bootloader-Abschnitt** ist dagegen für speziellen Programmcode bestimmt, der das Beschreiben des Programmspeichers ermöglicht.

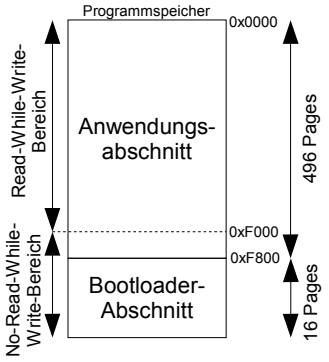


Abbildung 2.2: Mögliche Aufteilung des Programmspeichers (nach [Atm04b])

Der ATmega128L besitzt die Fähigkeit, seinen eigenen Programmspeicher zu überschreiben. Dies ermöglicht flexible, durch den Prozessor gesteuerte Updates der Anwendungssoftware während des Systembetriebs. Das Schreiben des Programmspeichers wird mit Hilfe des Bootloader-Abschnitts realisiert: Programmcode, der innerhalb des Bootloader-Abschnitts liegt, kann den gesamten Speicher überschreiben, inklusive sich selbst. Hierfür wird die SPM-Anweisung („**S**tore **P**rogram **M**emory“) verwendet, die innerhalb des Anwendungsabschnitts deaktiviert ist.

Für das Beschreiben des Programmspeichers muss noch zwischen zwei weiteren Bereichen unterschieden werden, dem **„Read-While-Write“-Bereich** (RWW-Bereich) sowie dem **„No-Read-While-Write“-Bereich** (NRWW-Bereich). Während ein Schreibzugriff auf eine Page des RWW-Bereichs erfolgt, kann gleichzeitig eine Page des NRWW-Bereichs gelesen werden. Erfolgt dagegen ein Schreibzugriff auf eine Page des NRWW-Bereichs, so ist der Prozessor bis zum Abschluss der Schreiboperation blockiert.

Der RWW-Bereich umfasst die ersten 480 Pages des Programmspeichers, der NRWW-Bereich die restlichen 32 Pages. Die Grenze zwischen dem RWW-Bereich und dem NRWW-Bereich muss damit nicht in allen Fällen mit der Grenze zwischen dem Anwendungsabschnitt und dem Bootloaderabschnitt übereinstimmen. Der Bootloader-Abschnitt liegt jedoch in jedem Fall innerhalb des NRWW-Bereichs. In der Praxis bedeutet dies, dass Schreiboperationen des Bootloaders auf den größten Teil des Anwendungsabschnitts die CPU nicht blockieren und dass damit parallel zur Abarbeitung des Schreibzugriffs weitere Operation durchgeführt werden können.

Abbildung 2.2 zeigt als Beispiel die Aufteilung des Programmspeichers in die einzelnen Bereiche mit einem Anwendungsbereich, der 496 Pages umfasst, und einem Bootloader-Abschnitt mit einer Größe von 16 Pages.

Der Bootloader-Abschnitt leitet seinen Namen aus der Tatsache ab, dass er im Normalfall ein so genanntes Bootloader-Programm enthält. Der ATmega128L kann so konfiguriert werden, dass bei jedem Neustart des Systems zunächst der Bootloader aufgerufen wird und nicht das Anwendungsprogramm. Der Bootloader entscheidet dann, ob neuer Pro-

grammcode im Speicher installiert werden muss. Ist dies der Fall, so führt er die notwendigen Schreiboperationen durch und startet erst anschließend das Anwendungsprogramm, ansonsten springt der Bootloader direkt in den Anwendungsbereich. Alternativ zur Ausführung bei jedem Neustart kann man den Bootloader auch per explizitem Sprungbefehl aus dem Anwendungsbereich erreichen. In diesem Fall kann der Bootloader nach dem Abschluss seiner Arbeiten entweder explizit per Sprungbefehl in den Anwendungsbereich wechseln oder einen Reset durchführen, wodurch der Programmablauf automatisch wieder in den Anwendungsabschnitt zurückkehrt.

#### Konfiguration

Das Verhalten des ATmega128L kann über mehrere Schalter, die so genannten „Fuses“, konfiguriert werden. Eine einzelne Fuse wird durch ein Bit repräsentiert und ist entweder gesetzt oder nicht gesetzt. Der ATmega128L besitzt drei Fuse-Bytes, innerhalb derer allerdings lediglich 18 Fuse-Bits mit einer Bedeutung belegt sind. Mit Hilfe der Fuses kann unter anderem konfiguriert werden, welche Größe der Bootloader-Abschnitt im Programmspeicher besitzen soll und ob bei einem Neustart des Systems in den Bootloader-Abschnitt oder in den Anwendungsabschnitt gesprungen werden soll.

### 2.1.2 Externer Flashspeicher

Der Atmel AT45DB041B [Atm04a] ist ein Flashspeicherbaustein, der über eine Kapazität von insgesamt 528 Kilobyte verfügt. Der Speicherplatz ist dabei in 2048 so genannte Pages aufgeteilt, jede Page umfasst damit einen 264 Byte großen Speicherbereich. Der Zugriff auf die Daten des Speicherbausteins erfolgt über ein serielles Interface, das so genannte „Serial Peripheral Interface“ (SPI). Dies bedeutet, dass Anweisungen, Adressen und Daten über den seriellen Eingang an den Speicherbaustein übertragen werden und am seriellen Ausgang die aus dem Speicher gelesenen Daten entgegen genommen werden können.

Für den Zugriff auf den Speicher existieren zwei Möglichkeiten: Der Lese- oder Schreibzugriff kann entweder direkt auf den Flashspeicher erfolgen, oder auf einen von zwei integrierten SRAM-Datenpuffern (siehe Abbildung 2.3). Die zwei Datenpuffer besitzen jeweils eine Kapazität von 264 Byte, sie können also je genau eine Page speichern. Beiden Zugriffsarten gemeinsam ist der serielle Zugriff auf die Daten: Nachdem im Rahmen der einleitenden Befehlssequenz an den Speicherbaustein eine Startadresse für den Lese- beziehungsweise Schreibvorgang definiert wurde, können in den folgenden Betriebszyklen Daten gelesen oder geschrieben werden, ohne erneut eine Zieladresse angeben zu müssen.

Um zwischen dem Lese- und Schreibzugriff sowie zwischen dem Direktzugriff auf den Flashspeicher und dem Zugriff auf einen der beiden Datenpuffer zu unterscheiden, wird jeder Zugriff durch eine Befehlssequenz eingeleitet. Die Befehlssequenz besteht aus dem Opcode, der den Typ des folgenden Zugriffs festlegt, sowie der Pagesdresse und/oder der

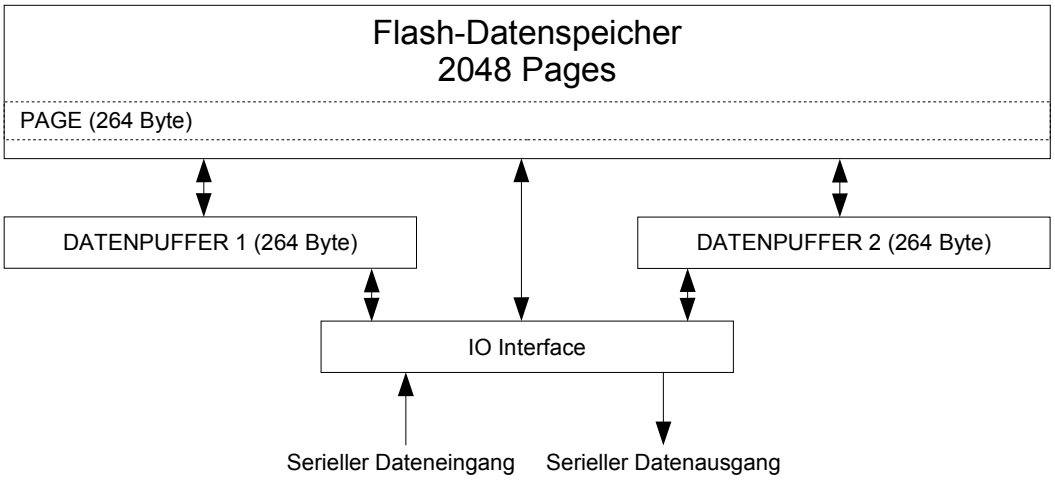


Abbildung 2.3: Zugriffsarten auf den Flashspeicher AT45DB041B (nach [Atm04a])

Byte-Adresse innerhalb einer Page. Tabelle 2.1 gibt einen Überblick über einige unterstützte Befehle und deren Parameter.

| Befehl | Parameter |
|---|---|
| Lese Datenpuffer 1 | Byteadresse im Puffer |
| Lese Datenpuffer 2 | Byteadresse im Puffer |
| Lade vom Speicher in Datenpuffer 1 | Pageadresse im Speicher |
| Lade vom Speicher in Datenpuffer 2 | Pageadresse im Speicher |
| Übertrage Datenpuffer 1 in Speicher | Pageadresse im Speicher |
| Übertrage Datenpuffer 2 in Speicher | Pageadresse im Speicher |
| Lösche Page im Speicher | Pageadresse im Speicher |

Tabelle 2.1: Ausschnitt der Befehlsliste des Atmel AT45DB041B

### 2.1.3 Funkschnittstelle

Die Mica2-Sensorsysteme sind in drei Varianten erhältlich, die sich jeweils nur in der verwendeten Funkschnittstelle unterscheiden: Die Funkschnittstelle der Sensorknoten arbeitet entweder im Frequenzbereich 868/916 MHz, 433MHz oder im Bereich von 315 MHz. Diese drei Versionen sind notwendig, da der verwendete, lizenzfrei nutzbare ISM-Frequenzbereich („**I**ndustrial, **S**cientific, and **M**edical band“) nicht weltweit einheitlich geregelt ist. Als Funkbaustein kommt in allen drei Varianten ein Chipcon CC 1000 zum Einsatz.

Die Datenübertragung erfolgt mit einer Übertragungsrate von 19.2 kbps, als Reichweite im Freien werden zwischen 150 und 300 Meter angegeben.

### 2.1.4 Sensorbausteine

Das Mica2-Grundsystem enthält noch keine Sensoren. Stattdessen können abhängig vom Bedarf der jeweiligen Anwendung verschiedene Sensorbausteine angeschlossen werden. Die Mica2-Systeme sind hierfür mit einem speziellen Mica2-Erweiterungssteckplatz ausgestattet. Für diese Standardschnittstelle sind von Crossbow bereits eine Vielzahl verschiedener Sensorbausteine verfügbar, beispielsweise Licht-, Beschleunigungs-, Feuchtigkeits-, Luftdruck- und Temperatursensoren. In Abbildung 2.1 ist der Mica2-Erweiterungssteckplatz an der Oberseite der Platine zu sehen.

## 2.2 TinyOS

TinyOS [HSW+00] ist ein Betriebssystem, das an der University of California in Berkeley speziell für den Einsatz in vernetzten, eingebetteten Systemen entwickelt wurde. Es ist inzwischen für mehrere Plattformen verfügbar und wird von zahlreichen Forschergruppen, vor allem im Bereich der Sensornetze, eingesetzt.

Ein Betriebssystem für Sensornetzanwendungen sollte einen sehr geringen Speicherverbrauch besitzen und möglichst energieeffizient arbeiten, gleichzeitig aber auch gewisse Realzeitanforderungen erfüllen und einen hohen Grad an Parallelität unterstützen. Mit einem Allzweck-Betriebssystem, das eine Vielzahl von Funktionen und Programmierschnittstellen zur Verfügung stellen muss, ist den Ressourcenbeschränkungen der Sensorsysteme jedoch nur schwer gerecht zu werden. Klassische Betriebssystemkonzepte wie die Trennung von Kernel und Anwendung, eine Multithreading-Architektur, oder auch virtueller Speicher lassen sich mit den beschränkten Fähigkeiten der Sensorknoten ebenfalls nur schwer umsetzen. Die Entwickler von TinyOS haben aus diesem Grund einen neuen Ansatz gewählt, der das Betriebssystem und die Anwendung eng miteinander verbindet und jeweils nur die Systemteile integriert, die von der Anwendung tatsächlich benötigt werden.

TinyOS baut auf einer Komponentenarchitektur auf, bei der sowohl die Anwendungen als auch das TinyOS-Betriebssystem aus einzelnen, miteinander verbundenen Komponenten bestehen. Für die Entwicklung der System- und Anwendungskomponenten von TinyOS steht mit nesC eine spezielle Programmiersprache zur Verfügung. Mehr zum Komponentenmodell von TinyOS und dessen Unterstützung durch nesC findet sich in Abschnitt 2.3.

TinyOS kennt keine feste Trennung zwischen dem Kernel des Betriebssystems und der Anwendungssoftware. Bei dem Betriebssystem handelt es sich auch nicht um ein separates Programm, auf das die einzelnen Anwendungen bei Bedarf aufsetzen können. Stattdessen werden die benötigten Komponenten von TinyOS gemeinsam mit der Anwendung kompiliert. Dabei entsteht ein einzelnes ausführbares Programm, das sowohl die Betriebssystem- als auch die Anwendungskomponenten enthält.

Neben einer Menge von System- und Anwendungskomponenten enthält ein solches ausführbares Programm einen **Scheduler**, der den einzelnen Bearbeitungssträngen Rechenzeit zuweist. Die Entwickler verzichteten auf den Einsatz eines aufwendigen „Multithreading“-Systems und unterstützen Parallelität in TinyOS stattdessen durch ein **Ereignis-basiertes Programmiermodell**. Dieses Programmiermodell baut auf drei Konzepten auf: **Commands**, **Events** und **Tasks**.

Durch den Aufruf eines **Command**-Handlers kann eine Komponente eine Anfrage an eine andere Komponente stellen. *Commands* dürfen den Prozessor grundsätzlich nicht mit zeitaufwendigen Operationen blockieren, sondern müssen bereits nach kurzer Berechnungszeit einen Ergebniswert zurückliefern. Dies bedeutet, dass eventuell auftretende längere Berechnungen in einen *Task* ausgelagert werden müssen. Sobald die Berechnung des *Tasks* abgeschlossen ist, kann die anfragende Komponente mit Hilfe eines *Events* über das Ergebnis der Berechnung informiert werden.

Als direkte oder indirekte Reaktion auf Ereignisse der Hardware werden **Events** generiert und in den Komponenten der Anwendung so genannte *Eventhandler* aufgerufen. Ein *Eventhandler* kann als Reaktion auf einen *Event* einen neuen *Task* erstellen und diesen dem Scheduler zur Ausführung übergeben, er kann aber auch *Events* in anderen Komponenten auslösen oder über *Commands* Funktionen anderer Komponenten aufrufen. Genau wie *Commands* dürfen auch *Events* nur eine kurze Laufzeit besitzen.

Während *Commands* und *Events* die Komponenten miteinander verbinden, sind **Tasks** für die eigentliche Arbeit der Komponenten verantwortlich. Sie werden vom TinyOS-Scheduler in einer FIFO-Warteschlange angeordnet und nacheinander abgearbeitet. *Tasks* können bei Bedarf jederzeit durch *Events* unterbrochen werden, dafür dürfen *Tasks* den Prozessor auch für längere Bearbeitungszeiten in Anspruch nehmen.

## 2.3 nesC

Die Programmiersprache nesC [GLvB+03] [GLCB03] wurde speziell für die Programmierung von Anwendungen in vernetzten eingebetteten Systemen entwickelt. Die bisherigen Implementierungen der Sprache zielen dabei besonders auf die Softwareentwicklung im Bereich der Sensornetze ab. Das Ziel von nesC ist, durch sein Sprachdesign die speziellen Anforderungen dieses Anwendungsgebiets besser zu unterstützen, als es beispielsweise die ansonsten hierfür gebräuchliche Programmiersprache C kann.

Das Programmiermodell von nesC unterstützt **ereignisorientierte Programmierung**, also das Ausführen von Funktionen in Reaktion auf bestimmte Systemereignisse. Die ereignisorientierte Programmierung erlaubt es auf einfache Art und Weise, Funktionen für die gezielte Reaktion auf Ereignisse in der Umgebung zu implementieren. Da in typischen Anwendungen im Bereich der Sensornetze überdurchschnittlich häufig auf solche externen Ereignisse reagiert werden muss, vereinfacht dieses Sprachkonzept das Erstellen typischer Programme.

Eine weitere Besonderheit von nesC ist sein einfaches, ereignisbasiertes **Nebenläufigkeitsmodell**.

nesC besitzt eine **komponentenbasierte Anwendungsarchitektur**: Anwendungen und Systembibliotheken setzen sich aus einzelnen Komponenten zusammen, die abhängig von den Anforderungen flexibel zusammengefügt und miteinander verbunden werden können. Wie bereits in Abschnitt 2.2 erwähnt, wurde die aktuelle Version des TinyOS-Betriebssystems in nesC geschrieben. Das Komponentenmodell von nesC konnte dementsprechend dazu verwendet werden, die Komponentenarchitektur von TinyOS umzusetzen.

### 2.3.1 nesC-Compiler

Bei nesC handelt es sich um eine Erweiterung der Programmiersprache C. Im ersten Schritt des Übersetzungsvorgangs wird der Programmcode der nesC-Quelldateien in regulären C-Programmcode übersetzt. Die Übersetzung in den architekturabhängigen Maschinencode erfolgt dann anschließend mit einem Standard-C-Compiler. Dieses zweistufige Vorgehen bietet mehrere Vorteile: Zum einen stellt es sicher, dass mit überschaubarem Aufwand nesC-Compiler für unterschiedliche Plattformen erstellt werden können, da für die meisten Plattformen bereits C-Compiler vorhanden sind. Die Aufgabe der C-Compiler ist es dann auch, den von nesC produzierten C-Quellcode in effizienten Maschinencode zu übersetzen. Ein weiterer Vorteil besteht darin, dass bestehender C-Quellcode in nesC-Programme integriert und so weiterverwendet werden kann.

Eine Haupteigenschaft von nesC ist seine komponentenbasierte Architektur. Während des Übersetzungsvorgangs wird die Aufteilung des Programmcodes in Komponenten jedoch nicht erhalten. Stattdessen wird der Code aller Anwendungskomponenten zusammen mit dem Code der Systemkomponenten in einer C-Quelldatei zusammengefügt und dann als Gesamtprogramm übersetzt. Nur sehr wenige, hardwarespezifische Funktionen werden aus externen Bibliotheken importiert und vom Linker mit dem Code des Hauptprogramms verbunden.

### 2.3.2 Sprachaufbau von nesC

#### Kommandos und Ereignisse

Kommandos (Schlüsselwort „command“) und Ereignisse (Schlüsselwort „event“) sind zwei Arten von Funktionen in nesC, mit deren Hilfe die Komponenten eines Programms miteinander kommunizieren. Durch den Aufruf eines Commands kann eine Komponente Anfragen an eine andere Komponente formulieren, während sie die andere Komponente durch das Auslösen eines Events (durch den Aufruf einer Eventhandler-Funktion) über bestimmte (System-)Ereignisse informiert.

```
interface BeispielInterface
{
  command result_t start();
  command result_t stop();
  event result_t fired();
}
```

Abbildung 2.4: Beispiel eines Interfaces in nesC

### Interfaces

Das Verhalten einer Komponente wird nach außen durch so genannte Interfaces beschrieben. Ein Interface gibt vor, welche Commands eine Komponente unterstützen muss, um ein Interface zu implementieren. Außerdem definiert sie eine Menge von Events, für die eine Komponente Behandlungsroutinen bereitstellen muss, falls sie das betreffende Interface verwenden möchte.

Abbildung 2.4 zeigt ein Interface mit zwei Commands und einem Event. Implementiert eine Komponente dieses Interface, so ist bekannt, dass die Komponente die Commands *start()* und *stop()* enthält und den Event *fired()* auslösen kann. Verwendet dagegen eine Komponente das Interface *BeispielInterface*, dann kann sie die Commands *start()* und *stop()* dieses Interfaces aufrufen und muss im Gegenzug einen Eventhandler zur Bearbeitung der *fired()*-Events bereitstellen.

### Module

Als Module (Schlüsselwort „module“) werden diejenigen Komponenten von nesC bezeichnet, die den eigentlichen Anwendungscode des nesC-Programms enthalten. Ein Modul kann dabei beliebig viele unterschiedliche Interfaces implementieren und für die Erfüllung seiner Aufgabe eine beliebige Anzahl von Interfaces verwenden. Abhängig von den angebotenen und verwendeten Interfaces enthält ein Modul Referenzen auf Commands und Eventhandler anderer Komponenten beziehungsweise bietet Commands und Eventhandler an, die von anderen Komponenten aufgerufen werden können.

Abbildung 2.5 zeigt anhand eines Beispiels den typischen Aufbau eines nesC-Moduls. Das Beispielmodul implementiert (*provides*) die Interfaces *BeispielInterface* und *StdControl* und verwendet (*uses*) das Interface *Leds*. Innerhalb des *implementation*-Abschnitts stehen die Implementierungen der Commands und Eventhandler des Moduls. In der Abbildung werden nur das Command *start()* des *BeispielInterface*-Interfaces sowie das Command *init()* des *StdControl*-Interfaces gezeigt. Innerhalb dieses Commands wird *Leds.init()* aufgerufen – das *init()*-Command des *Leds*-Interfaces.

```
module BeispielModul
{
  provides {
    interface BeispielInterface;
    interface StdControl;
  }

  uses {
    interface Leds;
  }
}

implementation {

  command result_t BeispielInterface.start() {
    [...]
  }

  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }

  [...]

}
```

Abbildung 2.5: Beispiel eines Moduls in nesC

```
configuration FlashReadWriter { }
implementation {
  components Main, BeispielModul, Test, LedsC;

  Main.StdControl -> BeispielModul;
  Test.BeispielInterface -> BeispielModul;
  BeispielModul.Leds -> LedsC;

}
```

Abbildung 2.6: Beispiel einer Konfiguration in nesC

### Konfigurationen

Die Module eines nesC-Programms definieren lediglich, welche Interfaces sie nach außen anbieten und welche Interfaces sie zu diesem Zweck benötigen. Von welchen Modulen die angebotenen Interfaces tatsächlich genutzt werden und welche Module die benötigten Interfaces zur Verfügung stellen, muss außerhalb der Module festgelegt werden. Diese Verbindung zwischen den einzelnen Modulen herzustellen ist die Aufgabe der Konfigurationskomponenten.

Eine Konfiguration (Schlüsselwort „configuration“) vereinigt eine Menge von Komponenten und sorgt für die Verbindungen zwischen angebotenen Interfaces einer Komponente und den entsprechenden benötigten Interfaces anderer Komponenten. Um ein vollständiges nesC-Programm erzeugen zu können, muss die Konfigurationskomponente sicherstellen, dass jedem angeforderten Interface ein Angebot des gleichen Interfaces gegenübersteht, und dass alle angebotenen Interfaces mit einem oder mehreren angeforderten Interfaces gleichen Typs verbunden sind.

Das Beispiel in Abbildung 2.6 zeigt eine Konfiguration, welche die vier Komponenten *Main*, *BeispielModul*, *Test* und *LedsC* verbindet. Die drei Zeilen, in denen ein Pfeil (`->`) enthalten ist, spezifizieren, welche Interfaces welcher Komponenten miteinander verbunden sind. Die Pfeilrichtung gibt dabei an, welche Komponente das Interface anbietet und welche Komponente es benutzt. Im Beispiel verwendet die Komponente *Main* das *StdControl*-Interface der Komponente *BeispielModul*. *Test* verwendet das Interface *BeispielInterface* der Komponente *BeispielModul* und *BeispielModul* wiederum verwendet das *Leds*-Interface der *LedsC*-Komponente. Ein Vergleich mit dem Modul in Abbildung 2.5 zeigt, dass alle von *BeispielModul* angebotenen und verwendeten Interfaces mit Interfaces anderer Komponenten verbunden wurden.

In den letzten Abschnitten wurde von der Verbindung zwischen Komponenten gesprochen, nicht von der Verbindung zwischen Modulen. Dies hängt damit zusammen, dass neben Modulen auch Konfigurationen miteinander verbunden werden können: Anstatt für alle Interfaces der enthaltenen Komponenten eine Verbindung festzulegen, kann eine

Konfiguration das betreffende Interface auch in seinen *provides*- beziehungsweise *uses*-Abschnitt aufnehmen. Nach außen hin tritt eine solche Konfiguration in diesem Fall auf wie ein Modul, das Interfaces implementiert und andere Interfaces referenziert. Mindestens eine weitere Konfigurations-Komponente ist nun notwendig, um diese Konfiguration mit weiteren Modulen oder auch weiteren Konfigurationen zu verdrahten.

In jedem nesC-Programm existiert genau eine Konfiguration auf höchster Ebene, die weder Interfaces exportiert noch Interfaces importiert. Von ihr aus sind alle Konfigurationen und über diese auch alle Module miteinander verbunden. Es handelt sich dabei jedoch nicht um eine rein hierarchische Beziehung, da es auch Querverbindungen zwischen Konfigurationen geben kann. Dies ist beispielsweise der Fall, wenn zwei Konfigurationen das gleiche Modul oder die gleiche Konfigurationen einbinden. Der nesC-Compiler enthält mit **nesdoc** ein Werkzeug, das die Beziehung der Komponenten einer Anwendung in Form eines Graphen abbildet (Ein Beispiel hierfür findet sich in Abbildung 6.1 in Abschnitt 6.1.1.).

### 2.3.3 Binäre Komponenten in nesC 1.2

In der Version 1.2 von nesC, die sich derzeit noch im Alpha-Stadium befindet, kommt ein neues Konzept zur nesC-Sprache hinzu: die Erstellung und Nutzung binärer Komponenten. Das Konzept der binären Komponenten ermöglicht dem Benutzer, einzelne nesC-Module oder auch ganze Programmteile getrennt vom Rest des Gesamtprogramms zu kompilieren. Die einzelnen binären Komponenten werden dann erst vom Linker zum Gesamtprogramm zusammengesetzt. Dieses Vorgehen ermöglicht es beispielsweise, dem Benutzer Komponenten in Form binärer Bibliotheken zur Verfügung zu stellen, ohne Zugriff auf den Sourcecode gewähren zu müssen.

#### Einbinden binärer Komponenten

Um den Zugriff auf eine binäre Komponente zu ermöglichen, wird ein neues Sprachkonstrukt verwendet, das anhand eines Beispiels in Abbildung 2.7 gezeigt wird. Eingeleitet durch das Schlüsselwort *component* wird hier – analog zur Darstellung in nesC-Modulen und nesC-Konfigurationen – definiert, welche Interfaces, Commands und Events die Binärkomponente anbietet (*provides*) beziehungsweise benötigt (*uses*). Die Binärkomponente aus dem Beispiel stellt demnach das Interface *StdControl* und den Befehl *init()* zur Verfügung und erfordert das Interface *Leds*.

Die Komponentendefinition einer Binärkomponente kann in einer Konfiguration auf die gleiche Art und Weise in ein nesC-Programm eingebunden und verdrahtet werden, wie es für Module oder Konfigurationen möglich ist. Die Komponentendefinition dient dabei als Platzhalter für die eigentliche Binärkomponente, so dass ein referenzierendes Programm kompiliert werden kann, ohne dass der Sourcecode der Binärkomponente vorliegen muss.

```
component BeispielKomponente
{
  provides {
    interface StdControl;
    command result_t init();
  }

  uses {
    interface Leds;
  }
}
```

Abbildung 2.7: Beispiel der Komponentendefinition einer Binärkomponente

Wenn der Compiler die Objektdatei für eine Anwendung erstellt, die auf Funktionen einer Binärkomponente zugreift, so müssen für die Zugriffe von der Anwendung auf die Binärkomponente externe Referenzen und für die Zugriffe der Binärkomponente auf die Anwendung entsprechend externe Symbole erstellt werden. Konkret bedeutet dies, dass der Compiler für alle von der Binärkomponente angebotenen Commands und alle von ihr verwendeten Events Referenzen auf externe Symbole erstellt. Daneben muss er für alle von der Binärkomponente verwendeten Commands und alle angebotenen Events externe Symbole erstellen, die von der Binärkomponente referenziert werden. Die Benennung der Symbole und der Symbolreferenzen geschieht nach folgendem Muster: „`Komponentenname$Interfacename$Event-oder-Commandname`“. Im Beispiel aus Abbildung 2.7 wird demnach unter anderem folgendes Symbol generiert: „`BeispielKomponente$StdControl$start()`“.

### Erstellen binärer Komponenten

Das Erstellen binärer Komponenten kann ebenfalls mit Hilfe des oben beschriebenen neuen Sprachkonstrukts geschehen. Man muss hierfür eine Komponentendefinition erstellen, deren angebotene und verwendete Interfaces, Commands und Events genau komplementär zu den angebotenen und verwendeten Interfaces, Commands und Events der zu erstellenden Binärkomponente sind. Möchte man die Binärkomponente aus dem Beispiel in Abbildung 2.7 erstellen, die das Interface *StdControl* und den Command *init()* zur Verfügung stellt und das Interface *Leds* benötigt, so erstellt man eine Komponentendefinition in der *provides* und *uses* vertauscht sind, in der also das Interface *Leds* angeboten wird und *StdControl* sowie *init()* benötigt werden.

Es ist wichtig, dass die zur Erstellung der Binärkomponente verwendete Komponentendefinition denselben Komponentennamen besitzt, wie die Komponentendefinition, die zum Einbinden der Komponente in nesC-Anwendungen verwendet wird. Nur auf diese Weise

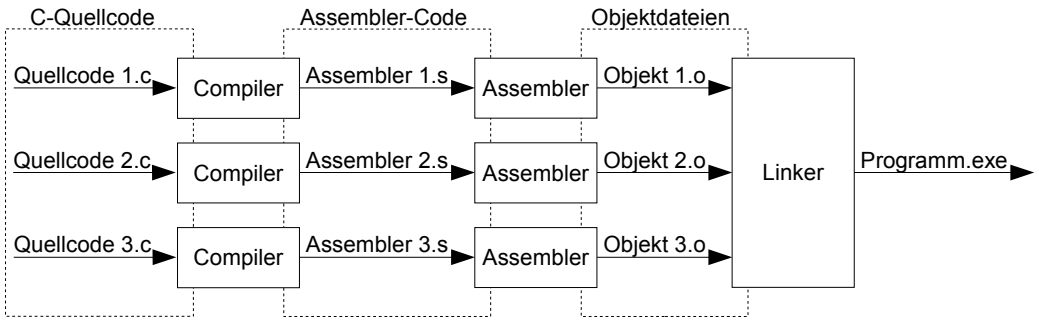


Abbildung 2.8: Der Kompiliervorgang

ist sichergestellt, dass die verwendeten Symbole der gegenseitigen Referenzen kompatibel zueinander sind und eine Verbindung der Objektdateien durch den Linker gelingen kann.

## 2.4 Kompilieren für AVR-Prozessoren

Für das Erstellen von AVR-Binärcode aus C-Programmen steht mit AVR-GCC eine spezielle Version der GCC-Compiler [Frec] zur Verfügung. Das Compilerpaket besteht aus den üblichen Teilkomponenten: Zunächst wird der C-Quellcode vom Compiler in Assemblercode umgewandelt, bevor anschließend der Assembler daraus eine Objektdatei erstellt. Die Aufgabe des Linkers ist es dann, alle Objektdateien eines Programms zusammenzufügen und zu einem Gesamtprogramm zu verbinden (siehe Abbildung 2.8).

In diesem Abschnitt wird auf einige Details des Kompiliervorgangs mit AVR-GCC eingegangen. Außerdem werden wichtige Eigenschaften der dabei erstellten Objektdateien und Programme vorgestellt.

### 2.4.1 Speicherlayout

Wie bereits in der Beschreibung des ATmega128L erwähnt, verwenden die AVR-Prozessoren die Harvard-Architektur für die Organisation des Speichers. Dies bedeutet, dass der Programmspeicher und der Datenspeicher voneinander getrennt arbeiten und unterschiedliche Adressbereiche belegen. Dieser Teil des Speicherlayouts ist also bereits durch die Hardware vorgegeben. Im Folgenden soll nun kurz beschrieben werden, wie der AVR-GCC-Compiler die Programmdaten und die Anwendungsdaten im Speicher verteilt.

### Datenspeicher

Man kann generell zwischen zwei Arten von Datensymbolen unterscheiden: den Datensymbolen, die mit einem Startwert initialisiert werden und den Datensymbolen ohne Initialisierungswert. Ein Datensymbol besitzt einen Initialwert, falls die zugehörige Variablendefinition im C-Quellcode eine Wertzuweisung ungleich 0 enthält, zum Beispiel `uint16_t TOS_LOCAL_ADDRESS = 1`. Der AVR-GCC-Linker behandelt die zwei Datensymbol-Arten unterschiedlich und platziert die Symbole deswegen in getrennten Abschnitten. Die Datensymbole mit Initialisierungswert werden in dem so genannten *.data*-Abschnitt gespeichert, die Symbole ohne Initialisierungswert im so genannten *.bss*-Abschnitt.

Die **Symbole des *.data*-Abschnitts** lassen sich jeweils eindeutig einer Objektdatei zuordnen, da es für jedes Symbol genau eine Objektdatei geben muss, die einen Initialwert für das Symbol enthält. Aus Sicht der anderen Objektdateien lässt sich vor dem Linken nicht entscheiden, ob es sich bei einem Datensymbol um ein Symbol des *.bss*-Abschnitts handelt, oder ob eine andere Objektdatei Initialisierungsdaten für das Symbol enthält und das Symbol aus diesem Grund Teil des *.data*-Abschnitts ist. Enthalten mehrere Objektdateien einen Initialwert für das gleiche Symbol, so generiert der Linker eine Fehlermeldung.

Bereits in den einzelnen Objektdateien ist das Layout der Datensymbole innerhalb der jeweiligen *.data*-Abschnitte vorgegeben und wird durch Einträge der Relocationtabelle, die relativ zum Beginn des *.data*-Abschnitts definiert sind (siehe Abschnitt 2.4.2), verwendet. Dementsprechend generiert der Linker den *.data*-Abschnitt des Gesamtprogramms, indem er die *.data*-Abschnitte der Objektdateien einfach aneinander reiht.

Da der Programmspeicher im Gegensatz zum Datenspeicher nicht flüchtig ist, werden darin die Initalisierungsdaten für die Symbole des *.data*-Abschnitts des Programms gespeichert. Nach dem Einschalten des Sensorknotens und während eines Startvorgangs in Folge eines Resets werden diese Initialisierungsdaten an die entsprechende Adresse im Hauptspeicher geladen. Hierfür ist eine spezielle Systembibliothek zuständig, die vom Linker eingebunden wird.

Die **Symbole des *.bss*-Abschnitts** sind nicht eindeutig einer Objektdatei als „Besitzer“ zuzuordnen, da sie in keiner der Objektdateien einen Initialwert zugewiesen bekommen. Die Zugriffe auf ein *.bss*-Symbol aus den einzelnen Objektdateien heraus erfolgen völlig gleichberechtigt. Aus diesem Grund können hieraus keine Vorgaben für das Layout der einzelnen Symbole abgeleitet werden. Stattdessen platziert der Linker des AVR-GCC die *.bss*-Symbole in der Reihenfolge, in der er sie erstmals liest.

Einen Sonderfall innerhalb des *.bss*-Abschnitts bilden statische Variablen, die innerhalb einer Funktion definiert werden. Auf diese Variablen kann nur aus der definierenden Funktion zugegriffen werden, nach außen bleiben sie unsichtbar. Dementsprechend lassen sich diese *.bss*-Variablen eindeutig einem Besitzer zuordnen. Der AVR-GCC-Compiler verzichtet aus diesem Grund darauf, solchen Variablen ein Symbol in der Objektdatei

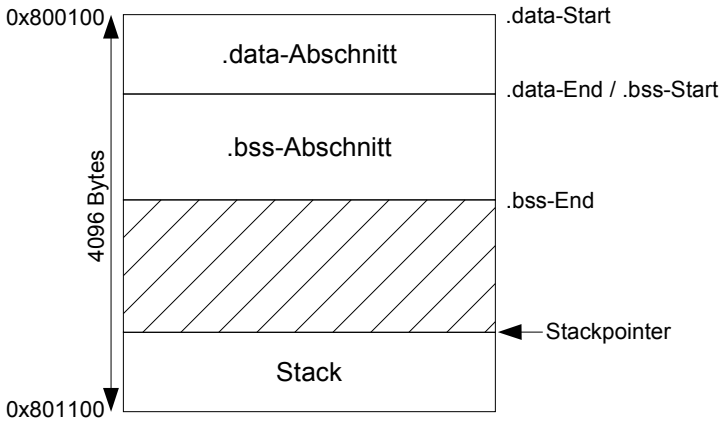


Abbildung 2.9: Layout des Datenspeichers

zuzuordnen (und verhindert dadurch eventuelle Probleme durch Namensgleichheiten). Stattdessen weist er Objektdateien, die statische Variablen in Funktionen enthalten, einen eigenen *.bss*-Abschnitt zu, dessen Größe der Gesamtgröße aller statischer Funktionsvariablen dieser Objektdatei entspricht. Zugriffe auf Werte dieser Variablen geschehen dann per Referenz auf den *.bss*-Abschnitt der Objektdatei plus dem entsprechenden Offset des gewünschten Wertes.

Für die Symbole des *.bss*-Abschnitts müssen keine Initialwerte geladen werden. Der Compiler nimmt für Symbole des *.bss*-Abschnitts jedoch grundsätzlich einen Startwert 0 an (Dementsprechend werden mit dem Wert 0 initialisierte Variablen dem *.bss*-Abschnitt zugeordnet und nicht dem *.data*-Abschnitt). Um auch dies bei Starts und Resets sicherzustellen, bindet der Linker eine weitere Systembibliothek ein, die den Speicherbereich des *.bss*-Abschnitts komplett mit Nullen initialisiert.

Zusätzlich zu den beschriebenen Daten des *.data*- und des *.bss*-Abschnitts muss der Hauptspeicher auch noch den **Stack** aufnehmen. Der Stack wird für die Speicherung temporärer Daten verwendet. Dazu gehören die Rücksprungadressen bei Funktionsaufrufen oder Interrupts sowie die lokalen Variablen der Funktionen in so genannten Aktivierungsrecords. Die Größe des Stacks ist variabel und wird nur durch die Gesamtgröße des Speichers und die Größe der Daten in den *.data*- und *.bss*-Abschnitten begrenzt. Ein Stackpointer zeigt auf das obere Ende des Stacks. Im ATmega128L ist der Stack dabei so implementiert, dass er von größeren Speicherbereichen zu kleineren Speicherbereichen hin wächst. Dementsprechend dekrementiert das Hinzufügen eines Datenobjektes per PUSH-Befehl den Stackpointer, während der POP-Befehl den Stackpointer inkrementiert.

Das Gesamtlayout des Datenspeichers mit *.data*-Abschnitt, *.bss*-Abschnitt und dem Stack ist in Abbildung 2.9 dargestellt.

### Programmspeicher

An vorderster Stelle im Programmspeicher liegt die so genannte **Interrupt-Vektortabelle**. Für jeden möglichen Interrupt des Systems ist darin eine Sprunganweisung enthalten, die angibt, an welcher Adresse im Programmspeicher der zugehörige Interrupt-Handler zu finden ist, der aufgerufen wird, sobald der Interrupt ausgelöst wird. Damit der Prozessor beim Auslösen eines Interrupts automatisch auf die korrekte Position innerhalb der Interrupt-Vektortabelle zugreift, muss sie am Beginn des Programmspeichers platziert sein. Alternativ lässt sich der Prozessor auch so konfigurieren, dass er die Interrupt-Vektortabelle am Beginn des Bootloader-Abschnitts (siehe Kapitel 2.1.1) erwartet.

Der ATmega128L kennt insgesamt 35 unterschiedliche Interrupts. Da die Tabelle für jeden Interrupt eine Sprunganweisung mit 4 Byte Größe enthält, umfasst die Interrupt-Vektortabelle insgesamt 140 Bytes.

Werden im Quellcode eines Programms **statische, konstante Daten** verwendet, so werden diese vom AVR-GCC-Compiler nicht im Datenspeicher, sondern direkt im Programmspeicher angelegt. Dies ist möglich, da auf diese Daten keine Schreibzugriffe erfolgen können. Der Compiler legt zu diesem Zweck in den betroffenen Objektdateien Datenabschnitte mit der Bezeichnung *.progmem.data* an. Soweit vorhanden, werden die *.progmem.data*-Abschnitte der Objektdateien im Programmspeicher direkt hinter der Interrupt-Vektortabelle angelegt.

Die **Codeabschnitte der einzelnen Objektdateien** folgen im Programmspeicher sequentiell aufeinander. Zusätzlich zu dem Programmcode der Objektdateien, die der Programmierer erstellt hat, kommen noch Codeabschnitte verschiedener Systembibliotheken, die für das Ausführen des Programms benötigt werden, sowie die vom Programmierer eingebundenen Code-Bibliotheken.

Wie bereits bei der Beschreibung des Datenspeichers erwähnt, enthält der Programmspeicher die **Initialisierungsdaten der Datensymbole des *.data*-Abschnitts**. Dabei handelt es sich um einen zusammenhängenden Datenblock, in dem die Initialisierungsdaten der einzelnen Symbole in der gleichen Reihenfolge aneinander gehängt werden, in der auch die Symbole innerhalb des *.data*-Abschnitts platziert sind. Auf diese Weise muss der komplette Datenblock einfach an die *data_start*-Adresse kopiert werden um alle *.data*-Symbole mit ihren Startwerten zu initialisieren. Der Datenblock mit den Initialisierungsdaten wird vom AVR-GCC im Programmspeicher direkt im Anschluss an den Programmcode der letzten Komponente platziert.

Abbildung 2.10 zeigt eine Übersicht über das Gesamtlayout des Programmspeichers mit den Interrupt-Vektoren, den statischen *.progmem.data*-Daten, den Codeabschnitten der Objektdateien sowie den Initialisierungsdaten des *.data*-Abschnitts.

Das Gesamtlayout sowohl des Datenspeichers als auch des Programmspeichers wird zwar – wie gezeigt – standardmäßig vom Linker des AVR-GCC-Pakets generiert, für einen korrekten Systembetrieb sind aber auch andere Aufteilungen des Speichers möglich. Koshy

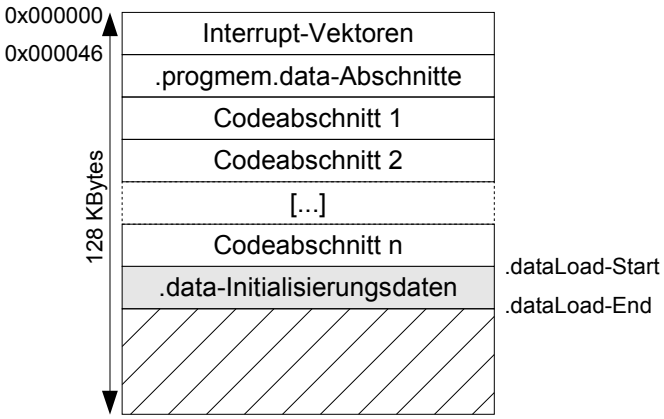


Abbildung 2.10: Layout des Programmspeichers

und Pandey [KP05] platzieren beispielsweise in ihrem Ansatz (siehe Abschnitt 3.2.4) die Initialisierungsdaten des *.data*-Abschnitts am Ende des Programmspeichers anstatt direkt hinter den Codeabschnitten der Objektdateien. Derartige Änderungen am Layout der Daten im Speicher können in vielen Fällen direkt im Linker mit Hilfe so genannter Linker Scripts umgesetzt werden.

### 2.4.2 Objektdateien

Eine Objektdatei enthält fünf unterschiedliche Arten von Informationen [Lev99]: Headerinformationen, den Objektcode, Debugginginformationen, die Relocationtabelle und die Symboltabelle. Das genaue Aussehen einer Objektdatei hängt dabei von dem verwendeten Objektformat ab. Der AVR-GCC-Compiler generiert seine Objektdateien im „Executable and Linking Format (ELF)“ [Too95].

Der **Header** der Objektdatei enthält allgemeine Informationen zu der Objektdatei, wie beispielsweise die Größe des enthaltenen Objektcodes oder den Namen der Quelldatei aus der die Objektdatei erstellt wurde.

Der **Objektcode** einer Objektdatei besteht aus einem oder mehreren Abschnitten. Üblicherweise gehört dazu ein *.text*-Abschnitt, der den Programmcode der Objektdatei enthält. Dazu kommt noch ein *.data*-Abschnitt, falls die Objektdatei vorinitialisierte Variablen enthält, sowie ein *.progmem.data*-Abschnitt, falls in der Objektdatei statische Datenstrukturen enthalten sind. Prinzipiell kann der Programmcode auch in einem Abschnitt untergebracht sein, der nicht mit *.text* bezeichnet ist. Von dieser Möglichkeit machen vor allem die Systembibliotheken Gebrauch.

Die **Debugginginformationen** der Objektdateien können für die Suche nach Fehlern verwendet werden, wenn das Programm innerhalb einer Debugging-Umgebung ausgeführt

wird. Für den normalen Betrieb haben die Debugginginformationen keine Bedeutung und werden vor dem Übertragen des Programmcodes auf die Sensorknoten entfernt.

Die **Relocationtabelle** und die **Symboltabelle** einer Objektdatei enthalten wichtige Informationen, die der Linker für das Erstellen des Programms benötigt. Ihr Inhalt wird im Folgenden genauer beschrieben.

### Die Symboltabelle

Die Symboltabelle gibt Auskunft über die in dem Objektcode definierten globalen Symbole sowie über die durch den Objektcode referenzierten Symbole, die von anderen Modulen importiert werden müssen. Ein Teil der Symbole repräsentiert Funktionen; diese Symbole werden im folgenden Funktionssymbole genannt. Andere Symbole stehen dagegen für Datenobjekte im Programm; diese werden im Folgenden als Objektsymbole bezeichnet.

Abbildung 2.11 zeigt beispielhaft den Ausschnitt einer Symboltabelle, die mit Hilfe von avr-objdump (siehe [Frea]) aus einer Objektdatei ausgelesen wurde. Am Ende jeder Zeile steht der Name des jeweiligen Symbols, direkt davor wird die Größe des Symbols ausgegeben. Die erste Spalte gibt – sofern bereits bekannt – die Adresse des Symbols an.

Die im Beispiel dargestellte Symboltabelle enthält vier Funktionssymbole: *main*, *calculateX*, *calculateY* und *getResult*. Die Funktionen *main* und *getResult* sind Teil des Programmcodes der Objektdatei, *main* liegt direkt am Beginn des Programmcodes an der Adresse 0 und besitzt eine Größe von 162 Bytes (entspricht `A2` Bytes hexadezimal). Analog dazu lässt sich aus der Symboltabelle ablesen, dass die Funktion *getResult* an der Byteadresse 292 (entspricht `124` hexadezimal) beginnt und eine Länge von 6 Bytes besitzt. Die Funktionen *calculateX* und *calculateY* werden durch den Programmcode der Objektdatei referenziert, sie selber sind aber nicht Teil des Objektes. Dementsprechend liegen in der Symboltabelle keine Werte für ihre Adresse und Größe vor und die Symbole sind als undefiniert (*UND*) markiert.

Neben den vier Funktionssymbolen enthält die Symboltabelle in Abbildung 2.11 drei Objektsymbole: *dataObject1*, *dataObject2* und *commonObject1*. *dataObject1* und *dataObject2* sind Teil des *.data*-Abschnitts der Objektdatei, das heißt, die zugehörigen Variablen bekommen in dieser Objektdatei einen Initialwert zugewiesen. Bei *commonObject1* ist dies nicht so eindeutig. Das Symbol wird in der Objektdatei referenziert, es bekommt hier aber keinen Initialwert zugewiesen. Das Symbol kann nun entweder Teil des *.data*-Abschnittes einer anderen Objektdatei sein, oder ohne Initialwert verwendet und dementsprechend Teil des *.bss*-Abschnitts des Gesamtprogramms werden. Worum es sich bei dem Symbol *commonObject1* tatsächlich handelt, kann erst während des Linkens festgestellt werden.

Auch aus der Programmdatei des Gesamtprogramms, das aus den einzelnen Objektdateien im Rahmen des Linkens hervorgeht, lässt sich eine Symboltabelle auslesen. In dieser Symboltabelle sind nun die endgültigen Symbolinformationen enthalten: Für alle Daten-

```
00000000 g     F .text 000000a2 main
00000000         *UND* 00000000 calculateX
00000000         *UND* 00000000 calculateY
00000124 g     F .text 00000006 getResult
00000000 g     O .data 00000002 dataObject1
00000002 g     O .data 00000002 dataObject2
00000001       O *COM* 00000001 commonObject1
```

Abbildung 2.11: Beispielhafter Ausschnitt einer Symboltabelle

```
00000006 R_AVR_CALL        .text+0x00000124
000003f6 R_AVR_CALL        calculateX
0000041c R_AVR_16          commonObject1
0000012a R_AVR_16          .data+0x00000002
```

Abbildung 2.12: Beispielhafter Ausschnitt einer Relocationtabelle

objekte steht fest, ob sie Teil des *.data*-Abschnitts oder des *.bss*-Abschnitts sind, daneben ist nun auch ihre Position im Speicher bekannt. Für die Funktionssymbole ist in dieser Symboltabelle die endgültige Position innerhalb des Gesamtprogramms und die Größe des Binärcodes der Funktion eingetragen.

### Die Relocationtabelle

Neben einer Symboltabelle enthalten Objektdateien auch eine Relocationtabelle. Darin sind alle notwendigen Anpassungen von Adressen im Programmcode verzeichnet, die beim Erstellen des Gesamtprogramms durch den Linker vorgenommen werden müssen. Anpassungen des Programmcodes der Objektdatei sind an allen Stellen notwendig, an denen auf eine andere Stelle im Daten- bzw. Programmspeicher verwiesen wird. Dies ist einerseits beim lesenden oder schreibenden Zugriff auf Datensymbole der Fall, andererseits bei Sprungverweisen auf andere Funktionen.

Ein Eintrag der Relocationtabelle setzt sich aus der Ausgangsadresse der Relocation, deren Typ sowie dem Zielsymbol der Relocation zusammen. In manchen Fällen besteht das Ziel eines Relocationeintrags auch aus einem Zielsymbol und einem Offset, der in diesem Fall zur Adresse des Zielsymbols hinzuaddiert werden muss. Diesen grundsätzlichen Aufbau einer Relocationtabelle zeigt auch das Beispiel in Abbildung 2.12, das einen kurzen Ausschnitt einer Relocationtabelle wiedergibt.

Bei dem ersten Eintrag der Tabelle handelt es sich um die Relocation einer Sprunganweisung (AVR_CALL) an der Adresse `0x0006`. Es handelt sich dabei um den Aufruf einer Funktion in der gleichen Objektdatei. Derartige lokale Sprünge werden vom AVR-GCC-Compiler grundsätzlich relativ zum Start des *.text*-Abschnitts ausgedrückt. Dementspre-

chend gibt der Eintrag als Sprungziel *.text* zuzüglich eines Offsets von `0x0124` (hexadezimal ausgedrückt) an.

Die zweite AVR_CALL-Relocation nimmt Bezug auf das Symbol *calculateX* aus der in Abbildung 2.11 gezeigten Symboltabelle. Der Relocationeintrag gibt an, dass an der Adresse `0x3f6` des Objektcodes eine Sprunganweisung auf das Symbol *calculateX* verweist.

Die beiden AVR_16-Relocations sind ähnlich zu interpretieren wie die gerade beschriebenen AVR_CALL-Relocations. Die erste Relocation an der Adresse `0x041c` verweist auf das Objektsymbol *commonObject1*, die zweite Relocation auf den Beginn des *.data*-Abschnitts zuzüglich eines lokalen Offsets von 2 Bytes. Ein wichtiger Unterschied zu den AVR_CALL-Relocations besteht darin, dass die AVR_16-Relocations auf den Datenspeicher und nicht den Programmspeicher verweisen. Dies hat entscheidende Auswirkungen auf die Bearbeitung der Relocationeinträge, da im Datenspeicher einzelne Bytes adressiert werden, während im Programmspeicher nur 2 Byte große Abschnitte adressiert werden können. Dementsprechend kann bei Referenzen auf den Datenspeicher direkt die Byteadresse des Zielsymbols eingetragen werden, während bei Referenzen auf den Programmspeicher der Adresswert zunächst noch durch zwei dividiert werden muss.

Offsets in der Relocationtabelle sind nicht nur wie im Beispiel gezeigt für die Symbole *.text* und *.data*, sondern auch für andere Symbole der Symboltabelle möglich. Greift eine Anweisung des Programmcodes beispielsweise auf ein Datenelement innerhalb eines Arrays zu, so enthält die Relocationtabelle hierfür eine Referenz auf das Symbol des Arrays und verwendet als Offset die Position des Datenelements innerhalb des Arrays.

Tabelle 2.2 gibt einen Überblick über die 19 unterschiedlichen Relocationtypen, die vom Assembler des AVR-GCC-Compilerpakets abhängig von der jeweiligen Referenz generiert und dann vom Linker passend aufgelöst werden.

Ohne eine vollständige Erklärung der 19 Relocationtypen zu liefern, soll im Folgenden kurz auf einige Details eingegangen werden: Die beiden wichtigsten Relocationtypen sind die bereits im Rahmen des Beispiels beschriebenen „R_AVR_CALL“ und „R_AVR_16“. Daneben wird in Programmen häufig der LDI-Prozessorbefehl („**L**oa**D** **I**mmediate“) eingesetzt, mit dessen Hilfe Konstanten in die Register des Prozessors geladen werden können. Für den LDI-Befehl sowie für verschiedene verwandte Befehle werden die Relocationtypen benötigt, deren Bezeichner die Abkürzung „LDI“ enthalten. Zum einen werden hierbei das untere Byte („LO8“), das obere Byte(„HI8“) und für manche AVR-Prozessoren auch das höchste Byte („HH8“) der Zieladresse getrennt voneinander behandelt. Zum anderen muss anhand des Relocationtyps unterschieden werden, ob die Relocation auf den Programmspeicher verweist („PM“) und ob die Zieladresse vor dem Eintrag negiert werden muss („NEG“). „R_AVR_32“ bezeichnet eine Relocation auf einen 32-Bit-Datenwert, „R_AVR_7_PCREL“ und „R_AVR_13_PCREL“ bezeichnen Relocations für Sprünge relativ zum aktuellen *Program Counter*.

In dem vom Linker erstellten Gesamtprogramm sind alle Referenzen auf Funktionen und Objekte aufgelöst. Dementsprechend kann das Gesamtprogramm zwar noch Informatio-

| Relocationtypen im AVR-GCC-Compiler |
|---|
| R_AVR_NONE |
| R_AVR_32 |
| R_AVR_7_PCREL |
| R_AVR_13_PCREL |
| R_AVR_16 |
| R_AVR_16_PM |
| R_AVR_LO8_LDI |
| R_AVR_HI8_LDI |
| R_AVR_HH8_LDI |
| R_AVR_LO8_LDI_NEG |
| R_AVR_HI8_LDI_NEG |
| R_AVR_HH8_LDI_NEG |
| R_AVR_LO8_LDI_PM |
| R_AVR_HI8_LDI_PM |
| R_AVR_HH8_LDI_PM |
| R_AVR_LO8_LDI_PM_NEG |
| R_AVR_HI8_LDI_PM_NEG |
| R_AVR_HH8_LDI_PM_NEG |
| R_AVR_CALL |

Tabelle 2.2: Übersicht über die Relocationtypen des AVR-GCC-Compilers

nen zu den Symbolen des Programms enthalten, eine Relocationtabelle besitzt es jedoch nicht mehr.

### 2.4.3 Linker

Der Linker realisiert, wie in Abbildung 2.8 gezeigt, den letzten Schritt des Kompiliervorgangs, in dem die einzelnen Objektdateien, aus denen ein Programm aufgebaut ist, zum eigentlichen Gesamtprogramm zusammengefügt werden. Zusätzlich zu den Objektdateien integriert der Linker dabei noch benötigte Systembibliotheken und stellt die Verbindungen zwischen den einzelnen Teilen her. Das vom Linker erstellte Gesamtprogramm kann anschließend auf der Zielplattform ausgeführt werden. Einfach ausgedrückt besteht die Hauptaufgabe des Linkers im Rahmen dieses Prozesses darin, abstrakte Bezeichner in konkrete Adressen zu übersetzen [Lev99].

Die einzelnen Aufgaben eines Linkers sind von den Eigenschaften des Compilers und des Zielsystems abhängig. Im Folgenden sollen einige wichtige Aspekte der Aufgaben des Linkers im AVR-GCC-Paket näher beleuchtet werden:

#### Zusammenfügen der Codesegmente

Der Linker bekommt als Eingabe eine Reihe von Objektdateien sowie die Pfade verschiedener Systembibliotheken. Er wählt die benötigten Codeabschnitte der Systembibliotheken aus und fügt diese mit dem Programmcode der Objektdateien – wie bei der

Erklärung des Programmspeichers im vorigen Abschnitt beschrieben – zu einem Gesamtprogramm zusammen. Beim AVR-GCC-Linker lässt sich dabei die Reihenfolge der Codeabschnitte der einzelnen Objektdateien im Gesamtprogramm direkt beim Aufruf des Linkers beeinflussen: Der Linker ordnet den Programmcode in der Reihenfolge an, in der die Objektdateien dem Linker übergeben werden.

### Erstellen des Layouts im Datenspeicher

Neben dem Zusammenstellen der Programmdaten muss der Linker auch das Layout der Symbole im Datenspeicher berechnen. Hierfür muss er zunächst den *.data*-Abschnitt des Gesamtprogramms generieren, indem er die *.data*-Abschnitte der einzelnen Objektdateien aneinanderhängt. Auch beim Festlegen der Reihenfolge der *.data*-Abschnitte hält sich der Linker an die Reihenfolge, in der ihm die Objektdateien übergeben werden. Im Anschluss an den *.data*-Abschnitt platziert der Linker die *.bss*-Symbole in der Reihenfolge, in der er sie erstmals einliest.

### Berechnen der Symboladressen

Sobald das Layout des Programms sowohl im Programmspeicher als auch im Datenspeicher feststeht, können die Adressen der Funktions- und der Datensymbole bestimmt werden. Für die Funktionssymbole ergibt sich die Zieladresse aus der Summe der Startadresse des jeweiligen Codeabschnitts innerhalb des Gesamtprogramms und dem lokalen Offset des Symbols. Die Adressen der Objektsymbole ergeben sich direkt aus ihrer Position innerhalb des Datenspeichers.

### Auflösen der Referenzen

Wenn der Linker die Zieladressen der Symbole ermittelt hat, kann er die Relocationtabellen der Objektdateien durchgehen und die darin enthaltenen Relocationeinträge auflösen. Neben der Zieladresse des Symbols verwendet der Linker zur Berechnung noch eventuell vorhandene lokale Offsets des Relocationeintrags.

Wie der Linker die Zieladresse einer Relocation berechnet, hängt vom Typ des Relocationeintrags ab. Dies soll anhand eines Beispiels veranschaulicht werden: Abbildung 2.13 zeigt vier Einträge einer Relocationtabelle. Dem besseren Verständnis halber wurden hinter den Einträgen in Klammern die jeweiligen Adressen der Symbole eingetragen. Unterhalb des Ausschnitts der Relocationtabelle ist der betroffene Bereich des Programmcodes als Hexcode einmal vor und einmal nach der Auflösung der Relocationeinträge dargestellt.

Der einfachere Fall liegt bei der Bearbeitung der zwei AVR_CALL-Relocations vor. Zunächst muss man jedoch feststellen, dass die Quelladresse der AVR_CALL-Relocations

```
00000152 R_AVR_LO8_LDI     AMStandard$counter (0x026b)
00000154 R_AVR_HI8_LDI     AMStandard$counter (0x026b)
00000156 R_AVR_CALL        TOS_post (0x0492)
0000015c R_AVR_CALL        OscilloscopeM$ADC$dataReady (0x09f2)


0150 8fbf80e0 90e00e94 0000c801 0e940000

wird zu:

0150 8fbf8be6 92e00e94 4902c801 0e94f904
```

Abbildung 2.13: Auflösen der Relocationeinträge im Programmcode

die Adresse der jeweiligen CALL-Anweisung angibt. Um das Sprungziel für diese CALL-Anweisungen zu setzen, muss der Wert der Relocation zwei Byte weiter hinten eingetragen werden – im Beispiel also an den Adressen `0x0158` und `0x015e`. Für keines der beiden Zielsymbole ist ein Offset definiert, so dass die Byteadressen der Symbole die Zieladressen der Relocations darstellen. Da im Programmspeicher nur gerade Byteadressen adressiert werden können, muss die Byteadresse vor dem Eintrag noch halbiert werden. Aus der Byteadresse `0x0492` wird so `0x0249`, aus `0x09f2` wird `0x04f9`. In „Little Endian"-Schreibweise ergibt sich der Eintrag `4902` an der Adresse `0x0158` für die erste AVR_CALL-Relocation und der Eintrag `f904` an der Adresse `0x015e` für die zweite AVR_CALL-Relocation.

Die beiden anderen Relocationeinträge beziehen sich auf die gleiche Zieladresse, der eine Eintrag verarbeitet das untere Byte (LO), der andere das obere Byte (HI). Die Zieladresse der Relocation entspricht der Byteadresse des Symbols `AMStandard$counter`. In diesem Fall ergibt sich der Wert `0x026b`. Da ein Verweis auf den Datenspeicher vorliegt, kann dieser Adresswert ohne weitere Berechnungen übernommen werden. Das untere Byte (`0x6b`) wird an die Adresse `0x0152` geschrieben, das obere Byte (`0x02`) an die Adresse `0x0154`. Bei Betrachtung des Ergebnisses der Relocations erkennt man, dass AVR_LO8_LDI und AVR_HI8_LDI ein kompliziertes Layout erzeugen, welches das zu schreibende Byte der Relocation jeweils auf zwei aufeinander folgende Bytes im Programmcode verteilt.

### 2.4.4 Übertragen und Installieren des Programms

Der AVR-GCC-Compiler erstellt ausführbare Dateien im „Executable and Linking Format (ELF)" [Too95]. Darin sind noch die Symboltabelle des Programms sowie eventuelle Debugginformationen enthalten. Vor dem Übertragen des Programms an den Sensorknoten wird die ELF-Datei in das Motorola S-Record-Format [Motb] konvertiert. Bei diesem Schritt werden auch die Symbol- und Debugginformationen entfernt, da diese

für die Ausführung des Programms auf dem Sensorknoten nicht benötigt werden. An den Sensorknoten werden somit die reinen Binärdaten des Programms in kodierter Form übertragen.

Das Motorola S-Record-Format kodiert Binärdaten in lesbarer Form (unter Verwendung des ASCII-Zeichensatzes) und fügt einen Prüfsummenmechanismus hinzu. Das Format ist eines der Standardformate für die Übertragung von Programmdaten an eingebettete Systeme.

Die Übertragung der Programmdatei geschieht mit Hilfe des Programms *uisp* [UIS]. *uisp* liest Programmdateien im S-Record-Format (und in verschiedenen weiteren Formaten) ein und überträgt diese über den seriellen oder den parallelen Port des Computers an ein so genanntes „Programming Board“ [Cro03a]. Der Mica2-Sensorknoten muss für die Programmierung über seine Standardschnittstelle direkt mit dem „Programming Board“ verbunden sein, eine Übertragung per Funk ist nicht möglich.

# 3 Verwandte Arbeiten

In diesem Kapitel werden verschiedene verwandte Ansätze beschrieben, die sich mit unterschiedlichen Möglichkeiten beschäftigen, den kompletten Programmcode oder Teile einer Sensornetzanwendung während des Systembetriebs dynamisch auszutauschen. Die vorgestellten Arbeiten lassen sich in vier Gruppen einteilen: Arbeiten, die einen **Komplettaustausch** des Programmcodes der Sensorknoten implementieren, **Differenzansätze**, die nur die Änderungen zur Vorversion an die Sensorknoten übertragen, **„Virtual Machine"-Ansätze**, die die Flexibilität für den Codeaustausch durch die Verwendung einer „Virtual Machine" erreichen, sowie **Middleware- und Betriebssystemansätze**, die die Fähigkeit zum Codeaustausch in einer Zwischenschicht zwischen Anwendung und Betriebssystem oder auch direkt im Betriebssystem implementieren.

## 3.1 Komplettaustausch

### 3.1.1 Xnp

Bei Xnp [Cro03b][JKB03] handelt es sich um einen sehr einfachen Ansatz für die Lösung des Codeaustausch-Problems, der in der Version 1.1 von TinyOS erstmals mitgeliefert wurde. Xnp überträgt grundsätzlich den vollständigen Programmcode ohne zu überprüfen, welche Programmteile sich geändert haben.

Die Xnp-Implementierung besteht aus einem **Java-Programm**, mit dessen Hilfe das Umprogrammieren der Sensorknoten auf der Basisstation angestoßen werden kann, sowie einer **Xnp-Komponente**, die als Teil der Anwendung auf den Sensorknoten installiert sein muss. Außerdem muss auf den Sensorknoten noch ein **Bootloader** installiert sein, der den neuen Programmcode in den Programmspeicher überträgt.

Die Installation eines neuen Programms erfolgt in drei Phasen. In der **Übertragungsphase** sendet die Basisstation den Programmcode in einzelnen Paketen an einen Sensorknoten. Dort werden die Daten im externen Flash gespeichert. Anschließend kann der Sensorknoten in der **Rückfragephase** eventuell verloren gegangene Pakete bei der Basisstation erneut anfordern. Sobald die benötigten Daten komplett im Sensorknoten vorliegen, übergibt das Programm die Kontrolle an den Bootloader, der dann in der **Programmierungsphase** den Programmcode in den Programmspeicher des Prozessors überträgt.

Xnp beherrscht keine komplizierten Netzwerktopologien, die eine Übertragung über Zwischenschritte erforderlich machen. Da im verwendeten Übertragungsprotokoll das Weiterleiten von Nachrichten nicht vorgesehen ist, können nur Sensorknoten in der direkten Umgebung (der 1-Hop-Umgebung) der Basisstation umprogrammiert werden.

### 3.1.2 Deluge

Das Ziel von Deluge [HC04] ist die zuverlässige Verteilung ganzer Codeimages in Sensornetzen. Deluge baut hierfür direkt auf dem Codeverbreitungsalgorithmus von Trickle [LPCS04] auf und erweitert den Algorithmus, um auch die Verteilung großer Codeblöcke zu unterstützen.

#### Codeverbreitung mit Trickle

Trickle [LPCS04] ist ein Algorithmus zur Verbreitung von Codeupdates in Sensornetzen. Die Entwickler gehen davon aus, dass die Verbreitung von Updates ein kontinuierlicher Prozess sein muss, da die Menge der durch Update-Nachrichten erreichbaren Knoten innerhalb eines Sensornetzes kontinuierlich schwankt. Dementsprechend müssen die Knoten eines Sensornetzes periodisch Metadaten austauschen, um feststellen zu können, welche Codeteile von anderen Knoten benötigt werden und an diese versendet werden müssen. Dem Algorithmus von Trickle liegt die Erkenntnis zu Grunde, dass der Aufwand, diese Metadaten periodisch zu versenden, schnell den Aufwand zum Versand der eigentlichen Codepakete übersteigt. Um möglichst wenig Metadaten-Nachrichten versenden zu müssen, verwendet Trickle ein Verfahren, das „polite gossiping“ genannt wird.

Das Grundprinzip von **„polite gossiping“** ist einfach: Jeder Knoten unterteilt die Zeit in gleichmäßig große Zeitintervalle und entscheidet zu einem zufälligen Zeitpunkt $t$ eines jeden Intervalls, ob er seine Code-Metadaten per Broadcast versendet oder nicht. Hat der Knoten in diesem Intervall bis zum Zeitpunkt $t$ bereits Broadcasts der gleichen Metadaten von anderen Knoten empfangen, so verzichtet er auf das Senden seiner Metadaten, um Wiederholungen zu vermeiden. Falls der Sensorknoten eine ältere Version der Metadaten empfangen hat, so versendet er die benötigten Codeabschnitte, um die Codebasis seiner Nachbarn auf den aktuellen Stand zu bringen. Hat der Sensorknoten dagegen Metadaten empfangen, die aktueller sind als seine eigenen Metadaten, so versendet er seine Metadaten, um anderen Knoten zu signalisieren, dass er ein Codeupdate benötigt.

Durch die Verwendung des im vorigen Abschnitt skizzierten „polite gossiping“-Algorithmus gelingt es Trickle einerseits, Code Updates schnell in Sensornetzen zu verteilen. Andererseits verursacht „polite gossiping“ nur wenig Aufwand für die Übertragung der Metadaten solange sich das Netzwerk in einem stabilen Zustand befindet. Da die Sensorknoten den Versand von Metadaten davon abhängig machen, ob zuvor bereits von anderen Knoten gleiche Daten versendet wurden, skaliert Trickle auch für große Knotendichten.

Außer in Deluge wird Trickle auch innerhalb des Codeupdate-Mechanismus der TinyOS Virtual Machine Maté [LC02] (siehe auch Abschnitt 3.3.1) verwendet. Maté verschickt mit Hilfe von Trickle regelmäßig eine Zusammenfassung der Versionsnummern seiner Routinen. Stellt ein Knoten fest, dass ein Nachbarknoten ältere Versionen einzelner Routinen besitzt, so verschickt Maté den Code dieser Routinen dreimal per Broadcast.

### Codeupdates mit Deluge

Deluge zerteilt die zu übertragenden Codeimages der Sensorknoten in **Pages** fester Größe. Diese Pages bilden die Grundlage für die Übertragung der Daten zwischen den einzelnen Sensorknoten. Jeder Knoten speichert die Versionsnummer seines aktuellen Codeimages und gleichzeitig einen Vektor mit Altersinformationen zu allen Pages, aus denen das Codeimage besteht. Die Altersinformation einer Page gibt an, in welcher Version des Codeimages sie zuletzt geändert wurde. Zusammen bilden die Versionsnummer des Codeimages und der Vektor der Altersinformationen das **Objektprofil des Sensorknotens**. Mit Hilfe des Trickle-Algorithmus informieren sich die Knoten gegenseitig über ihre aktuellen Objektprofile. Empfängt nun ein Knoten ein Objektprofil einer neueren Version, so kann er anhand der Altersinformationen der Pages bestimmen, welche Pages er neu empfangen muss, um sein Codeimage auf den Stand der neuen Version zu bringen.

Beim Versand eines Updates sendet beziehungsweise empfängt ein Knoten immer nur eine Page gleichzeitig, aufgeteilt in $N$ einzelne Pakete. Dieses Vorgehen vereinfacht die Zuordnung der Pakete, da immer nur eine feste Anzahl von $N$ Paketen zwischen den Knoten unterwegs sein kann.

Eine wichtiges Merkmal von Deluge ist das so genannte **„spatial multiplexing“**: Ein Knoten nimmt eine vollständig empfangene Page sofort in sein Objektprofil auf, selbst wenn noch nicht alle anderen Pages der neuen Codeimage-Version empfangen wurden. Auf diese Weise können sich neue Pages sehr schnell im Netzwerk verteilen, da mit jeder Übertragung einer Page sofort die Anzahl möglicher Sender steigt. Damit es zu möglichst wenig Konflikten und Interferenzen bei der Übertragung unterschiedlicher Pages mit *„spatial multiplexing“* kommt, verwendet Deluge mehrere Regeln. Beispielsweise fordert jeder Knoten die von ihm benötigten Pages grundsätzlich nur in aufsteigender Reihenfolge der Page-Nummern an, daneben hat der Transfer einer Page niedriger Nummer Priorität gegenüber dem Transfer von Pages mit höheren Nummern.

Nachdem ein Knoten alle benötigten Pages empfangen hat, kann er die neue Version des Codeimages installieren. Zu diesem Zweck wird ein Systemreset durchgeführt. Bei jedem Reset eines mit Deluge ausgestatteten Sensorknoten wird ein **Bootloader** aufgerufen. Falls kein neues Codeimage vorliegt, ist die einzige Aktion dieses Bootloaders die normale Anwendung des Sensorknotens zu starten. Enthält der externe Speicher des Sensorknotens dagegen Daten für ein neues Codeimage, so kopiert der Bootloader erst diese Daten in den Programmspeicher und startet anschließend die neue Anwendung des Sensorknotens.

Ein entscheidender Unterschied zwischen Xnp und Deluge besteht darin, dass Deluge in der Lage ist, Code-Images auch über mehrere Zwischenknoten zu verteilen. Da Deluge inzwischen standardmäßig Bestandteil des TinyOS-Systems ist, kristallisiert es sich als Nachfolger für Xnp als Standardverfahren für Codeupdates in TinyOS-Sensornetzen heraus.

## 3.2 Differenzansätze

Unter der Annahme, dass sich bei einem großen Teil der Updates der Software von Sensorknoten lediglich kleine Teile der Codebasis ändern, erscheinen Verfahren, die auf dem Austausch kompletter Pages oder gar des kompletten Codeimages basieren, sehr ineffizient. Verschiedene Forschergruppen verfolgen aus diesem Grund den Ansatz, anstatt des kompletten Codes nur die Änderungen zur letzten installierten Version an die Sensorknoten zu übertragen und so bei kleinen Änderungen mit deutlich weniger übertragenen Daten auszukommen.

### 3.2.1 Algorithmus von Reijers und Langendoen

Die Grundidee des Verfahrens von Reijers und Langendoen [RL03] ist, Ähnlichkeiten zwischen der alten und einer neuen Version der Software von Sensorknoten auszunutzen und lediglich die Unterschiede als Update zu übertragen. Zur Berechnung eines **Änderungsprotokolls** zwischen der alten und der neuen Softwareversion verwendet es einen Algorithmus, der vom Ansatz her dem diff-Algorithmus von Unix gleicht.

Das Änderungsprotokoll besteht aus einer Folge von Anweisungen, die spezifizieren, wie die alte Softwareversion in die neue zu übertragen ist. Die zwei grundlegenden Operationen sind hierbei *copy* und *insert*: *copy* kopiert eine Folge von Bytes von einer spezifizierten Position des alten Codeimages in das neue Codeimage. Die Annahme dahinter ist, dass viele Teile des bisherigen Codes an eine andere Position verschoben wieder auftauchen, zum Beispiel weil neuer Anwendungscode davor eingefügt wurde. *insert* fügt eine Folge von Bytes neu in das Codeimage ein.

Da zu erwarten ist, dass einige Teile des Codes in der neuen Version ähnlich, aber nicht identisch wieder auftauchen, gibt es zusätzlich die Anweisung *repair*. *repair* arbeitet auf einer *copy*-Anweisung und ersetzt einzelne Bytes an spezifizierten Offsets. Ändern sich in einer langen Codesequenz nur wenige Bytes, so benötigt eine einzelne *copy*-Anweisung zusammen mit einer *repair*-Anweisung weniger Platz im Änderungsprotokoll als eine Folge von *copy*- und *insert*-Anweisungen.

Reijers und Langendoen mussten feststellen, dass bei der Verwendung von *copy*, *insert* und *repair* in den Änderungsprotokollen selbst bei vergleichsweise kleinen Änderungen sehr große Änderungsprotokolle nötig sind. Der Grund hierfür liegt in den Auswirkungen, die das Verschieben von Teilen des Codes oder der Nutzdaten hat: Wird in eine

neue Version Code eingefügt, so verschiebt sich aller dahinter liegender Code entsprechend nach hinten. Problematisch ist dies, da nun alle Referenzen auf den verschobenen Code (gleiches gilt für verschobene Daten) angepasst werden müssen. Zur Lösung dieses Problems wurde die so genannte *patch list*-Anweisung eingeführt. Sie passt alle Adressen im Code, die zwischen einer spezifizierten Start- und Zieladresse liegen, um einen bestimmten Offset an. Damit werden beispielsweise alle CALL-Referenzen, die auf eine Zieladresse hinter dem eingefügten Code zeigen, um einen Offset erhöht, welcher der Größe des eingefügten Codes entspricht. Da jedoch nicht überprüft wird, ob es sich bei den angepassten Daten tatsächlich um Adressen und nicht etwa um konstante Daten handelt, müssen einzelne durch eine *patch list*-Anweisungen angepasste Daten im Anschluss wieder durch *repair*-Anweisungen zurückgesetzt werden.

Als Alternative oder Ergänzung zur Verwendung der *patch list*-Anweisung schlagen Reijers und Langendoen vor, zwischen Funktionen Pufferplatz einzufügen, damit die Funktionen wachsen oder schrumpfen können ohne dabei anderen Code zu verschieben. Sie weisen jedoch darauf hin, dass dieser Lösung durch den beschränkten Speicherplatz der Sensorknoten enge Grenzen gesetzt sind.

Reijers und Langendoen haben das beschriebene Verfahren für Sensorknoten der EYES-Plattform [EYE] implementiert.

### 3.2.2 Implementierung von Ye, Yamamoto und Stathopolous

Ye, Yamamoto und Stathopolous [YYS03] haben den Algorithmus von Reijers und Langendoen für die Mica2-Plattform implementiert und um einige Funktionen erweitert, um die Generierung der Änderungsskripte weiter zu optimieren.

Sie verwenden in ihren Änderungsprotokollen ebenfalls *copy*, *insert*, *repair* und die *patch list*-Anweisung, sie fügen allerdings einige Varianten der *copy*-Anweisung hinzu, die in bestimmten Fällen weniger Platz im Änderungsprotokoll benötigen.

Um ein besseres Matching zwischen dem Code der Originaldatei und dem Code der neuen Version zu erreichen, zerteilen sie beide Dateien in die Code- und Datenbereiche der einzelnen Objektdateien und wenden den Diff-Algorithmus getrennt für die einzelnen Teile an. Anschließend fügen sie die generierten Änderungsprotokolle zusammen und optimieren sie mit einem Postprocessor, der beispielsweise angrenzende *copy*-Anweisungen miteinander verbindet.

Ein wichtiger Unterschied zum Vorgehen von Reijers und Langendoen ist, dass sie für jeden Befehl eines Änderungsprotokolls abschätzen, wie viele Anweisungen für seine Verarbeitung auf dem Prozessor des Motes notwendig sind. Basierend auf diesen Daten versuchen sie bei der Erstellung des Änderungsprotokolls sowohl die Größe des Änderungsprotokolls als auch die Anzahl der benötigten Anweisungen zur Verarbeitung des Protokolls zu optimieren. Diese Abschätzung lohnt sich insbesondere für die *patch list*-

Anweisungen, da ihre Ausführung viele Vergleichsoperationen benötigt und dementsprechend nur bei einer entsprechend großen Anzahl von Ersetzungen lohnenswert ist.

Die vorliegende Implementierung wurde für die Verwendung in MOAP [SHE03] entwickelt. MOAP („**M**ultihop **O**ver-the-**A**ir **P**rogramming“) ist ein Algorithmus für die Verteilung von Programmcode, der speziell für Mica2-Motes entwickelt wurde. In der Evaluation in Kapitel 7 dieser Ausarbeitung wird der Differenzansatz von Ye, Yamamoto und Stathopolous verwendet und dort als MOAP-Diff bezeichnet.

### 3.2.3 Ansatz von Jeong und Culler

Jeong und Culler [JC04][Jeo04] setzen in ihrer Implementierung eines Updatealgorithmus auf eine Variante des Rsync-Algorithmus [Tri99], der für die Synchronisierung binärer Daten über eine bidirektionale Kommunikationsverbindung geringer Bandbreite entwickelt wurde. Der größte Unterschied zum klassischen Rsync-Algorithmus besteht darin, dass Jeong und Culler einen Großteil der Berechnung auf die Seite des zentralen Rechners verlagern, um die Ressourcen der Sensorknoten zu schonen. Auch bei diesem Ansatz wird zunächst auf dem zentralen Rechner ein Änderungsprotokoll berechnet, an die Sensorknoten übertragen und dort wieder in ein komplettes Codeimage umgesetzt.

Der Rechner muss für die Berechnung des Änderungsprotokolls die Binärdaten in $n$ Blöcke fester Größe $B$ zerteilen. Für jeden dieser $n$ Blöcke berechnet er einen Hashwert und trägt diesen in eine Tabelle ein. Anschließend durchläuft der Algorithmus die zu installierenden Binärdaten Byte um Byte. An jeder Byteposition berechnet er den Hashwert des hier beginnenden $B$ Byte großen Blocks. Ist ein Block mit diesem Hashwert in der Tabelle eingetragen, so muss er nicht übertragen werden, sondern kann aus dem bestehenden Programmcode übernommen werden.

Als großen Vorteil ihres Ansatzes geben die Autoren an, dass die Implementierung unabhängig von der Hardware der Sensorknoten eingesetzt werden kann, da kein Wissen über die Struktur des analysierten Programms verwendet wird. Darüber hinaus verzichtet der Ansatz auf komplexe Operationen wie das Reparieren von geänderten Adressreferenzen, die zwar die Größe des zu übertragenden Codes reduzieren, gleichzeitig aber auch mehr teure Zugriffe auf den externen Flashspeicher erfordern. Die Evaluation des Algorithmus [Jeo04] zeigt aber auch, dass dieses Vorgehen nur bei vergleichsweise geringen Änderungen des Programmcodes erfolgreich ist. Anscheinend finden sich bei größeren Änderungen nur noch wenige Blöcke der Größe $B$, die komplett aus dem ursprünglichen Programmcode übernommen werden können.

### 3.2.4 Inkrementelles Linken

Koshy und Pandey [KP05] gehen mit ihrem Differenzansatz über die reine Generierung von Änderungsprotokollen hinaus: Sie argumentieren, dass ein Ignorieren der Struktur

einer Anwendung bei der Erstellung der Änderungsprotokolle in vielen Fällen zu nicht optimalen Ergebnissen führt. So kann schon das Übertragen einer relativ kleinen Änderung am Sourcecode eines Programms ein sehr großes Änderungsprotokoll erfordern, wenn eine Änderung zahlreiche Codeverschiebungen und daraus resultierend viele Anpassungen bei den Sprungzielen verursacht. Um solchen Effekten vorzubeugen, setzen Koshy und Pandey bereits beim Prozess des Linkens der Objektdateien zur ausführbaren Datei an. Sie haben eine angepasste Version des AVR-GCC-Linkers erstellt, die den Code bereits beim ersten Linken auf mögliche zukünftige Änderungen vorbereitet und beim Linken eines geänderten Programms versucht das ursprüngliche Layout möglichst unverändert beizubehalten. Ein wichtiges Konzept ist dabei der so genannte **Slop Space**, der als Pufferbereich zwischen die Codeabschnitte der einzelnen Funktionen eingefügt wird. Ändert sich später durch eine Änderung die Größe einer Funktion, so kann sie an ihrer Position bleiben, solange sie nicht über den Slop Space am Ende der Funktion hinauswächst.

Der Standardlinker des AVR-GCC-Compilerpakets behandelt geänderte oder neue Funktionen genau wie alle anderen Funktionen, da er keine Vergleiche zwischen der alten und der neuen Version der Objektdateien durchführt. Dementsprechend wird neuer Code dieser Funktionen in vielen Fällen inmitten der ausführbaren Datei platziert. Um dies zu vermeiden, enthält die angepasste Linkerimplementierung von Koshy und Pandey einen **Memory Manager**, der den Funktionen ihren Speicherplatz zuweist. Bei Änderungen untersucht der Memory Manager, ob innerhalb des Codeimages ausreichend freier Platz zur Speicherung der geänderten oder neuen Funktion vorhanden ist; ansonsten platziert er sie am Ende des Programmcodes. Der alte Code einer verschobenen Funktion wird nicht sofort überschrieben, es werden lediglich alle Referenzen auf die Funktion zur neuen Position umgeleitet. Der freigewordene Speicherplatz steht dann bei späteren Updates zur Verfügung und kann durch Funktionen der passenden Größe aufgefüllt werden. Da der Linker beim Erstellen einer neuen Programmversion die alten Programmdaten in die Betrachtungen einbezieht und versucht, das bestehende Layout weitestgehend zu erhalten, sprechen die Autoren von einem **inkrementellen Linker**.

Wie in den zuvor vorgestellten Ansätzen wird nach dem Linken durch den Vergleich des originalen und des geänderten Codeimages mit Hilfe eines Differenzalgorithmus ein Änderungsprotokoll generiert und an die Sensorknoten übertragen.

Der große Vorteil des inkrementellen Linkens besteht darin, dass bei einem Update einer oder mehrerer Funktionen das Speicherlayout der von den Änderungen nicht betroffenen Funktionen sich nicht ändert, da keine Funktionen an einen anderen Speicherplatz verschoben werden müssen. Dementsprechend sind auch keine Anpassungen der Sprungadressen notwendig, was ein kleineres Änderungsprotokoll und weniger Aufwand bei der Bearbeitung durch die Sensorknoten ermöglicht.

Ein Nachteil des Verfahrens liegt im großen Speicherplatzverbrauch des Programmcodes, der durch den Slop Space zwischen den einzelnen Funktionen verursacht wird. Im Laufe mehrerer Updatevorgänge können der Programm- und Datenspeicher zudem stark fragmentiert werden, so dass ein Verschieben einzelner Programmteile oder eine Neuinstallation des gesamten Programms notwendig wird.

## 3.3 „Virtual Machine“-Ansätze

### 3.3.1 Maté

Die Entwickler von Maté [LC02] verfolgen einen „Virtual Machine“-Ansatz, um den einfachen und effizienten Austausch von Programmcode in TinyOS-Sensornetzen zu ermöglichen. Anstatt Anwendungen direkt als Binärcode auf den Sensorknoten zu installieren, führt jeder Sensorknoten einen Bytecode-Interpreter aus, der die eigentlichen Anwendungen in einem speziellen Bytecode-Format aus dem Speicher einliest, die Maté-Anweisungen interpretiert und in entsprechende TinyOS-Operationen umsetzt.

Ein Installieren von neuem Programmcode auf Maschinenebene ist durch die Verwendung des „Virtual Machine“-Ansatzes nicht mehr notwendig, da kein Binärcode im Programmspeicher ersetzt werden muss. Aus Sicht des Prozessors handelt es sich bei den Anwendungsprogrammen nur um Daten, die vom „eigentlichen“ Programm eingelesen und bearbeitet werden.

Der Bytecode jeder Anwendung ist in mehrere so genannte **Codekapseln** aufgeteilt. Jede dieser Kapsel besteht aus einem Bezeichner, einer Versionsnummer sowie bis zu 24 Anweisungen. Da jede Anweisung in einem Byte kodiert ist, passt die Kapsel noch in ein einzelnes TinyOS-Datenpaket und kann in einer einzelnen Nachricht über das Netzwerk an andere Sensorknoten übertragen werden.

Die Codekapseln stellen in Maté die Einheiten dar, in denen Programmcode empfangen und installiert werden kann. Empfängt ein Knoten eine Codekapsel, so überprüft er anhand des Bezeichners und der Versionsnummer, ob die empfangene Kapsel eine bereits vorhandene Kapsel ersetzt und tauscht sie dann gegebenenfalls aus. Sollte sich eine Änderung einer Anwendung auf mehr als eine Codekapsel auswirken, so muss sichergestellt werden, dass ein Sensorknoten alle geänderten Codekapseln erhält und parallel installiert.

Das Sprachkonzept von Maté enthält einfache Befehle, um neue Kapseln im Sensornetz zu verbreiten. Nutzen die Sensorknoten diesen Mechanismus, so genügt es, die neue Version einer Kapsel auf einem der Sensorknoten zu installieren. Dieser kann die Kapsel dann an seine Nachbarn weiterleiten. Schritt für Schritt verbreitet sich der Programmcode auf diese Weise im ganzen Netz. Die Entwickler sprechen in diesem Zusammenhang von **„Code Infection“**.

Ein großer Vorteil der Verwendung von Bytecode anstelle von nativem Binärcode liegt in der geringeren Größe der zu übertragenden Programme. Mit Maté-Bytecode lassen sich die Programme auf einer sehr hohen Sprachebene ausdrücken, wodurch sich auch relativ komplexe TinyOS-Programme mit nur wenigen Maté-Bytecode-Anweisungen ausdrücken lassen. Da jede Bytecode-Anweisung nur ein Byte umfasst, lassen sich viele relativ komplexe Programmmodule in einem einzelnen TinyOS-Datenpaket an die Sensorknoten übertragen.

Ein mögliches Problem bei der Verwendung von Maté stellt dagegen die eingeschränkte Ausdrucksfähigkeit des verwendeten Befehlssatzes dar. Mit den Bytecode-Befehlen von Maté lassen sich nicht alle Anwendungssemantiken formulieren, die bei der direkten Programmierung des Sensorknotens – zum Beispiel mit Hilfe von nesC – möglich wären. Die Entwickler nennen als ein problematisches Beispiel die Programmierung spezieller mathematischer Funktionen. Um möglichst viele Anwendungen in Maté realisieren zu können, wurde der Befehlssatz für typische Aufgaben in Sensornetzanwendungen optimiert. Zusätzlich können acht benutzerdefinierte Funktionen hinzugefügt werden, mit denen sich auch spezielle Funktionen realisieren lassen. Allerdings muss die Semantik dieser Funktionen schon vor der Verteilung der Sensorknoten feststehen, da die Funktionen in die Implementierung der Maté-Virtual Machine integriert werden müssen.

Der Flexibilität von Maté bezüglich der Installation neuer Codesegmente stehen die Kosten für die Verwendung der virtuellen Bytecode-Anweisungen gegenüber. Die Entwickler haben einen bis zu 33,5-fachen Overhead in der Anzahl der benötigten Maschinenanweisungen bei einfachen Befehlen gegenüber direkten Binärcode-Implementierungen festgestellt. Dieser Overhead verursacht neben einem größeren Zeitbedarf auch einen deutlich größeren Energieverbrauch bei der Bearbeitung der Befehle. Bei sehr häufigen Updates des Programmcodes kann sich die Verwendung von Maté trotzdem rentieren, da die geringe Größe der Codekapseln Einsparungen bei der Übertragung ermöglicht. Bei unregelmäßigeren Codeupdates überwiegen die Vorteile einer effizienteren Implementierung durch Binärcode.

### 3.3.2 SensorWare

Die Entwickler von SensorWare [BHS03] verwenden ebenfalls einen „Virtual Machine“-Ansatz, gehen im Vergleich zu Maté aber von einer deutlich mächtigeren Sensorplattform aus, die mehr Programm- und Hauptspeicher bietet. SensorWare benötigt alleine bereits 179 Kilobyte Programmspeicher und passt dadurch nicht zu den in dieser Arbeit (und von Maté) verwendeten Mica2-Sensorknoten, die nur 128 Kilobyte Programmspeicher zur Verfügung stellen. Im Gegenzug ist das Sprachkonzept von SensorWare mächtiger als das von Maté und kann auch komplexe Programminhalte darstellen.

Auf den einzelnen Sensorknoten läuft jeweils ein Betriebssystem, auf dem die SensorWare-Schicht aufsetzt. Neben mobilen Kontrollskripten („Control Scripts“), die von der SensorWare-Schicht ausgeführt werden, enthalten die Sensorknoten auch statische Anwendungen und statische Dienste, die allgemeine Aufgaben des Sensorknotens erledigen und generische Dienste bereitstellen.

Der Fokus von SensorWare liegt weniger auf der dynamischen Programmierung der einzelnen Sensorknoten, als vielmehr auf der Beantwortung komplexer Benutzeranfragen durch das Sensornetz. Analog zum Konzept der „Active Networks“ [TSS+97], in denen der Benutzer kleine Programme in die Knoten eines Computernetzes injizieren kann, schlagen die Autoren die Verwendung eines „Active Sensor“-Konzeptes vor. Zur Beant-

wortung komplexer Anfragen wird ein entsprechendes Kontrollskript in einen der Knoten des Sensornetzes injiziert. Teil der Ausführung des Kontrollskripts durch Sensorware ist die Verteilung des Skripts an weitere Sensorknoten, wobei diese Verteilung durch das Kontrollskript selbst gesteuert wird. Auf diese Weise verteilt sich die Anfrage (und damit das Skript) über weitere Knoten und sammelt so die zur Beantwortung der Anfrage benötigten Daten ein.

## 3.4 Middleware- und Betriebssystemansätze

### 3.4.1 Impala

Bei Impala [LM03] handelt es sich um einen Middleware-Ansatz für Sensornetze mit dem Ziel, Sensornetzanwendungen modular, adaptiv und austauschbar gestalten zu können. Die Autoren fügen zu diesem Zweck oberhalb des Betriebssystems der Sensorknoten eine Middleware-Schicht ein, die für die darauf aufbauenden Anwendungen die Rolle des Betriebssystems übernimmt, Ressourcen verwaltet und als Ereignisfilter dient. Die Middleware-Schicht erlaubt es zudem, die auf ihr aufbauenden Anwendungen dynamisch anzupassen oder auch auszutauschen.

Die Impala-Middleware besteht aus drei Hauptkomponenten: dem **Event Filter**, dem **Application Adapter** sowie dem **Application Updater**.

Der **Event Filter** nimmt Ereignisse vom Betriebssystem und der Hardware der Sensorknoten entgegen und verteilt sie an den *Application Adapter*, den *Application Updater* und die Anwendungen des Sensorknoten. Impala kennt fünf unterschiedliche Ereignistypen: *Timer-Ereignisse*, *Empfangs-* und *Sendeereignisse* für die Kommunikation über die Funkschnittstelle, *Daten-Ereignisse*, welche die Verfügbarkeit neuer Sensordaten anzeigen, sowie *Geräteereignisse*, die im Falle von Gerätefehlern ausgelöst werden. Die Anwendungen, der *Application Adapter* und der *Application Updater* bestehen jeweils aus einer Menge von Event-Handlern, die nach Aufrufen durch den *Event Filter* in Aktion treten.

Der **Application Adapter** passt die Eigenschaften der Anwendungen und der Anwendungsprotokolle in Reaktion auf bestimmte Systemereignisse an. Derartige Anpassungen können genutzt werden, um die Effizienz der Anwendung zu steigern oder um auf Hardwareausfälle zu reagieren. Der *Application Adapter* trifft seine Entscheidungen mit Hilfe so genannter „Adaptation Finite State Machines“, die die Bedingungen für Anpassungen der einzelnen Anwendungen definieren. Es ist dabei die Aufgabe des Anwendungsprogrammierers, für seine Anwendung eine entsprechende „Adaptation Finite State Machine“ bereitzustellen.

Der **Application Updater** ist für den Empfang und die Installation neuer Versionen der Anwendungen zuständig. Jede Anwendung ist dafür in Module aufgeteilt. Für jedes dieser Module wird eine Versionsnummer gepflegt. Die Sensorknoten tauschen regelmäßig

Versionsinformationen aus und fordern bei Bedarf aktuellere Module bei ihren Nachbarn an. Nachdem eine neue Anwendung empfangen wurde, führt der *Application Updater* zunächst einige Sicherheitsprüfungen des neuen Codes durch und führt anschließend einen Linker auf den einzelnen Modulen aus, um die neue Anwendung in das System zu integrieren.

Die Autoren gingen bei der Entwicklung von Impala von anderen Hardwarevoraussetzungen aus als sie in dieser Arbeit gegeben sind. Für eine prototypische Implementierung verwendeten sie „Pocket PC"-Rechner, auf denen jeweils Linux als Betriebssystem installiert war.

### 3.4.2 Contiki

Bei Contiki [DGV04] handelt es sich um ein Betriebssystem, das speziell für Anwendungen in Sensornetzen und ähnlich ressourcenbeschränkten Umgebungen entwickelt wurde. Für die vorliegende Arbeit ist Contiki interessant, da die Entwickler die Möglichkeit zum dynamischen Austausch einzelner Komponenten als eine der grundlegenden Eigenschaften derartiger Systeme einstufen und die Fähigkeit zum dynamischen Codeaustausch beim Design des Contiki-Systems dementsprechend bereits standardmäßig integriert haben. Im Folgenden wird nicht das komplette Contiki-System beschrieben (siehe hierzu die Veröffentlichungen auf der Internetseite des Projektes [Con]) sondern nur die Möglichkeiten des Codeaustauschs.

Das Contiki-System besteht aus einem **Systemkern** sowie mehreren **Anwendungs- und Dienstprozessen**. Während der Systemkern statisch auf den Sensorknoten installiert und nach dem Ausbringen der Sensorknoten nicht mehr modifiziert wird, kann der Programmcode der Anwendungs- und Dienstprozesse dynamisch geladen und jederzeit gegen andere Komponenten ausgetauscht werden.

Dienstprozesse bieten nach außen hin Funktionen (= einen Dienst) an, die von einem oder mehreren anderen Prozessen genutzt werden können. Diese Funktionen werden über das so genannte **Service Interface** angesprochen. Das Service Interface enthält neben der Versionsnummer der Dienstimplementierung eine Tabelle mit Zeigern auf die Implementierungen der einzelnen Funktionen.

Die verschiedenen installierten Dienste werden vom **Service Layer** verwaltet, der zum Systemkern des Contiki-Systems gehört. Der Service Layer überwacht die aktiven Dienste und bietet die Möglichkeit, nach bestimmten Diensten zu suchen.

Anwendungskomponenten, die einen Dienst verwenden, enthalten hierfür einen Service-Stub. Alle Referenzen des Anwendungscodes auf Funktionen des Dienstes werden mit diesem Stub verlinkt. Beim ersten Zugriff der Anwendung auf den Dienst sucht der Stub unter Zuhilfenahme des Service Layers den Dienstprozess, der die geforderten Funktionen implementiert. Abbildung 3.1 zeigt die dabei entstehende Verbindung der Funktionsaufrufe in einem Anwendungsprozess zu den Implementierungen der Funktionen im Service-

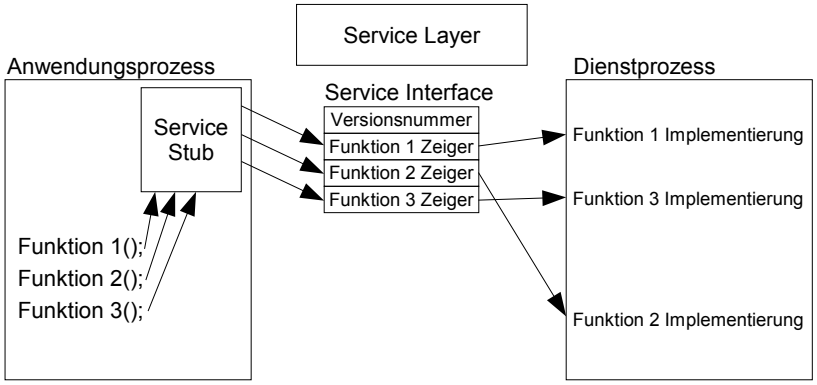


Abbildung 3.1: Verbindung der Funktionsaufrufe mit der Implementierung in Contiki (nach [DGV04])

prozess. Bei weiteren Zugriffen auf den Dienst kennt der Service Stub bereits die Adresse des zugehörigen Service Interfaces, so dass der Zugriff auf den Dienstprozess über das Interface ohne weitere Suche erfolgen kann.

Da der Stub der Anwendungskomponente die Funktionen des Dienstes über das Service Interface aufruft, kann ein Austausch des Dienstprozesses für die Anwendung völlig transparent ablaufen: Der Stub ruft weiterhin die Funktionen im Service Interface auf, nur verbergen sich dahinter nun eventuell Zeiger auf andere Funktionen.

Der Vorteil dieses Vorgehens besteht darin, dass sich der Austausch oder das Hinzufügen einzelner Dienste und Anwendungen sehr einfach gestaltet und nur einen geringen Aufwand auf den Sensorknoten erfordert. Ein Nachteil ist dagegen die niedrigere Effizienz im laufenden Betrieb, da jeder Aufruf einer Dienstfunktion über mehrere Zwischenschritte weitergeleitet wird.

Contiki wurde auf verschiedene Plattformen portiert, eine Implementierung existiert auch für die in der vorliegenden Arbeit verwendete AVR-Architektur von Atmel.

# 4 TinyCubus

Das `TinyCubus`-Projekt [MLM+04][MLM+05] an der Universität Stuttgart entwickelt ein System mit dem Ziel, die Anpassung von Anwendungen in Sensornetzen an wechselnde Aufgaben und an heterogene Umgebungsbedingungen durch entsprechende Systemsoftware zu unterstützen.

Der Zielsetzung des Projekts liegt die Erkenntnis zu Grunde, dass mit der fortschreitenden Entwicklung im Bereich der Sensornetze die Komplexität der entwickelten Systeme stetig ansteigt. Zum einen entsteht eine Vielzahl unterschiedlicher Hardware-Plattformen, die sich in ihren Eigenschaften und Fähigkeiten unterscheiden. Zum anderen entwickeln sich verschiedenste Anwendungsgebiete, deren jeweilige Anforderungen an die Sensorknoten und die Vernetzung der Sensorknoten stark differieren können. Dazu kommt, dass selbst einzelne Sensornetze inzwischen häufig heterogen aufgebaut sind; also aus Sensorknoten unterschiedlicher Fähigkeiten bestehen.

Die im letzten Absatz geschilderte steigende Komplexität in Sensornetzen begründet, dass Anpassungsfähigkeit zu einer Kernaufgabe in Sensornetzen wird und integraler Bestandteil der Anwendungsstruktur werden muss. Anstatt jedoch die Fähigkeit zur Anpassung an unterschiedliche Bedingungen und unterschiedliche Anforderungen wieder und wieder im Code verschiedenster Anwendungen zu implementieren, soll das `TinyCubus`-System diese Aufgaben übernehmen und der Anwendung eine flexible und adaptive Infrastruktur zur Verfügung stellen.

## 4.1 Konzept

`TinyCubus` besteht aus drei Teilen, dem **Tiny Data Management Framework**, dem **Tiny Cross-Layer Framework** und der **Tiny Configuration Engine**, die im Folgenden näher vorgestellt werden sollen.

### 4.1.1 Tiny Data Management Framework

Das Tiny Data Management Framework bietet eine Auswahl typischer Komponenten für das Management von Daten in Sensornetzanwendungen. Dazu gehören beispielsweise Komponenten für Datenaggregation, Datenreplikation, Caching und Prefetching von Daten. Für die verschiedenen Komponententypen dürften üblicherweise mehrere Implementierungen unterschiedlicher Eigenschaften existieren. Die Aufgabe des Tiny Data

Management Framework ist nun, aus der Menge der zur Verfügung stehenden Implementierungen die für das jeweilige System passende Implementierung auszuwählen.

Auf den Sensorknoten ist normalerweise jeweils nur eine Implementierung der benötigten Komponenten enthalten. Ist ein Austausch oder eine Erweiterung notwendig, so lädt das Tiny Data Management Framework die benötigte Komponente von anderen Sensorknoten oder nutzt als Quelle einen speziellen Knoten, der eine Komponentenbibliothek zur Verfügung stellt.

Die unterschiedlichen Algorithmen, die einen bestimmten Komponententyp realisieren, werden anhand dreier unterschiedlicher Kriterien klassifiziert: den **Anforderungen der Anwendung**, den **Systemparametern** sowie den **Optimierungsparametern**. Eine mögliche Anforderung der Anwendung ist beispielsweise eine hohe Zuverlässigkeit bei der Datenübertragung. Ein Beispiel für einen Systemparameter ist die Mobilität der Knoten im Sensornetz, als Optimierungsparameter kommt beispielsweise ein geringer Energieverbrauch oder eine geringe Latenz bei der Datenübertragung in Frage. Das Tiny Data Management Framework wählt aus den zur Verfügung stehenden Komponenten diejenige aus, die den Anforderungen der Anwendung, den Systemparametern sowie den Optimierungswünschen am besten gerecht wird. Ändern sich diese Parameter während des Systembetriebs, so kann das Tiny Data Management Framework entscheiden, die Komponente zu Gunsten einer besser geeigneten auszutauschen.

### 4.1.2 Tiny Cross-Layer Framework

In Sensornetzanwendungen ist eine strikte Trennung zwischen den einzelnen Schichten einer Anwendung nicht immer wünschenswert, da durch die Trennung Optimierungspotenziale verloren gehen. Ein Beispiel [MLM+05] hierfür sind die Nachbarschaftsinformationen des Sensorknotens, die unter Umständen sowohl in der Link Layer-Komponente als auch in einer der Anwendungskomponenten benötigt werden. Sind die Komponenten strikt voneinander getrennt, so müssen die Informationen in beiden Komponenten, also doppelt, gesammelt werden. Gibt man dagegen die Trennung zwischen den einzelnen Schichten der Anwendung auf, so verliert man zwangsläufig auch die Vorteile, die eine saubere Modularisierung bieten kann.

Um den Informationsaustausch zwischen den Schichten einer Anwendung zu ermöglichen, arbeitet das Tiny Cross-Layer Framework als Vermittler zwischen den Komponenten: Die Komponenten greifen nicht direkt auf Daten anderer Schichten zu, stattdessen bildet das Tiny Cross-Layer Framework ein so genanntes **State Repository**, in dem Cross-Layer-Informationen gespeichert werden. Auf diese Cross-Layer-Informationen können dann alle Schichten der Anwendung zugreifen.

Mit Hilfe einer speziellen Beschreibungssprache können die einzelnen Komponenten definieren, welche Cross-Layer-Informationen sie über das State Repository anderen Komponenten anbieten und welche Informationen sie aus dem State Repository benötigen. Durch diese Trennung zwischen dem Anbieter und dem Nutzer der Informationen er-

fordert der Austausch einer Komponente, die Cross-Layer-Informationen anbietet, nicht zwangsläufig auch den Austausch der Komponenten, die diese Informationen nutzen – vorausgesetzt die neue Komponente liefert dem State Repository die gleichen Informationen wie die alte Komponente.

Eine weitere Aufgabe des Tiny Cross-Layer Frameworks ist das Management der Callback-Funktionen zwischen den einzelnen Komponenten. Hier ist darauf zu achten, dass nach dem Austausch einer Komponente die Callback-Funktionen, die diese Komponente betreffen, auf die neue Komponente angepasst werden.

### 4.1.3 Tiny Configuration Engine

Innerhalb des `TinyCubus`-Systems ist die Tiny Configuration Engine für die Verteilung und die Installation von Programmcode in Sensornetzen zuständig. Sie soll den Austausch sowohl von Anwendungskomponenten als auch von Systemkomponenten ermöglichen. Unterstützt wird sie hierbei von einem **Topologiemanager**, der für die Konfiguration der Knoten im Sensornetz untereinander sorgt und jedem Sensorknoten eine **Rolle** innerhalb des Sensornetzes zuteilt.

#### Rollenzuordnungsalgorithmus

Rollen sind ein sehr wichtiges Konzept innerhalb von `TinyCubus`. Sie bestimmen, welche Aufgaben ein Sensorknoten in Zusammenarbeit mit seinen Nachbarn zu erfüllen hat. Da die meisten Sensornetze heterogen aufgebaut sind, haben Knoten unterschiedliche Aufgaben zu erfüllen und ein Algorithmus wird benötigt, der jedem Knoten seine Rolle zuweist. Steht die exakte Konfiguration nicht bereits im Voraus fest, so kann die Rollenzuteilung erst nach dem Ausbringen geschehen und muss in dynamisch veränderlichen Sensornetzen unter Umständen auch angepasst werden.

Ein Beispiel für unterschiedliche Rollen, die Sensorknoten zu erfüllen haben, ist die Aufteilung eines Sensornetzes in Datenquellen und Aggregationsknoten: Während die Datenquellen Sensordaten aufzeichnen und diese innerhalb des Sensornetzes verschicken, sind die Aggregationsknoten dafür zuständig, Sensordaten zu empfangen, diese anhand festgelegter Kriterien zu sammeln und zu aggregieren und sie anschließend in Richtung der Basisstation weiterzuleiten.

Der Rollenzuordnungsalgorithmus wird dezentral durch die einzelnen Sensorknoten ausgeführt. Soweit möglich wird die Rolle des Knoten auf Basis lokaler Informationen berechnet, ansonsten müssen die Knoten im Rahmen des Algorithmus mit ihren Nachbarn kommunizieren. Die Regeln, die der Algorithmus anwendet, sind in der so genannten **Rollenspezifikation** im Speicher aller Knoten abgelegt.

### Rollenbasierter Codeverteilungsalgorithmus

Der Codeverteilungsalgorithmus der Tiny Configuration Engine verwendet Informationen über die Rollen der Sensorknoten, um Codeupdates nur an die Menge der Knoten zu schicken, die das Update (aufgrund ihrer Rolle) tatsächlich benötigen.

Ein Knoten einer bestimmten Rolle $R$ beginnt die Codeverteilung, indem er die Daten per Broadcast an seine ***k*-Hop-Nachbarn** verschickt. Dies bedeutet, dass der Broadcast von dem Knoten ausgehend zunächst nur über maximal $k$ Sprünge weitergeleitet wird. Von der Menge der auf diese Weise erreichten Knoten verschicken nun die Knoten, die ebenfalls die Rolle $R$ besitzen, die Daten erneut an ihre $k$-Hop-Nachbarn weiter. Abhängig von der Topologie des Netzwerks und der Rollenzuordnung sind unterschiedliche Werte für $k$ notwendig, um alle Sensorknoten der Rolle $R$ zu erreichen. Je geringer $k$ gewählt werden kann, desto effizienter verhält sich natürlich der Codeverteilungsalgorithmus. Im Idealfall $k = 1$ muss jeder Sensorknoten die Daten nur an seine direkten Nachbarn weiterleiten.

Für jede Rolle $R$ in einem Sensornetz lässt sich analytisch bestimmen, welcher Wert für $k_R$ benötigt wird, um alle Sensorknoten der Rolle $R$ zu erreichen. Muss dagegen nicht garantiert werden, dass alle Knoten der Rolle $R$ erreicht werden, so kann ein niedrigeres $k$ gewählt werden, wodurch der Codeverteilungsalgorithmus effizienter arbeitet.

## 4.2 Diplomarbeit von Robert Sauter

Robert Sauter hat im Rahmen seiner Diplomarbeit [Sau05] den rollenbasierten Codeverteilungsalgorithmus als Teil der Tiny Configuration Engine implementiert. Die Implementierung enthält einen einfachen **Topologiemanager**, einen **Rollenzuordnungsalgorithmus** sowie den eigentlichen **rollenbasierten Codeverteilungsalgorithmus**.

Der **Topologiemanager** eines Sensorknoten speichert in der Implementierung nur Informationen zu den Rollen der Nachbarknoten und stellt basierend darauf verschiedene Informationen bereit, die der Rollenzuordnungsalgorithmus für die Berechnung der Rolle des Sensorknoten benötigt.

Die Implementierung des **Rollenzuordnungsalgorithmus** verarbeitet momentan die Spezifikationssprache für die Angabe der Rollenspezifikationen noch nicht, sondern ist auf hart kodierte Regeln angewiesen. Als großes Problem haben sich für die Implementierung Schleifenbildungen bei der Rollenzuordnung erwiesen: In bestimmten Fällen tauschen die Knoten des Sensornetzes in ständigem Wechsel ihre Rollen, ohne in einen stabilen Zustand zu konvertieren.

Der **Codeverteilungsalgorithmus** wurde durch die Implementierung wie im vorigen Abschnitt beschrieben umgesetzt. Für den Versand der Metadaten zwischen den Knoten setzt die Implementierung momentan Flooding ein.

Existierende Dateisystemlösungen für TinyOS, insbesondere das in vielen Anwendungen genutzte Matchbox-Dateisystem [Gay03b][Gay03a], sind für die Codeverteilungskomponente nicht geeignet. Da Matchbox vor allem für das Logging von Sensordaten entwickelt wurde, beherrscht es nur das sequentielle Schreiben und Lesen von Daten, nicht aber den wahlfreien Zugriff auf beliebige Adressen innerhalb einer Datei. Diese Fähigkeit ist für die Speicherung der per Funk empfangenen Programmpakete aber zwingend erforderlich, um auch Out-Of-Order-Datenpakete an der korrekten Position speichern zu können. Für die Speicherung der empfangenen Daten im externen Flash der Sensorknoten wurde aus diesem Grund ein eigenes Dateisystem entwickelt, das so genannte **„Random Access File System“** (RAFS). RAFS implementiert den wahlfreien Zugriff auf Dateien, indem es den verfügbaren Speicher im externen Flash in Blöcke fester Größe aufteilt und die von einer Datei belegten Blöcke in einer „File Used Block“-Liste speichert. Freie Blöcke werden dagegen in einer „Freelist“ für spätere Zugriffe vorgehalten.

## 4.3 Anforderungen an diese Diplomarbeit

Ein Ziel der Arbeit des `TinyCubus`-Projektes ist es, den Knoten eines Sensornetzes den selbstständigen Austausch einzelner Komponenten ihrer Anwendungs- oder Systemsoftware zu ermöglichen, ohne dass dies explizit von einer zentralen Einheit gesteuert werden muss. Die Aufgabe dieser Diplomarbeit besteht darin, den grundsätzlichen Mechanismus für die Updates einzelner Komponenten auf den Sensorknoten zu entwickeln.

Als Voraussetzung für den Austausch von Komponenten muss zunächst ein Mechanismus entworfen und umgesetzt werden, mit dessen Hilfe TinyOS-Anwendungen in mehrere getrennte Komponenten aufgeteilt werden können. Dabei bietet es sich an, die bestehende Komponentenarchitektur der nesC-Programmiersprache auszunutzen. Die Verbindungen zwischen den Komponenten einer Anwendung sollten über klar definierte Interfaces erfolgen, um `TinyCubus` im Laufe der weiteren Entwicklung den Austausch einer Komponente gegen eine alternative Implementierung zu vereinfachen.

Wie bereits im vorigen Abschnitt geschildert, wurde die Verteilung neuer Komponenten an die Sensorknoten bereits in einer früheren Arbeit behandelt. Es kann hier also davon ausgegangen werden, dass die benötigten Daten für das Update einer Komponente bereits im Speicher der Sensorknoten vorliegen. Definiert werden muss dagegen, welche Daten für die Installation einer neuen Komponente an die Sensorknoten übertragen werden müssen.

Auf der Seite der Sensorknoten muss diese Arbeit einen Mechanismus bereitstellen, der neue Komponenten von dem Codeverteilungsalgorithmus entgegennimmt und in die bestehende Programmstruktur integriert. Dabei müssen die Verbindungen zwischen den einzelnen Komponenten angepasst beziehungsweise neu erstellt werden. Das Vorgehen ähnelt dabei den Aktionen eines Linkers beim Kompilieren eines Programms.

Fragen, welche die Entscheidung zur Installation einer empfangenen Komponente betreffen, wie beispielsweise ein Versionsmanagement der Komponenten oder ein Mechanismus für die Adaption an lokale Begebenheiten durch bestimmte Komponentenkonfigurationen, gehen über die Zielsetzung dieser Arbeit hinaus und sollen in zukünftigen Arbeiten behandelt werden. Der in dieser Arbeit zu entwickelnde Update-Mechanismus entspricht also zunächst im Ansatz anderen Codeupdate-Verfahren (siehe Kapitel 3): Ein zentraler Rechner, dem die Konfiguration der Sensorknoten bekannt sein muss, verteilt Updates einzelner Komponenten im Netz, die wiederum von den Sensorknoten ohne weitere Prüfung installiert werden.

Die Entwicklung des Systems soll prototypisch für Sensorknoten vom Typ Mica2 [Crob] erfolgen.

# 5 Entwurf

In diesem Kapitel wird der Entwurf des Gesamtsystems vorgestellt. Zunächst wird hierfür im nächsten Abschnitt das generelle Vorgehen grob skizziert und die einzelnen Arbeitsschritte, die ein kompletter Updatevorgang umfasst, werden genauer beschrieben. Anschließend wird auf einige Annahmen eingegangen, auf denen die Implementierung aufbaut. Besonders wichtig ist der Aufbau der für die Implementierung verwendeten Datenstrukturen, der im letzten Abschnitt dieses Kapitels beschrieben wird.

## 5.1 Vorgehen und Arbeitsschritte

### Vorgehen

Die Sensornetzanwendungen werden vor der Installation auf den Sensorknoten in mehrere Komponenten aufgeteilt, die getrennt voneinander kompiliert werden können und über klar definierte Interfaces miteinander verbunden sind. Neben dem Programmcode der Sensornetzanwendung werden bei der Installation auch Metadaten im Speicher der Sensorknoten abgelegt, die den Aufbau des Programms beschreiben und die Symboltabelle sowie die Relocationtabellen des Programms enthalten. Mit Hilfe dieser Metadaten können zu einem späteren Zeitpunkt neue Komponenten in den Programmcode integriert werden. Im Falle der Mica2-Motes wird der externe Flash für die Speicherung dieser Metadaten genutzt.

Muss der Programmcode der Sensorknoten aktualisiert werden, so werden lediglich die tatsächlich geänderten Komponenten zusammen mit dafür benötigten Metadaten an die Sensorknoten übertragen. Auf den Sensorknoten werden die Daten der neuen Komponente nach dem Empfang ebenfalls im externen Flashspeicher abgelegt.

Die eigentliche Berechnung des neuen Programmcodes findet nicht parallel zum normalen Programmablauf statt. Sobald die Daten vollständig empfangen sind, wird stattdessen die Anwendung angehalten und der Updateprozess wird separat gestartet.

Bei der Berechnung des neuen Programmcodes werden die Daten im externen Flash zwischengespeichert und erst am Ende komplett in den Programmspeicher übertragen. Dies ist vorteilhaft, da der externe Flash häufiger beschrieben werden kann als der Programmspeicher und zudem in allen Situationen zugreifbar ist.

Nach dem Abschluss der Vorbereitungen wird der Programmcode in den Programmspeicher übertragen und die neu installierte Anwendung kann gestartet werden.

### Arbeitsschritte

Die notwendigen Arbeiten für den Austausch einer Komponente lassen sich in sieben getrennte Arbeitsschritte aufteilen, die aufeinander folgend abgearbeitet werden müssen:

1. **Vorbereitung des Programmcodes und der Metadaten:** Im ersten Schritt müssen auf dem zentralen Rechner die Daten der auszutauschenden Komponente für das Update vorbereitet werden. Hierfür muss der pure Binärcode aus der Objektdatei der Komponente extrahiert werden. Daneben müssen die Symbol- und Relocationinformationen der Komponente in einer binären Repräsentation vorbereitet werden.

2. **Übertragung der neuen Komponente an die Sensorknoten:** Die Übertragung der Komponentendaten an die Sensorknoten erfolgt im zweiten Schritt. Hierfür wird der bereits bestehende Codeverteilungsalgorithmus genutzt.

3. **Speicherung des Codes und der Metadaten:** Auf den Sensorknoten müssen der Programmcode und die Metadaten der neuen Komponente empfangen und zunächst im externen Speicher der Sensorknoten abgelegt werden. Dieser Schritt erfolgt parallel zum normalen Programmablauf der installierten Sensornetzanwendung.

4. **Aktualisieren der Symboltabelle:** Sobald die Daten der neuen Komponente vollständig empfangen wurden, kann mit der Bearbeitung des Komponentenupdates begonnen werden. Hierfür müssen in diesem Schritt zunächst die bestehenden Symbolinformationen des installierten Programms mit den Symbolinformationen der neuen Komponente aktualisiert und durch neue Symboleinträge ergänzt werden.

5. **Austausch der Relocationtabelle:** Mit dem neuen Programmcode der Komponente wurde auch eine neue Relocationtabelle an den Sensorknoten übertragen. In diesem Schritt muss die Relocationtabelle der alten Komponente durch die Daten der neuen Relocationtabelle ersetzt werden.

6. **Anpassung der Referenzen:** In diesem Schritt müssen die Relocationtabellen aller Komponenten des Programms durchlaufen werden. Alle durch das Update der Komponente potentiell geänderten Referenzen des Programmcodes müssen dabei auf Basis der neuen Symboldaten angepasst werden.

7. **Installation des neuen Programmcodes:** Nachdem der Programmcode in den vorigen Schritten im externen Speicher vorbereitet wurde, muss er im letzten Schritt in den Programmspeicher des Prozessors übertragen werden.

Als Bezeichnung für das Gesamtsystem wurde `FlexCup` ausgewählt. Dies steht für „**FLEX**ible **C**ode **UP**dates“. Tabelle 5.1 gibt einen Überblick über die einzelnen Arbeitsschritte und definiert eine Zuordnung zu den jeweils zuständigen Anwendungsteilen. Der erste Arbeitsschritt wird auf dem zentralen Rechner mit dem so genannten `FlexCup-`

| | |
|---|---|
| 1) Vorbereitung des Programmcodes und der Metadaten | `FlexCup`-Analyzer |
| 2) Übertragung der neuen Komponente an die Sensorknoten<br>3) Speicherung des Codes und der Metadaten | Codeverteilungs-algorithmus |
| 4) Aktualisieren der Symboltabelle<br>5) Austausch der Relocationtabelle<br>6) Anpassung der Referenzen | `FlexCup`-Linker |
| 7) Installation des neuen Programmcodes | `FlexCup`-Bootloader |

Tabelle 5.1: Aufteilung der Arbeitsschritte auf die `FlexCup`-Anwendungen

**Analyzer** ausgeführt. Die Verarbeitung der neu empfangenen Komponente übernimmt der `FlexCup`-**Linker**, während der `FlexCup`-**Bootloader** am Ende die Programmdaten in den Programmspeicher des Sensorknoten überträgt.

## 5.2 Annahmen

Die Entwicklung der `FlexCup`-Anwendung erfolgte auf Basis einer Reihe von Grundannahmen, die für den Betrieb von `FlexCup` erfüllt sein müssen. Teilweise handelt es sich dabei um einschränkende Annahmen, auf die nach der Integration von `FlexCup` in das Gesamtsystem im Rahmen der weiteren Entwicklung von `TinyCubus` verzichtet werden kann, zum Teil stellen die Annahmen aber auch grundsätzliche Anforderungen an das Sensorsystem dar, die für den Betrieb von `FlexCup` zwingend erfüllt sein müssen.

#### Zentraler Rechner mit globalem Wissen vorhanden

Bevor ein Codeupdate an die einzelnen Knoten eines Sensornetzes übertragen und nötigenfalls installiert werden kann, müssen zunächst der zu übertragende Programmcode und vor allem die zugehörigen Metadaten vorbereitet werden. Hierfür wird ein zentraler Rechner mit globalem Wissen vorausgesetzt.

Das benötigte globale Wissen bezieht sich primär auf die konsistente Vergabe der Bezeichner für die Komponenten sowie die Funktions- und Objektsymbole innerhalb des Netzwerks. Würden zwei Instanzen im Netzwerk unabhängig voneinander Bezeichner vergeben und diese bei der Erstellung der Metadaten verwenden, so müsste mit Doppelbelegungen gerechnet werden. Bei der Auflösung der Referenzen könnte es dann zu falschen Verbindungen zwischen den Komponenten kommen.

Globales Wissen bezüglich der Konfiguration der einzelnen Sensorknoten wird nicht zwingend vorausgesetzt, die einzelnen Sensorknoten können also prinzipiell autonom über die Installation einzelner Komponenten entscheiden.

### Sensorknoten speichern benötigte Metadaten

Um den Programmcode empfangener Komponenten installieren zu können, benötigen die Sensorknoten nicht nur die Metadaten der neuen Komponente, sondern auch Informationen zum Gesamtprogramm sowie den einzelnen bereits installierten Komponenten. Es wird vorausgesetzt, dass der Sensorknoten diese Informationen speichert und `FlexCup` zur Verfügung stellt. `FlexCup` wiederum aktualisiert die Informationen im Rahmen der Installation einer neuen Komponente.

### Update liegt im externen Flash bereit

Der `FlexCup`-Linker geht davon aus, dass alle im Rahmen eines Updates benötigten Daten bereits im externen Flash des Sensorknotens vorliegen. Ein Algorithmus für die Übertragung der Programmdaten an die Sensorknoten wurde bereits von Robert Sauter im Rahmen seiner Diplomarbeit entwickelt (siehe Abschnitt 4.2), prinzipiell lässt sich `FlexCup` aber auch mit beliebigen anderen Verfahren zur Verteilung von Programmcode kombinieren.

### Kein Komponenten- und Versionsmanagement notwendig

In dieser Arbeit wird vereinfachend angenommen, dass die Entscheidung zur Installation einer neuen Komponente beim Aufruf des `FlexCup`-Linkers bereits extern getroffen wurde, so dass die neue Komponente in jedem Fall installiert werden kann. Es wird dementsprechend nicht überprüft, ob die zu installierende Komponente innerhalb der Anwendung überhaupt benötigt wird, ein Versionsabgleich zwischen der neuen Komponente und einer eventuell bereits installierten Version der Komponente findet ebenfalls nicht statt.

Die Entscheidung über die Installation oder den Austausch einer Komponente gehört zu den Aufgaben des Tiny Data Management Frameworks (siehe Abschnitt 4.1) und muss im Rahmen zukünftiger Arbeiten realisiert werden. Einen wichtigen Aspekt stellen hierbei auch die Abhängigkeiten zwischen den Komponenten dar. Es muss sichergestellt werden, dass die installierten Versionen der Komponenten dem `FlexCup`-Linker die Auflösung aller Referenzen ermöglichen. Konkret bedeutet dies, dass alle von einer Komponente benötigten Ressourcen von einer anderen Komponente innerhalb des Gesamtprogramms bereitgestellt werden müssen.

### Deaktivieren und Neustart des Systems möglich

`FlexCup` geht davon aus, dass die Anwendungsszenarien es zulassen, im Rahmen des Austauschs einer Komponente die Anwendung kurzzeitig zu deaktivieren und einen Neustart

des Systems durchzuführen. Ein vollständig unterbrechungsfreier Betrieb des Sensorsystems kann während des Update-Prozesses aus zweierlei Gründen nicht gewährleistet werden. Zum einen benötigen Teile des Mechanismus für einen gewissen, möglichst kurzen Zeitraum die alleinige Kontrolle über das System. Zum anderen ändert das Update einer Komponente eventuell Teile des Speicherlayouts im Datenspeicher. Ohne einen Neustart des Systems müssten im Speicher befindliche Daten überwacht und bei Bedarf umsortiert werden, um die Integrität der Daten über das Update hinaus sicherstellen zu können.

Benötigt die Anwendung eines Sensorknotens Zustandsdaten, die über einen Neustart hinaus erhalten bleiben müssen, so müssen diese vor dem Beginn des Codeupdates explizit gespeichert werden. Zu diesem Zweck sollte in `TinyCubus` zukünftig ein Mechanismus integriert werden, der auf Aufforderung alle Zustandsdaten des Sensorknotens speichert und nach einem Codeupdate wieder restauriert.

## 5.3 Aufbau der Daten im Speicher der Sensorknoten

### 5.3.1 Entwurfsprinzipien

#### Minimaler Speicherverbrauch

Wie alle Ressourcen in Sensornetzanwendungen sind auch der flüchtige und der nichtflüchtige Speicher der Sensorknoten starken Beschränkungen unterworfen. Die Metadaten der installierten Anwendungen müssen aus diesem Grund in einem möglichst Platz sparenden Format gespeichert werden. Außerdem ist darauf zu achten, dass nur die tatsächlich notwendigen Metadaten auf den Sensorknoten gespeichert werden.

Eine Platz sparende Speicherform der Metadaten entspricht auch dem Ziel, möglichst nur geringe Datenmengen über die Funkschnittstelle an die Sensorknoten übertragen zu müssen. Da Einsparungen bei den zu übertragenden Datenvolumina besonders hoch zu bewerten sind, kann es sich in manchen Fällen lohnen, für die Übertragung der Metadaten ein gesondertes, noch Platz sparenderes Format einzusetzen.

#### Effiziente Verarbeitung

Neben dem Ziel der kompakten Speicherung ist beim Entwurf der Metadaten auch darauf zu achten, dass der Aufbau der Daten eine effiziente Verarbeitung ermöglicht. Besonders Platz sparende Kodierungen können unter Umständen einen großen Aufwand der Sensorknoten bei der Dekodierung und Verarbeitung der Metadaten erfordern. Es gilt deswegen einen Kompromiss zu finden zwischen der platzoptimalen Speicherung der Metadaten und der effizienten Verarbeitung dieser Daten durch die Sensorknoten.

Der in Abschnitt 5.3.3 vorgestellte Aufbau der Metadaten entspringt keiner einmaligen Designentscheidung, sondern ist in seiner jetzigen Form durch mehrfache Rückkopplung

während der Implementierung des `FlexCup`-Systems entstanden. Dadurch konnten die Anforderungen der einzelnen Algorithmen an den Aufbau der Metadaten in den Entwurf eingearbeitet werden.

#### Globale Eindeutigkeit der Bezeichner

Der zentrale Rechner, der die Metadaten für die Übertragung neuer oder geänderter Komponenten an die Sensorknoten vorbereitet, ist nicht zwangsläufig über die genaue Konfiguration aller Sensorknoten im Netzwerk informiert. Aus diesem Grund ist es wichtig, dass alle verwendeten Bezeichner global eindeutig sind. Würde ein Bezeichner mehrmals in unterschiedlichen Kontexten verwendet, so könnte es passieren, dass zwei Sensorknoten die Daten einer empfangenen Komponente unterschiedlich interpretieren. Selbst bei einem längst nicht mehr verwendeten Bezeichner einer entfernten oder ausgetauschten Komponente ist es prinzipiell möglich, dass der Programmcode der alten Komponente und damit auch der alte Bezeichner noch auf irgendeinem Sensorknoten verwendet wird.

### 5.3.2 Bezeichner

Die Metadaten der Programme in `FlexCup` benötigen Bezeichner, um unterschiedliche Komponenten sowie die Symbole in den Symbol- und Relocationtabellen unterscheiden zu können. In den Relocationtabellen gibt es zudem unterschiedliche Relocationtypen, die anhand eines Relocationtypbezeichners unterschieden werden.

**Komponentenbezeichner:** Um die einzelnen Komponenten eines Programms unterscheiden zu können, bekommt jede Komponente einen Komponentenbezeichner zugewiesen. Für jeden dieser Bezeichner ist 1 Byte Speicherplatz vorgesehen, es können also maximal 256 unterschiedliche Komponenten identifiziert werden.

**Symbolbezeichner:** Für die Bezeichner der Einträge in der Symboltabelle werden je zwei Bytes Speicher reserviert. Das höchstwertige Bit jedes Symbolbezeichners wird allerdings innerhalb von `FlexCup` als Markierungsbit eingesetzt, so dass nur 15 der 16 Bits für den eigentlichen Bezeichner zur Verfügung stehen. Insgesamt sind damit 32768 unterschiedliche Symbolbezeichner möglich.

**Relocationtypbezeichner:** Wie in Tabelle 2.2 aufgeführt, gibt es insgesamt 19 unterschiedliche Relocationtypen, die vom AVR-GCC-Compiler erzeugt werden. Für die Speicherung eines Bezeichners, der die Angabe des jeweiligen Typs eines Relocationeintrags erlaubt, ist damit 1 Byte Speicherplatz ausreichend.

### 5.3.3 Struktur der Metadaten auf dem Sensorknoten

Innerhalb der Metadaten wurde folgende Struktur festgelegt: An vorderster Stelle werden allgemeine Informationen gespeichert, die das Gesamtprogramm beschreiben

| | |
|---|---|
| Startadresse der Kopie des Programmspeichers | 2 Bytes |
| Startadresse der Initialisierungsdaten (*.data*-Abschnitt) | 2 Bytes |
| Größe der Initialisierungsdaten (*.data*-Abschnitt) | 2 Bytes |
| Startadresse der Symboltabelle | 2 Bytes |
| Startadresse der neuen Komponente | 2 Bytes |

Tabelle 5.2: Aufbau der allgemeinen Programminformationen

(**Programminformationen**). Direkt darauf folgen die Metadaten der einzelnen Komponenten (**Komponenteninformationen**). Dahinter wird die **Symboltabelle** gespeichert und im Anschluss daran folgen die **Relocationtabellen** der einzelnen Komponenten.

Momentan werden alle Adressinformationen innerhalb der Metadaten in Datenstrukturen mit einer Größe von zwei Bytes gespeichert. Damit alle Teile der Metadaten mit diesem Adressraum erreichbar sind, müssen sie innerhalb der ersten 64 Kilobyte des externen Flash gespeichert werden. Für alle bisher untersuchten Anwendungen ist diese Speichergröße mehr als ausreichend. Bei Bedarf sollte eine Erweiterung der Adressfelder auf 3 Bytes Größe problemlos möglich sein, führt dann aber zwangsläufig zu einem größeren Speicherverbrauch. Alternativ dazu könnten, wie bei der Adressierung des Programmspeichers in AVR-Prozessoren, zwei Bytes große Blöcke anstatt einzelner Bytes adressiert werden. Dadurch stünden immerhin 128 Kilobyte des externen Flash zur Verfügung. Eine weitere Möglichkeit ist die Verwendung von Offsetwerten, die vor dem Speicherzugriff zu den einzelnen Adressen addiert werden müssten. Durch die Verwendung verschiedener Offsets für verschiedene Bereiche könnte so der vollständige externe Flash verwendet werden.

### Programminformationen

Im ersten Abschnitt der Metadaten werden allgemeine Programminformationen gespeichert, die sich auf das aktuell installierte Gesamtprogramm beziehen. Tabelle 5.2 gibt einen Überblick über den Inhalt und den Aufbau des Abschnitts. Da die Programminformationen mit 10 Bytes eine feste Größe besitzen, kann auf alle Einträge sowie auf den Beginn der nachfolgenden Komponenteninformationen direkt zugegriffen werden.

Der letzte Eintrag „Startadresse der neuen Komponente“ steht nur indirekt mit dem installierten Programm in Verbindung: Falls eine neue Komponente zur Installation im Speicher bereitliegt, so speichert dieser Eintrag die Startadresse der Metadaten der neuen Komponente. Diese Information wird während des Updatevorgangs benötigt, um direkt auf die Metadaten der neuen Komponente zugreifen zu können.

| | | |
|---|---|---|
| Anzahl der Komponenten **A** | 1 Byte | |
| Komponentenbezeichner | 1 Byte | **A**-mal |
| Komponentengröße | 2 Bytes | |
| Startadresse der Komponente | 2 Bytes | |
| Startadresse der Relocationtabelle | 2 Bytes | |
| Startadresse des *.data*-Abschnitts | 2 Bytes | |
| Größe des *.data*-Abschnitts | 2 Bytes | |
| Größe des *.data*-Abschnittpuffers | 1 Byte | |

Tabelle 5.3: Aufbau der Komponenteninformationen

| | | |
|---|---|---|
| Anzahl der Funktionssymbole **A** | 2 Bytes | |
| Anzahl der Objektsymbole **B** | 2 Bytes | |
| Symbolbezeichner | 2 Bytes | **A**-mal |
| Symboladresse | 2 Bytes | |
| Symbolbezeichner | 2 Bytes | **B**-mal |
| Symboladresse | 2 Bytes | |
| Symbolgröße | 2 Bytes | |

Tabelle 5.4: Aufbau der Symboltabelle

### Komponenteninformationen

In den Komponenteninformationen werden Metadaten der einzelnen Programmkomponenten gespeichert. Am Anfang des Abschnitts steht die Anzahl der Komponenten, darauf folgen dann für jede Komponente 12 Bytes an Metadaten (siehe Tabelle 5.3).

Das Datenfeld „Größe des *.data*-Abschnittpuffers“ wird bei der Verwendung von Pufferspeicher zwischen den *.data*-Abschnitten der Komponenten benötigt. Dabei handelt es sich um eine Optimierung, die in Abschnitt 6.6.2 beschrieben wird.

### Symboltabelle

Da die Startadresse der Symboltabelle von der Größe der Komponenteninformationen abhängt, wird sie in einem Datenfeld der allgemeinen Programminformationen gespeichert.

Der Aufbau der Symboltabelle selbst ist in Tabelle 5.4 abgebildet. Funktions- und Objektsymboleinträge werden getrennt voneinander in zwei separaten Abschnitten gespeichert. Für jeden Funktionssymboleintrag müssen der Symbolbezeichner und die Symboladresse, für jeden Objektsymboleintrag zusätzlich auch noch die Größe des Symbols gespeichert werden. Jeder Funktionssymboleintrag erfordert damit 4 Bytes Speicherplatz, jeder Objektsymboleintrag 6 Bytes.

### Relocationtabellen

Für jede Komponente des installierten Programms wird in den Metadaten eine Tabelle mit den Relocationeinträgen zu dem Programmcode der Komponente gespeichert. Die Startadresse dieser Relocationtabellen kann jeweils in den zugehörigen Komponenteninformationen gefunden werden.

Um möglichst wenig Informationen speichern zu müssen, wird in den Metadaten zwischen vier unterschiedlichen Typen von Relocationeinträgen unterschieden: **„Internal Call“**-Einträge, **„External Call“**-Einträge, **„Storage Relocation ohne Offset“**-Einträge, sowie **„Storage Relocation mit Offset“**-Einträge. Wie Tabelle 5.5 zeigt, enthalten die Relocationtabellen für jeden dieser Typen einen eigenen Abschnitt. Um die einzelnen Abschnitte innerhalb der Relocationtabelle schnell erreichen zu können, wird am Anfang der Tabelle die Größe der vier Abschnitte gespeichert.

**„Internal Call“-Abschnitt:** In diesem Abschnitt sind alle internen R_AVR_CALL-Einträge der Relocationtabelle zu finden, also alle Einträge, bei denen es sich um einen Verweis innerhalb der Komponente handelt. Diese Einträge besitzen die Form „*Quelladresse* `R_AVR_CALL .text+`*Offset*“. Speichern muss die Relocationtabelle die Quelladresse, von der die Referenz ausgeht, sowie den Zieloffset innerhalb der Komponente.

Um den Zieloffset nicht für alle Einträge angeben zu müssen, werden die Einträge nach Zieloffsets sortiert gespeichert. Zu diesem Zweck enthält der „Internal Call“-Abschnitt zunächst ein Datenfeld, das angibt, wie viele unterschiedliche Ziele durch interne Sprünge referenziert werden. Getrennt für jedes dieser Ziele folgt nun die Angabe des Zieloffsets, sowie eine Liste mit den Quelladressen der Referenzen auf diesen Zieloffset.

**„External Call“-Abschnitt:** Der „External Call“-Abschnitt ist ähnlich aufgebaut wie der „Internal Call“-Abschnitt. Er enthält alle Einträge der Relocationtabelle, die sich auf externe Aufrufe beziehen („*Quelladresse* `R_AVR_CALL` *Zielsymbol*“). Speichern muss die Relocationtabelle die Quelladresse der Referenz sowie den Symbolbezeichner des Sprungziels. Auch in diesem Abschnitt wird Speicherplatz dadurch eingespart, dass die Relocationeinträge gruppiert nach Symbolbezeichnern aufgeführt werden, wodurch nicht für jeden Eintrag die Angabe des Symbolbezeichners notwendig ist.

**„Storage Relocation ohne Offset“-Abschnitt:** Referenzen auf Datensymbole können entweder direkt ein Symbol referenzieren, oder zusätzlich zum Symbolnamen noch einen Offset angeben, der bei der Auflösung des Relocationeintrags zur Symboladresse hinzuaddiert werden muss. Im „Storage Relocation ohne Offset“-Abschnitt müssen die Quelladresse, der Typ sowie das Zielsymbol des Relocationeintrags gespeichert werden.

Um eine kompaktere Speicherung zu erreichen, werden die Relocationeinträge zunächst bezüglich ihres Relocationtyps gruppiert. Innerhalb der Menge der Relocationeinträge gleichen Typs werden die Einträge nach Zielsymbol sortiert eingetragen. Auf diese Weise muss die Angabe des Relocationtyps nur je einmal, die Angabe des Zielsymbols nur

| | | | |
|---|---|---|---|
| Größe der Relocationtabelle | 2 Bytes | | |
| Größe des „Internal Call“-Abschnitts | 2 Bytes | | |
| Größe des „External Call“-Abschnitts | 2 Bytes | | |
| Größe des „Storage Relocation ohne Offset“-Abschnitts | 2 Bytes | | |
| Größe des „Storage Relocation mit Offset“-Abschnitts | 2 Bytes | | |
| Anzahl der „Internal Call“-Ziele **A** | 1 Byte | | |
| Offset des Sprungziels der Relocations-Gruppe | 2 Bytes | **A**-mal $0 \leq U < A$ | |
| Anzahl der Relocations $\mathbf{B_U}$ für das jeweilige Ziel | 1 Byte | | |
| Adresse der Relocationquelle | 2 Bytes | $\mathbf{B_U}$-mal | |
| Anzahl der „External Call“-Ziele **C** | 1 Byte | | |
| Symbolbezeichner des Sprungziels der Relocations-Gruppe | 2 Bytes | **C**-mal $0 \leq V < C$ | |
| Anzahl der Relocations $\mathbf{D_V}$ für das jeweilige Ziel | 1 Byte | | |
| Adresse der Relocationquelle | 2 Bytes | $\mathbf{D_V}$-mal | |
| Anzahl der „Storage Relocation“-Typen ohne Offset **E** | 1 Byte | | |
| „Storage Relocation“-Typ-Bezeichner | 1 Byte | | |
| Anzahl der „Storage Relocation“-Ziele $\mathbf{F_W}$ | 1 Byte | | **E**-mal $0 \leq W < E$ |
| Symbolbezeichner des Relocationziels | 2 Bytes | $\mathbf{F_W}$-mal $0 \leq X < F_W$ | |
| Anzahl der Relocations $\mathbf{G_X}$ | 1 Byte | | |
| Adresse der Relocationquelle | 2 Bytes | $\mathbf{G_X}$-mal | |
| Anzahl der „Storage Relocation“-Typen mit Offset **H** | 1 Byte | | |
| „Storage Relocation“-Typ-Bezeichner | 1 Byte | | |
| Anzahl der „Storage Relocation“-Ziele $\mathbf{I_Y}$ | 1 Byte | | |
| Symbolbezeichner des Relocationziels | 2 Bytes | | **H**-mal $0 \leq Y < H$ |
| Anzahl der Relocations $\mathbf{J_Z}$ | 1 Byte | $\mathbf{I_Y}$-mal $0 \leq Z < I_Y$ | |
| Adresse der Relocation-Quelle | 2 Bytes | $\mathbf{J_Z}$-mal | |
| Offset vom Relocation-Symbol | 2 Bytes | | |

Tabelle 5.5: Aufbau der Relocationtabellen

je einmal pro Relocationtyp erfolgen. Der genaue Aufbau des resultierenden „Storage Relocation ohne Offset“-Abschnitts ist Tabelle 5.5 zu entnehmen.

**„Storage Relocation mit Offset“-Abschnitt:** Die Einträge des „Storage Relocation mit Offset“-Abschnitts entsprechen in ihrem Aufbau den Einträgen des „Storage Relocation ohne Offset“-Abschnitts. Der einzige Unterschied besteht darin, dass für jeden Eintrag dieses Abschnitts zusätzlich ein Datenfeld für den Offset vorhanden ist, der bei der Auflösung des Eintrags zur Symboladresse hinzuaddiert werden muss.

Ein „Storage Relocation mit Offset“-Eintrag kann auf ein Datensymbol des Programms Bezug nehmen und mit Hilfe des Offsets den Zugriff auf einen bestimmten Teilbereich des repräsentierten Datenobjekts realisieren. Alternativ kann sich ein Eintrag auch auf den *.data*-Abschnitt der jeweiligen Komponente beziehen und so eine „*.data* + Offset“-Relocation abbilden. Für die Speicherung und Bearbeitung der Relocationeinträge macht dies keinen Unterschied, da die *.data*-Abschnitte der Komponenten jeweils durch normale Symboleinträge in der Symboltabelle repräsentiert werden.

### 5.3.4 Struktur der Programmdaten

Neben der Struktur der Metadaten muss auch eine Strukturierung für die Programmdaten festgelegt werden. Dazu gehört zum einen die Struktur der Daten des Programmspeichers, zum anderen muss auch das Layout der Anwendungsdaten im RAM, also im Datenspeicher der Sensorknoten, definiert werden. In beiden Fällen wird hierfür das Standardlayout des AVR-GCC-Linkers übernommen, so dass die Struktur der in Abschnitt 2.4.1 beschriebenen Struktur entspricht.

### 5.3.5 Struktur der Metadaten einer neuen Komponente

Für den Austausch einer Komponente auf den Sensorknoten müssen neben dem neuen Programmcode der Komponente auch verschiedene Metadaten an die Sensorknoten geschickt werden. Zum einen müssen die **Komponenteninformationen** der Komponente aktualisiert werden, außerdem müssen eine neue **Relocationtabelle** sowie neue **Symbolinformationen** an den Sensorknoten übertragen werden.

Der im Folgenden beschriebene Aufbau der Metadaten einer neuen Komponente gilt sowohl während der Übertragung der Metadaten an die Sensorknoten, als auch für die temporäre Speicherung der Metadaten im externen Flash der Sensorknoten, bevor die Daten verarbeitet und mit den Metadaten des Programms kombiniert werden.

#### Komponenteninformationen

Die Felder der zu übertragenden Komponenteninformationen, dargestellt in Tabelle 5.6, ähneln den Feldern der Komponenteninformationen in den Metadaten der Sensorknoten.

| Komponentenbezeichner | 1 Byte |
|---|---|
| Komponentengröße | 2 Bytes |
| Startadresse des Komponente | 2 Bytes |
| Startadresse der Relocationtabelle | 2 Bytes |
| Größe des *.data*-Abschnitts | 2 Bytes |

Tabelle 5.6: Aufbau der Komponenteninformationen einer neuen Komponente

Für die Übertragung können jedoch die Abschnitte „Startadresse des *.data*-Abschnitts“ und „Größe des *.data*-Abschnitt-Puffers“ eingespart werden, da diese Informationen erst lokal auf den Sensorknoten generiert werden.

### Relocationtabelle

Die Relocationtabelle der neuen Komponente entspricht in ihrem Format vollständig den Relocationtabellen in den Metadaten der Sensorknoten, wie sie in Tabelle 5.5 dargestellt sind.

### Symbolinformationen

Es ist nicht notwendig, mit einer neuen Komponente auch die komplette Symboltabelle des Programms erneut zu übertragen. Es genügt stattdessen, die für die Komponente relevanten und die durch die Komponente beeinflussbaren Symbolinformationen zu übertragen und mit diesen Informationen dann die Symboltabelle der Sensorknoten zu aktualisieren. Konkret werden die folgenden Informationen benötigt:

- **Funktionssymbole:** Es müssen nur die Symbolinformationen derjenigen Funktionssymbole übertragen werden, deren Adressen innerhalb der neuen Komponente liegen. Diese Funktionssymbole geben Aufschluss über die Startadressen der einzelnen Funktionen innerhalb der neuen Komponente. Diese Information wird benötigt, um die Referenzen auf den Programmcode der Komponente korrekt setzen zu können.
- ***.data*-Symbole:** Besitzt die Komponente einen *.data*-Abschnitt, so muss die Position der einzelnen *.data*-Symbole innerhalb dieses Abschnitts übertragen werden. Auf den Sensorknoten kann mit diesen Informationen dann die Adresse der Symbole im Datenspeicher bestimmt werden.
- **Objektsymbole:** Zusätzlich zu den Symbolen des *.data*-Abschnitts der Komponente müssen auch Informationen zu allen Objektsymbolen übertragen werden, die durch den Programmcode der Komponente referenziert werden. Die Sensorknoten können dann bestimmen, ob es sich bei den Symbolen um Symbole des *.data*-Abschnitts einer anderen Komponente handelt, oder ob es sich um ein bestehendes oder ein neues Symbol des *.bss*-Abschnitts handelt. Um fehlende Objektsymbole

| | | |
|---|---|---|
| Anzahl der Funktionssymbole **A** | 2 Bytes | |
| Anzahl der *.data*-Symbole **B** | 2 Bytes | |
| Anzahl der kleinen Objektsymbole **C** | 2 Bytes | |
| Anzahl der großen Objektsymbole **D** | 2 Bytes | |
| Symbolbezeichner | 2 Bytes | **A**-mal |
| Symboladresse | 2 Bytes | |
| Symbolbezeichner | 2 Bytes | **B**-mal |
| Lokaler Symboloffset | 2 Bytes | |
| Symbolbezeichner | 2 Bytes | **C**-mal |
| Symbolgröße | 1 Byte | |
| Symbolbezeichner | 2 Bytes | **D**-mal |
| Symbolgröße | 2 Bytes | |

Tabelle 5.7: Aufbau der Symbolinformationen der neuen Komponente

im *.bss*-Abschnitt anlegen zu können, muss die Größeninformation der Symbole mit übertragen werden. Eine Angabe der Symboladresse ist dagegen nicht notwendig, da die Sensorknoten autonom über die Platzierung der Symbole im Datenspeicher entscheiden können.

Die meisten Datensymbole eines Programms sind (erheblich) kleiner als 256 Bytes, für die Speicherung ihrer Größeninformation reicht damit 1 Byte aus. Dieser Umstand wird für Einsparungen beim Umfang der Daten ausgenutzt, indem bei der Übertragung zwischen kleinen und großen Objektsymbolen unterschieden wird. Für kleine Objektsymbole genügt ein einzelnes Byte für die Speicherung der Größeninformation, während für große Objektsymbole zwei Bytes reserviert werden müssen. Für die zu übertragenden Symbolinformationen ergeben sich somit vier Abschnitte: Der **Funktionssymbolabschnitt**, der ***.data*-Symbolabschnitt**, der **„Kleine Objektsymbole“-Abschnitt** und der **„Große Objektsymbole“-Abschnitt**. Der Aufbau der gesamten Symbolinformationen ist in Tabelle 5.7 dargestellt.

### 5.3.6 Gesamtaufbau der Daten im externen Flash

Abbildung 5.1 gibt anhand eines Beispiels einen Überblick über den Gesamtaufbau der Daten im externen Flash. Das Beispiel zeigt die Daten eines einfachen Programms, das aus den drei Komponenten A, B und C aufgebaut ist. Zusätzlich liegen in dieser Momentaufnahme im externen Flash noch die Daten einer Komponente B’ bereit, mit der die Komponente B ersetzt werden soll.

Am Beginn des externen Flash werden die Metadaten des Programms platziert, ihr detaillierter Aufbau wird in Abschnitt 5.3.3 erklärt. Zwischen den einzelnen Abschnitten der Metadaten wird jeweils eine gewisse Menge Speicherplatz freigelassen (in der Abbildung schraffiert dargestellt), um für mögliche zukünftige Größenänderungen gerüstet zu sein.

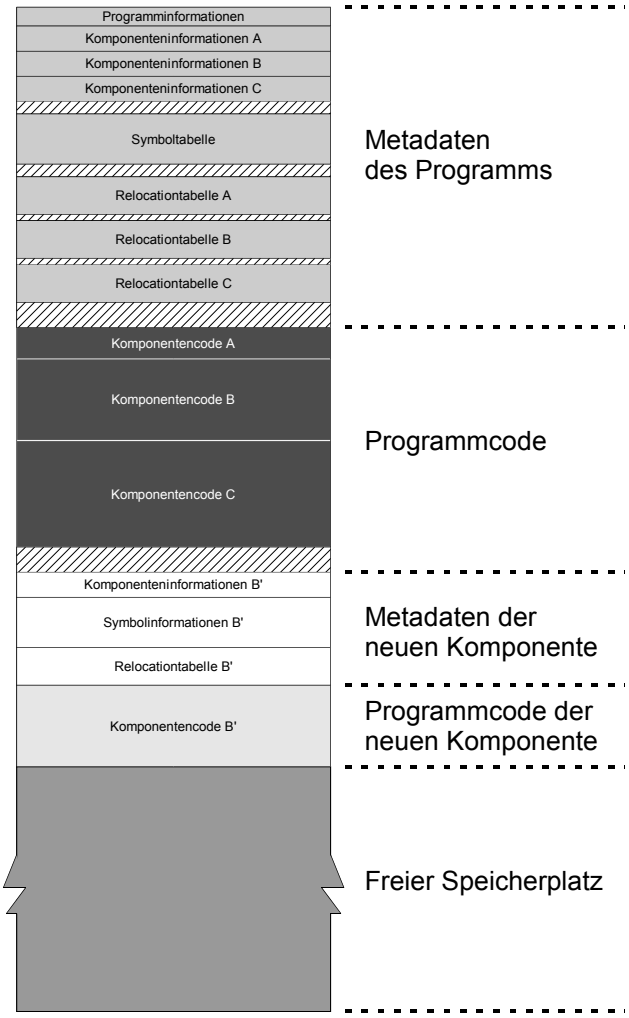


Abbildung 5.1: Aufbau der Daten im externen Flash

Auf die Metadaten folgt im externen Flashspeicher der Programmcode der Anwendung (siehe Abschnitt 5.3.4).

Da im dargestellten Fall eine neue Komponente zur Installation bereitliegt, folgen die Metadaten (siehe Abschnitt 5.3.5) und anschließend der Programmcode der neuen Komponente.

# 6 Implementierung

Nach der Vorstellung des Gesamtsystems im letzten Kapitel werden in diesem Kapitel die einzelnen Schritte der Implementierung des `FlexCup`-Systems beschrieben. Hierbei wird insbesondere genauer auf den `FlexCup`-Analyzer, den `FlexCup`-Linker und den `FlexCup`-Bootloader eingegangen.

## 6.1 Aufteilung von nesC-Anwendungen in Komponenten

Um bei TinyOS-Anwendungen mit `FlexCup` einzelne Komponenten austauschen zu können, müssen die Anwendungen vor der Installation zunächst in geeignete Komponenten aufgeteilt werden. Da die prinzipiell bereits vorhandene Komponentenstruktur der nesC-Programme nach dem Kompilieren nicht automatisch erhalten bleibt, muss dies durch manuelle Eingriffe erreicht werden. Für diese Aufgabe wird das neue Sprachkonzept von nesC zum Erstellen binärer Komponenten eingesetzt (siehe auch Abschnitt 2.3.3).

Der Benutzer muss zukünftig seine Anwendungs- und die TinyOS-Systemkomponenten auf eine sinnvolle Anzahl binärer Komponenten aufteilen, die binären Komponenten zu einem Gesamtprogramm zusammenfügen und dieses anschließend auf den Sensorknoten installieren. Danach hat er die Möglichkeit, mit `FlexCup` jederzeit einzelne binäre Komponenten auszutauschen.

### 6.1.1 Aufteilen der Systemkomponenten

Typische Anwendungen, die für TinyOS-Sensornetze geschrieben wurden, enthalten nur eine überschaubare Zahl an eigenen Anwendungskomponenten, verwenden aber gleichzeitig viele unterschiedliche TinyOS-Module und Konfigurationen. Im Extremfall besteht eine Anwendung nur aus einem einzelnen Anwendungsmodul sowie einer Konfigurationsdatei, die das Modul mit verschiedenen Systemkomponenten verbindet. Vor diesem Hintergrund betrachtet wird klar, dass sich die Aufteilung einer Anwendung in Binärkomponenten nicht auf den Bereich des reinen Anwendungscodes beschränken, sondern auch die verwendeten Systemkomponenten von TinyOS umfassen sollte.

Momentan generiert nesC aus den Anwendungskomponenten zusammen mit den verwendeten Systemkomponenten eine einzelne C-Quelldatei. Welche Systemkomponenten dabei tatsächlich zum Einsatz kommen, ist für den Anwendungsprogrammierer zunächst

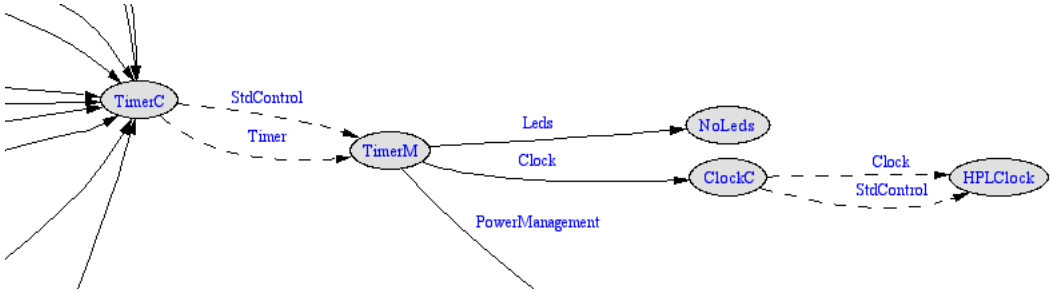


Abbildung 6.1: Die Timer-Komponente

nicht ersichtlich, da die von ihm integrierten Komponenten rekursiv weitere Komponenten verwenden können. Für die Aufteilung der Anwendung in Binärkomponenten muss nun jede dieser Systemkomponenten einer Binärkomponente zugewiesen werden.

### Vorgehen

Für große Teile der in den Anwendungen enthaltenen TinyOS-Systemkomponenten kann recht intuitiv eine Aufteilung in Binärkomponenten gefunden werden, die für unterschiedliche, mit diesen Systemkomponenten realisierte Anwendungen Bestand hat.

Für die Entscheidung, welche Komponenten zu einer Binärkomponente zusammengefasst werden sollten, hat sich die folgende Faustregel als hilfreich erwiesen: Besitzt eine Komponente Verbindungen zu einer großen Zahl anderer, voneinander unabhängiger Komponenten, so sollte auf Basis dieser Komponente eine neue Binärkomponente gebildet werden. In diese neue Binärkomponente sollten dann wiederum die Komponenten aufgenommen werden, die mit der ersten Komponente semantisch in enger Verbindung stehen und gleichzeitig keine oder nur wenige eigene Verbindungen nach außen besitzen.

Verdeutlicht werden soll dieses Vorgehen anhand eines Beispiels, das in Abbildung 6.1 dargestellt ist. Die Abbildung zeigt einen Ausschnitt der graphischen Übersichtsdarstellung einer nesC-Anwendung. Die Grafik wurde mit Hilfe des Programms *nesdoc* erzeugt, das als Teil des nesC-Compilerpakets automatisch Dokumentationen für nesC-Programme erstellt.

Der Ausschnitt zeigt eine Sammlung von Komponenten, die zusammen die Funktionalität eines Zeitgebers (Timer) implementieren. Die Komponente *TimerC* stellt anderen Modulen das Interface *Timer* sowie das Interface *StdControl* zur Verfügung. Über die *Timer*-Schnittstelle kann eine Komponente einen Timer registrieren, der dann in frei definierbaren Zeitintervallen die Komponente per Event-Aufruf aktiviert. Sowohl das Interface *Timer* als auch das Interface *StdControl* werden von zahlreichen anderen Komponenten der Anwendung verwendet. Erkennbar ist dies an den vielen eingehenden Pfeilen in der graphischen Darstellung. Es bietet sich demnach an, für den Timer eine eigene Binärkomponente zu erstellen. Ebenfalls in der graphischen Darstellung erkennbar ist, dass

```
component TimerComponent
{
  provides {
    interface Timer[uint8_t id];
    interface StdControl;
  }
  uses {
    interface PowerManagement;
  }
}
```

Abbildung 6.2: Komponentendefinition der Timer-Komponente

die semantisch zum Timer passenden Komponenten *TimerM*, *ClockC* und *HPLClock* so gut wie keine Beziehungen nach außen (zu anderen Komponenten beziehungsweise Binärkomponenten) besitzen. Die einzige Ausnahme ist die Verbindung von *TimerM* zu einer Implementierung des *PowerManagement*-Interfaces (NoLeds ist eine leere Implementierung und kann hier ignoriert werden). Gemäß der Faustregel können also *TimerC*, *TimerM*, *ClockC* und *HPLClock* zu einer Timer-Binärkomponente zusammengefasst werden. Abbildung 6.2 zeigt die Komponentendefinition dieser Binärkomponente.

Ein weiteres Kriterium, das bei der Aufteilung des Systems in Binärkomponenten gut angewendet werden kann, sind die Erwartungen bezüglich des Austauschs einzelner Komponenten. Rechnet man bei Komponenten damit, dass sie unabhängig von anderen Komponenten ausgetauscht werden könnten, so sollte man für diese Komponenten eine eigene Binärkomponente definieren.

Selbstverständlich lassen sich die in diesem Abschnitt beschriebenen Kriterien und Regeln nicht nur für die Aufteilung der Systemkomponenten von TinyOS einsetzen. Bei sehr großen Anwendungen lässt sich auf die gleiche Weise auch eine Aufteilung der Anwendungskomponenten finden.

### Erwartete Situation in der Zukunft

Sobald der Mechanismus für das Erstellen von Binärkomponenten in nesC endgültig verfügbar ist, ist damit zu rechnen, dass eine steigende Anzahl der Anwendungen und Bibliotheken in TinyOS auch oder sogar primär als Binärkomponenten vorliegen. Gut vorstellbar ist darüber hinaus, dass in Zukunft das TinyOS-System selber in Form einer Sammlung von Binärkomponenten bereitliegt. Der Compiler muss in diesem Fall nur noch den Anwendungscode kompilieren und mit den benötigten Binärkomponenten des TinyOS-Systems verlinken.

Nimmt man an, dass zukünftig die Systemkomponenten bereits in passende Binärkomponenten aufgeteilt sind, so muss sich der Anwendungsentwickler nur noch um die Aufteilung des von ihm erstellten Anwendungscodes kümmern. Gerade bei kleineren An-

wendungen kann auch diese Aufteilung entfallen und der Anwendungscode komplett in einer einzelnen Binärkomponente untergebracht werden. Der zusätzliche Aufwand des Entwicklers für das Zerteilen in Binärkomponenten bleibt damit auf jeden Fall gering, da er sich nur um seine eigenen Anwendungskomponenten kümmern muss. Hier besitzt der Entwickler bereits einen recht guten Überblick über mögliche, zur Semantik der Anwendung passende Aufteilungen.

Während also momentan das Aufteilen der Systemkomponenten in sinnvolle Binärkomponenten einen erheblichen Aufwand verursacht, wird dieser Aufwand in Zukunft voraussichtlich vollständig entfallen oder nur in geringem Umfang notwendig sein. Dies ist eine wichtige Voraussetzung für den Erfolg von `FlexCup`, da hier eine sinnvolle Aufteilung der Anwendung in Binärkomponenten zwingend erforderlich ist.

### 6.1.2 Manuelle Anpassungen

#### Fehler des Compilers

Wie bereits in Abschnitt 2.3.3 erwähnt, befindet sich die Version 1.2 von nesC, in der die Funktionalität zum Erstellen binärer Komponenten erstmals enthalten ist, derzeit noch im Alpha-Stadium. In der verwendeten Version sind offenbar noch Fehler enthalten, die einige manuelle Anpassungen am generierten C-Programmcode notwendig machten.

Neben einigen kleineren Schwierigkeiten besteht das Hauptproblem darin, dass der nesC-Compiler momentan noch nicht mit parametrisierbaren Interfaces in Binärkomponenten zurechtkommt. Enthält die Komponentendefinition einer Binärkomponente ein parametrisierbares Interface, so versäumt es der Compiler beim Erstellen des Gesamtprogramms, die Verbindungsfunktionen zu erstellen, die das parametrisierbare Interface der Komponente mit den entsprechenden Interfaces der Partnerkomponenten verbindet. Als Folge kommt es zu Fehlermeldungen durch den Linker, da nicht für alle Funktionsaufrufe der Komponenten entsprechende Funktionen als Gegenstück zur Verfügung stehen.

Das Problem lässt sich manuell auf relativ einfache Weise provisorisch beheben. Es ist lediglich notwendig, die fehlenden Funktionen aus dem Quellcode der unzerteilten Fassung des Programms zu kopieren und in den Quellcode der Verbindungskomponente einzufügen.

Da kein konzeptueller Grund für die mangelnde Unterstützung parametrisierbarer Interfaces besteht, kann man damit rechnen, dass der Fehler bis zur endgültigen Version des nesC-Compilers beseitigt wird.

#### Spezielle Anforderungen der TinyOS-Komponenten

Standardmäßig sind die Komponenten des TinyOS-Systems nicht darauf ausgerichtet, isoliert voneinander in binäre Komponenten kompiliert zu werden. An einigen Stellen

können aus diesem Grund manuelle Anpassungen notwendig sein, damit die Komponenten auch nach der Zerteilung der Anwendung die erwartete Funktionalität bieten.

Konkret aufgetreten ist ein solches Problem im Fall der bereits vorgestellten Timer-Komponente. Standardmäßig ermittelt der nesC-Compiler beim Erstellen einer TinyOS-Anwendung, wie viele Timer-Instanzen innerhalb der Anwendung verwendet werden. Diese Information wird im C-Programmcode der Anwendung dazu verwendet, die Größe der Datenstrukturen des Timers festzulegen. Erstellt man nun jedoch eine Timer-Binärkomponente getrennt von der restlichen Anwendung, so kann der nesC-Compiler natürlich die Anzahl der verwendeten Timer-Instanzen nicht mehr ermitteln und trägt Null als Zahl der Timer-Instanzen ein.

Eine – zumindest provisorische – Lösung dieses Problems bestand darin, die Anzahl der Timer-Instanzen im Quellcode der Timer-Binärkomponente fest auf den Maximalwert von 32 einzustellen. Durch diese Festlegung ist die Timer-Binärkomponente nicht mehr auf die Information zur Anzahl der Timer-Instanzen angewiesen, dafür wird nun standardmäßig eine Datenstruktur für 32 Timer angelegt und an den entsprechenden Stellen des Programms durchlaufen, was bei nur wenigen Verwendungen des Timers sowohl Platz als auch Rechenzeit verschwendet.

Es bleibt offen, ob bei weiteren, in den Experimenten dieser Arbeit nicht eingesetzten Komponenten des TinyOS-Systems ähnliche Probleme auftreten können. Diese Frage ist problematisch, da Fehler dieser Art nicht zwangsläufig durch Fehlermeldungen des Compilers angezeigt werden und dementsprechend nicht auf Anhieb erkennbar sein müssen. Die Lösung liegt jedoch erneut in der Erwartung, dass zukünftig die Standardkomponenten des TinyOS-Systems bereits als Binärkomponenten vorliegen. Bei diesen Standardkomponenten sollten derartige Probleme dann bereits vom Entwickler beseitigt worden sein.

## 6.2 FlexCup-Analyzer

Auf dem zentralen Rechner eines Sensornetzes sind einige Vorbereitungen notwendig, um `FlexCup` für den dynamischen Austausch von Komponenten einsetzen zu können. Vor dem Ausbringen der Sensorknoten muss zusätzlich zum Codeimage der Anwendung auch der Programmcode des `FlexCup`-Bootloaders installiert werden. Außerdem müssen die in Kapitel 5.3 definierten Metadaten, welche die Eigenschaften des installierten Codeimages beschreiben, erstellt und im externen Flash der Sensorknoten abgelegt werden, damit sie für die Bearbeitung eingehender Codeupdates zur Verfügung zu stehen.

Für das Erstellen und Übertragen der Metadaten ist innerhalb von `FlexCup` der `FlexCup`-Analyzer zuständig. Er analysiert den Programmcode der Sensoranwendungen und generiert daraus die spezifizierten Metadaten im Binärformat.

### 6.2.1 Analyse des Programmcodes

Um ein Programm analysieren und die Metadaten berechnen zu können, benötigt der `FlexCup`-Analyzer die ausführbare Datei des Gesamtprogramms, die Objektdateien der Komponenten sowie die Map-Datei des Linkers. In der Map-Datei speichert der AVR-GCC-Linker Informationen über die verwendeten Objektdateien sowie über deren Layout innerhalb der ausführbaren Datei. Um beim Kompilieren des Programms eine solche Map-Datei zu erzeugen, muss dem Linker der Kommandozeilenparameter „`-Map` *Dateiname*“ übergeben werden.

Auf Basis der Daten der Map-Datei ermittelt der `FlexCup`-Analyzer, aus welchen Komponenten die ausführbare Datei aufgebaut ist und wie diese innerhalb der ausführbaren Datei angeordnet sind. Außerdem liest er Informationen zum Layout der Daten des Programms innerhalb des Datenspeichers aus.

Für die Berechnung der Metadaten benötigt der `FlexCup`-Analyzer nun noch verschiedene Informationen der einzelnen Objektdateien und des Gesamtprogramms. Insbesondere die Symboltabelle des Gesamtprogramms und die Relocationtabellen der Objektdateien sind wichtige Datenquellen. Da der `FlexCup`-Analyzer vollständig in Java implementiert ist, wird das Hilfsprogramm „avr-objdump“ aus den GNU Binutils [Frea] verwendet, um diese Informationen aus den Objektdateien auszulesen. Prinzipiell könnte diese Informationsbeschaffung auch direkt über eine C-Funktionsbibliothek, der „Binary File Descriptor Library“ [Fred], erfolgen. Diese Funktionsbibliothek wird auch von den GNU Binutils eingesetzt.

### 6.2.2 Speicherung der Informationen auf dem PC

Um die Objektdateien und das Gesamtprogramm nicht mehrfach analysieren zu müssen, wenn mehr als einmal Metadaten generiert werden oder Auswertungen durchgeführt werden sollen, werden die ermittelten Informationen direkt nach der Analyse in einer XML-Datei abgespeichert. Als Format für den Export der Daten wurde XML ausgewählt, da bereits die XML-Klassen der Standard-Java-API einen sehr komfortablen Export und Import großer Datenmengen variabler Größe ermöglichen. Neben den Metadaten der analysierten Sensoranwendung enthält die XML-Datei auch den vollständigen Programmcode der Anwendung. Auf diese Weise sind alle Informationen in einer einzelnen Datei gespeichert. Dies vereinfacht die Archivierung verschiedener Softwareversionen, was sich bereits während der Entwicklung von `FlexCup` als äußerst nützlich erwiesen hat.

Ein weiterer wichtiger Grund für die Speicherung aller Daten ist die Konsistenz der Bezeichner zwischen unterschiedlichen Versionen eines Programms. Der `FlexCup`-Analyzer erstellt für das Schreiben der Metadaten eine Abbildung zwischen den Namen der Symbole in der Symboltabelle des Programms und den Kurzbezeichnern, die auf den Sensorknoten verwendet werden. Nur durch die Speicherung dieser Informationen kann sicher-

gestellt werden, dass unterschiedliche Versionen eines Programms gleichnamige Symbole auch mit den gleichen Kurzbezeichnern identifizieren.

### 6.2.3 Erzeugen der Metadaten

Nach der Analyse eines Programms kann der `FlexCup`-Analyzer die Metadaten der kompletten Sensoranwendung im Binärformat ausgeben. Hierzu wandelt er die Einträge der Symboltabelle und der Relocationtabellen in das in Abschnitt 5.3 definierte, binäre Format um. Zur Übersetzung der Symbolnamen sowie der Komponentennamen in die entsprechenden Kurzbezeichner kommt eine spezielle Mapping-Komponente zum Einsatz, die dafür sorgt, dass die Abbildung auch über Programmgrenzen hinweg konsistent bleibt.

Der `FlexCup`-Analyzer bietet die Möglichkeit, die Metadaten einer Anwendung zusammen mit einer Kopie des Codeimages im Binärformat zu speichern. Diese Binärdaten sind notwendig, um den externen Flash der Sensorknoten vor deren Verteilung mit den Ausgangsdaten für spätere Codeupdates zu befüllen. Alternativ dazu ist es auch möglich, die Metadaten einer einzelnen Komponente gemeinsam mit den Binärdaten aus der Objektdatei dieser Komponente zu exportieren, um diese anschließend als Codeupdate an die Sensorknoten zu verschicken.

Für Test- und Simulationszwecke können mit dem `FlexCup`-Analyzer auch die Metadaten einer kompletten Sensoranwendung gemeinsam mit einer Kopie des Codeimages sowie den Metadaten und Binärdaten einer einzelnen Komponente geschrieben werden. Dies entspricht dem Abbild des externen Flashspeichers vor dem Beginn eines Codeupdates und kann verwendet werden, um die Arbeit des `FlexCup`-Linkers zu simulieren und zu testen, ohne auf dem Sensorknoten zuvor den Code einer neuen Komponente empfangen zu müssen.

### 6.2.4 Auswertungsfunktionen

Neben dem Erzeugen der Metadaten beherrscht der `FlexCup`-Analyzer noch verschiedene Auswertungsfunktionen, die dem Benutzer genauer Auskunft über die Eigenschaften eines Programms und dessen Metadaten geben. Es lassen sich beispielsweise Informationen zur Größe der Symbol- und Relocationtabellen sowie zu deren Zusammensetzung abrufen.

Es ist auch möglich, zwei unterschiedliche Versionen eines Programms anhand ihrer Metadaten zu vergleichen. Der `FlexCup`-Analyzer gibt dann eine Zusammenfassung der Unterschiede aus und erlaubt so Rückschlüsse auf den zu erwartenden Aufwand des `FlexCup`-Linkers beim Verarbeiten eines entsprechenden Codeupdates.

Die Auswertungsfunktionen des `FlexCup`-Analyzers sind für den Betrieb von `FlexCup` zwar nicht notwendig, sie haben sich aber sowohl bei der Entwicklung als auch bei der Evaluation als sehr hilfreich erwiesen und dürften auch zukünftig für das Verständnis des Systemverhaltens hilfreich sein.

| |
|---|
| 1) Austausch des Programmcodes der Komponente |
| 2) Aktualisieren der Symboltabelle |
| 3) Austausch der Relocationtabelle |
| 4) Überprüfen und Aktualisieren der Referenzen |
| 5) Rücksetzen der Symbolmarkierungen |

Tabelle 6.1: Einzelschritte des `FlexCup`-Linkers

## 6.3 FlexCup-Linker

Der `FlexCup`-Linker realisiert das zentrale Element dieser Arbeit, den dynamischen Austausch von Programmcode durch eine neue Komponente. Aus diesem Grund geht der folgende Abschnitt etwas genauer auf die Implementierung der Einzelschritte ein, die notwendig sind, um die Aufgaben des Linkers zu erfüllen. Im darauf folgenden Abschnitt 6.3.2 werden anschließend einige weitere wichtige Details der Implementierung angesprochen.

### 6.3.1 Einzelschritte

Die Arbeit des `FlexCup`-Linkers lässt sich in fünf Einzelschritte aufgliedern, die im Folgenden genauer beschrieben werden. Ein Überblick über die Einzelschritte ist in Tabelle 6.1 zu sehen.

#### Austausch des Programmcodes der Komponente

Im ersten Teilschritt wird der alte Programmcode der Komponente gegen den neu empfangenen Codeabschnitt ausgetauscht. Zu diesem Zweck muss zunächst identifiziert werden, welche der Komponenten des Programms auszutauschen ist. Dies geschieht durch einen Vergleich des Bezeichners der neuen Komponente mit den Bezeichnern der momentan installierten Komponenten.

Nachdem die auszutauschende Komponente gefunden ist, vergleicht man die Codegröße der alten und der neuen Komponente anhand der Informationen aus den Metadaten beider Komponenten. Es kommt zu folgender Fallunterscheidung:

- **Codegröße identisch:** Entspricht die Codegröße der alten Komponente der Codegröße der neuen Komponente, so kann der alte Programmcode direkt mit dem neuen Programmcode überschrieben werden.
- **Codegröße angewachsen:** Falls die neue Komponente größer ist als die zu ersetzende Komponente, so müssen zunächst alle Komponenten, die im Programmspeicher dahinter liegen, nach hinten verschoben werden, um Platz für die neue Komponente zu schaffen. Alle Funktionssymbole, die in den verschobenen Komponenten

liegen, müssen entsprechend ihrer neuen Position angepasst werden. Anschließend kann der neue Programmcode an die Position der alten Komponente geschrieben werden.

- **Codegröße geschrumpft:** Ist die Größe des Komponentencodes kleiner geworden, so müssen die dahinter liegenden Komponenten analog zum vorigen Fall nach vorne verschoben werden.

In Abschnitt 6.6 wird eine Optimierung beschrieben, die es in den meisten Fällen erlaubt, auch im Falle von Größenänderungen auf das Verschieben von Komponenten zu verzichten.

Die neue Komponente bringt neben dem geänderten Programmcode eventuell auch Änderungen am *.data*-Abschnitt der Komponente (siehe Kapitel 2.4.1) und neue Initialisierungsdaten für diesen Abschnitt mit sich. Hat sich die Größe des *.data*-Abschnitts der Komponente geändert, so müssen dahinter liegende Datenobjekte zunächst entsprechend verschoben werden. Anschließend werden die alten Initialisierungsdaten der Komponente mit den Initialisierungsdaten der neuen Komponente überschrieben.

### Aktualisieren der Symboltabelle

Nach dem Austausch des Programmcodes folgt das Aktualisieren der Symboltabelle. Hierbei werden, getrennt nach Funktions- und Objektsymbolen, die aktuelle Symboltabelle sowie die mitgelieferten Symbolinformationen der neuen Komponente parallel durchlaufen, die einzelnen Einträge miteinander verglichen und gegebenenfalls aktualisiert.

Funktionssymboleinträge müssen angepasst werden, falls sich die Startadresse der Funktion innerhalb der Komponente geändert hat. Außerdem ist es möglich, dass neue Funktionssymbole hinzukommen, falls die neue Komponente zusätzliche Funktionen unterstützt. Diese neuen Funktionssymbole müssen an der entsprechenden Stelle in die Symboltabelle des Programms eingefügt werden.

Bei den Objektsymboleinträgen kann es zu ähnlichen Änderungen kommen: Bei Symbolen innerhalb des *.data*-Abschnitts einer Komponente kann es zu Adress- sowie Größenänderungen kommen, bei Symbolen des *.bss*-Abschnitts sind nur Änderungen der Größe möglich. Größenänderungen wirken sich dabei auch auf andere Objektsymbole aus, die teilweise innerhalb des Datenspeichers verschoben werden müssen, um Platz für die neue Größe des Objekts zu schaffen. Kommen neue Objektsymbole hinzu, so müssen diese in die Symboltabelle eingefügt werden. Für neue Symbole des *.data*-Abschnitts wurde die Größe des *.data*-Abschnitts bereits im vorigen Schritt angepasst, Speicherplatz für neue Symbole des *.bss*-Abschnitts kann am Ende des *.bss*-Abschnitts reserviert werden.

Um später bei der Verwendung der Symboltabelle erkennen zu können, bei welchen Einträgen der Symboltabelle es im Rahmen des Updates zu Änderungen gekommen ist, werden die geänderten Symbole markiert. Für diese Markierung wird das höchstwerti-

ge Bit der Symbolbezeichner verwendet. Ist dieses Bit gesetzt, so signalisiert dies dem Linker, dass der Adresseintrag des Symbols geändert wurde.

Nicht implementiert ist bisher das Entfernen nicht mehr benötigter Objektsymbole aus der Symboltabelle. Es handelt sich dabei um eine durchaus schwierige Aufgabe, da nur durch ein Untersuchen der Einträge aller Relocationtabellen eindeutig festgestellt werden kann, ob ein Symbol noch referenziert und dementsprechend noch benötigt wird. In einigen Fällen kann es auch sinnvoll sein, nicht mehr benötigte Symbole zunächst nicht aus der Symboltabelle zu entfernen. Dies zahlt sich beispielsweise aus, falls die alte Komponente zu einem späteren Zeitpunkt wieder installiert wird und die Symbole erneut benötigt werden. In diesem Fall ist das Aktualisieren eines Eintrags mit deutlich geringerem Aufwand verbunden als das Einfügen eines neuen Symbols. Auf der anderen Seite belegen solche Symbole sowohl Speicherplatz in der Symboltabelle als auch im Datenspeicher, in dem sie Platz für das repräsentierte Objekt freihalten. Zukünftige Arbeiten sollten demnach eine Lösung entwickeln, bei der nicht mehr benötigte Einträge zunächst in der Symboltabelle verbleiben und dann, entweder in regelmäßigen Abständen, oder bei auftretender Speicherknappheit, eine Art Garbage Collection durchgeführt wird, die alle unnötigen Einträge aus der Symboltabelle entfernt.

### Austausch der Relocationtabelle

Im Vergleich zum Aktualisieren der Symboltabelle ist der Austausch der Relocationtabelle eine einfachere Aufgabe. Die neue Komponente bringt ihre Relocationtabelle bereits im korrekten Format mit, so dass die alte Tabelle nur mit den Daten der neuen Relocationtabelle überschrieben werden muss.

Um mögliche Größenänderungen der Relocationtabellen abfangen zu können, wird bereits bei der Vorbereitung der Ursprungs-Metadaten eine gewisse Menge Speicherplatz zwischen den einzelnen Relocationtabellen freigelassen. Das Kopieren der neuen Relocationtabelle gestaltet sich demnach nur dann aufwendiger, wenn die Größe der neuen Relocationtabelle auch über diesen Puffer hinaus anwächst. In diesem Fall müssen dahinter liegende Daten verschoben werden, um Platz für die neue Relocationtabelle zu schaffen.

### Überprüfen und Aktualisieren der Referenzen

Die Hauptaufgabe des `FlexCup`-Linkers ist das Anpassen der Referenzen, die aufgrund eines Updates einer Komponente geändert werden müssen. Dies geschieht im Rahmen des dritten Teilschritts. Der Linker muss dabei die vier Referenztypen behandeln, die in den Relocationtabellen der Komponenten vorkommen: **„Internal Call“-Referenzen**, **„External Call“-Referenzen**, **„Storage Relocation“-Referenzen ohne Offset** sowie **„Storage Relocation“-Referenzen mit Offset**.

Eine Anpassung der **„Internal Call“-Referenzen** ist nur für Komponenten notwendig, die innerhalb des Programmspeichers verschoben wurden, sowie grundsätzlich für die neue Komponente. Im Rahmen des ersten Schritts des `FlexCup`-Linkers mussten andere Komponenten nur dann verschoben werden, falls sich die Codegröße der Komponente geändert hat.

Die „Internal Call“-Referenzen beziehen sich auf die Startadresse der jeweiligen Komponente, sind also immer der Form *.text + Offset*. Für die Aktualisierung der „Internal Call“-Referenzen werden dementsprechend die Startadresse der jeweiligen Komponenten, die als Teil der Komponenteninformationen in den Metadaten gespeichert ist, und der lokale Offset der jeweiligen Referenz benötigt. Der Linker berechnet dann die Zieladresse als Summe aus Startadresse der Komponente und lokalem Offset der jeweiligen Referenz und trägt diesen Wert an die Quelladresse des internen Sprungs ein.

Um die **„External Call“-Referenzen** anpassen zu können, muss der Linker zunächst die Zielsymbole der Relocationeinträge in der Symboltabelle finden. Dabei kommt ihm zugute, dass sowohl die Symboltabelle als auch die „External Call“-Sprungziele in der Relocationtabelle aufsteigend nach Symbolbezeichnern sortiert sind. Die Adressinformation des Zielsymbols gibt dann das Sprungziel des Relocationeintrags vor und kann vom `FlexCup`-Linker an der entsprechenden Stelle des Programmcodes eingetragen werden.

Für die Bearbeitung der **„Storage Relocation“-Referenzen ohne Offset** muss der Linker zunächst erneut das Zielsymbol des jeweiligen Relocationeintrags in der Symboltabelle identifizieren. Anschließend kann er auf Basis der Symboladresse des Zielsymbols die Referenzen im Programmcode aktualisieren. In welcher Form die Adressinformation dabei in den Programmspeicher zu schreiben ist, hängt vom Typ der Relocation ab.

Um Platz bei der Speicherung der Relocationtabellen zu sparen, sind die „Storage Relocation“-Referenzen zuerst nach dem Typ der Relocation und dann nach ihrem Zielsymbol sortiert, wodurch die Typinformation und der Symbolbezeichner nicht für jeden Relocationeintrag wiederholt werden müssen (siehe auch Abschnitt 5.3.3). Beim Durchlaufen der „Storage Relocation“-Referenzen birgt dies jedoch den Nachteil in sich, dass die Symboltabelle des Programms mehrfach durchlaufen werden muss, da die Symbolbezeichner nur innerhalb des gleichen Relocationtyps aufsteigend sortiert sind.

Zu beachten ist zudem, dass sich „Storage Relocation“-Referenzen nicht nur auf Objektsymbole, sondern in verschiedenen Fällen auch auf Funktionssymbole beziehen können. Dies ist beispielsweise der Fall, wenn sich Referenzen auf statische, konstante Daten beziehen, die vom Compiler im Programmspeicher platziert wurden. Abhängig vom Typ der Relocation müssen deswegen die Funktionssymboleinträge anstatt der Objektsymboleinträge der Symboltabelle durchsucht werden.

Die Bearbeitung der **„Storage Relocation“-Referenzen mit Offset** entspricht weitgehend der Bearbeitung der „Storage Relocation“-Referenzen ohne Offset. Der einzige Unterschied liegt darin, dass zu der Zieladresse des Symbols noch der in der Referenz spezifizierte Offset hinzugezählt werden muss.

Bei der Bearbeitung der „External Call“-Referenzen sowie der „Storage Relocation“-Referenzen mit oder ohne Offset kommen die bereits erwähnten Markierungen der Einträge in der Symboltabelle zum Einsatz. Nur für den Fall, dass das Symbol als geändert markiert ist, müssen die Referenzen im Programmcode angepasst werden, ansonsten ist der eingetragene Wert noch korrekt. Eine Ausnahme stellt die Relocationtabelle der neuen Komponente dar: Die Referenzen des neuen Programmcodes wurden noch nicht von einem Linker gesetzt und sind dementsprechend nur mit Standardwerten belegt. Aus diesem Grund müssen hier die Zieladressen aller Einträge der Relocationtabelle berechnet und die Referenzen entsprechend gesetzt werden, unabhängig davon, ob die referenzierten Symbole geändert wurden oder nicht.

#### Rücksetzen der Symbolmarkierungen

Als letzter Schritt des `FlexCup`-Linkers müssen die Markierungen der Funktions- und Objektsymboleinträge in der Symboltabelle zurückgesetzt werden, um wieder eine neutrale Ausgangssituation für zukünftige Durchläufe des `FlexCup`-Linkers herzustellen. Hierfür werden beide Teile der Symboltabelle durchlaufen und, wo nötig, die Symbolbezeichner entsprechend korrigiert.

### 6.3.2 Weitere Implementierungsdetails

#### Zugriff auf den externen Datenspeicher

Verteilt über den ganzen Programmcode des `FlexCup`-Linkers wird immer wieder lesend und schreibend auf den externen Flashspeicher der Mica2-Sensorknoten zugegriffen. Durch diese sehr große Zahl der Zugriffe hat die Implementierung der Zugriffsfunktionen einen entscheidenden Einfluss auf die Leistung des `FlexCup`-Linkers.

Die C-Funktionsbibliothek für AVR-Prozessoren *AVR Libc* [AVR] bietet selbst keine Unterstützung für Zugriffe auf den externen Flash, sie stellt lediglich Funktionen für den Zugriff auf das interne Configuration EEPROM sowie für das Schreiben des Programmspeichers der AVR-Prozessoren zur Verfügung. Die Implementierung des Zugriffs auf den externen Flash bleibt somit dem jeweiligen Anwendungsprogramm, in diesem Fall also dem `FlexCup`-Linker, überlassen.

Im Falle der Mica2-Sensorknoten wird der externe Flashspeicher durch den Speicherbaustein AT45DB041B [Atm04a] von Atmel realisiert, der seriell an den Prozessor angebunden ist (siehe auch Abschnitt 2.1.2). Es existiert also jeweils nur ein einzelner Kanal, um sequenziell Bits an den Flashbaustein zu übertragen beziehungsweise um sequentiell Bits von ihm zu empfangen. Bedingt durch diesen seriellen Zugriff muss vor jedem Zugriff zunächst eine Initialisierungssequenz an den Flashbaustein übertragen werden. Sie besteht aus einem Befehlscode, der den Typ der gewünschten Operation spezifiziert, sowie Adressdaten, wie der Adresse der Page und der Byteadresse innerhalb der Page.

Abhängig vom übertragenen Befehlscode erwartet der Flashbaustein in der Folge unterschiedliche Adressdaten sowie in manchen Fällen zusätzliche „Don't Care"-Bytes, bevor anschließend – erneut sequentiell – Nutzdaten in den Flashspeicher geschrieben oder aus dem Flashspeicher gelesen werden können.

Beim Entwurf der Funktionen für den Zugriff auf den externen Datenspeicher galt es zwischen zwei in Konflikt stehenden Zielen abzuwägen: Auf der einen Seite ist es wünschenswert, den Zugriff auf den Datenspeicher für die anderen Programmteile möglichst transparent zu gestalten und die Implementierungsdetails des eigentlichen Zugriffs in entsprechenden Funktionen zu kapseln. Da die Zugriffe auf den externen Datenspeicher jedoch sehr teuer sind, ist es andererseits wichtig, die Speicherzugriffe möglichst genau steuern zu können, um das Wissen der anderen Programmteile für die Steuerung des Speicherzugriffs nutzen zu können.

Der AT45DB041B Speicherbaustein besitzt zwei SRAM-Datenpuffer, die jeweils eine komplette Page der Größe 264 Byte aufnehmen können. Beim Zugriff auf eine Adresse im externen Speicher besteht demnach entweder die Möglichkeit, zunächst die dazugehörige Page in einen der zwei Datenpuffer zu laden, oder es kann – unter Umgehung der Pufferfunktionalität – auch direkt auf die Adresse zugegriffen werden. Für die Entscheidung, welche Pages vor dem Zugriff in einem Datenpuffer abgelegt werden sollen und vor allem welche Page hierfür aus dem Datenpuffer gelöscht werden soll, wäre möglichst viel Anwendungswissen über die darauf folgenden Speicherzugriffe wünschenswert.

Für Standardzugriffe auf den externen Flash wurde innerhalb des `FlexCup`-Linkers eine Schnittstelle implementiert, die der Schnittstelle der AVR Libc-Bibliothek für den Zugriff auf das interne EEPROM stark ähnelt. Es werden eine Funktion zum Lesen und eine Funktion zum Schreiben eines einzelnen Bytes (**external_flash_read_byte** und **external_flash_write_byte**) sowie analog dazu eine Funktion zum Lesen und eine Funktion zum Schreiben eines Words (**external_flash_read_word** und **external_flash_write_word**) bereitgestellt. Als Parameter erwarten die Funktionen die Byte-Adresse der zu lesenden bzw. der zu schreibenden Daten.

Bei jedem Aufruf einer dieser Funktionen wird für die übergebene Speicheradresse zunächst die zugehörige Pageadresse und die Byteadresse innerhalb der Page berechnet. Liegt die Page bereits in einem der beiden Puffer bereit, so kann der Zugriff auf die gewünschte Byteadresse direkt erfolgen. Ist dies nicht der Fall, so muss die angeforderte Page zunächst in den Puffer geladen werden und dabei eine der beiden momentan gepufferten Pages verdrängen. Als Ersetzungsstrategie kommt eine Trivialversion des Least-Recently-Used-Algorithmus zum Einsatz: Ersetzt wird der Datenpuffer, dessen letzter Zugriff länger zurückliegt. Sobald auf eine Adresse des Datenpuffers schreibend zugegriffen wird, wird der Puffer als geändert markiert und er muss beim nächsten Ersetzen oder spätestens am Ende des Programms in den externen Flashspeicher zurückgeschrieben werden.

Der Aufwand für den Zugriff auf ein Byte oder ein Word im externen Flash unterscheidet sich also abhängig davon, ob die angeforderten Daten bereits im Puffer bereitliegen und ob

vor dem Zugriff andere Daten zunächst in den externen Flash zurückgespeichert werden müssen. Konkret gibt es die folgenden drei Fälle:

1. Die angeforderten Daten liegen bereits in einem der zwei Puffer und können direkt zugegriffen werden. Es fällt nur der Aufwand für den Lese- beziehungsweise den Schreibzugriff an.

2. Die Daten sind in keinem der beiden Puffer enthalten, können jedoch direkt in den Puffer geladen werden, da ein Zurückspeichern des bisherigen Pufferinhalts nicht notwendig ist. Es fällt Aufwand für das Laden des Puffers sowie für den Lese- beziehungsweise Schreibzugriff auf den Puffer an.

3. Die Daten sind in keinem der beiden Puffer enthalten, die Daten im Puffer müssen vor dem Ersetzen zurückgeschrieben werden. Es fällt hierbei Aufwand für das Rückschreiben des Puffers, für das Laden des Puffers mit den neuen Daten sowie für den Lese- beziehungsweise Schreibzugriff an.

In allen Fällen unvermeidbar ist der Aufwand für den Lese- oder Schreibzugriff auf den Puffer: Zunächst müssen der Befehlscode, die Byteadresse und zusätzliche „Don't Care"-Bytes an den Dateneingang des Flashbausteins übertragen werden, bevor die Nutzdaten geschrieben oder gelesen werden können. Für das Lesen eines oder mehrerer aufeinander folgender Bytes aus dem Puffer des externen Flash umfasst diese Initialisierungssequenz fünf zu schreibende Bytes; um das Schreiben in den Puffer einzuleiten, genügt eine Initialisierungssequenz mit einem Umfang von vier Bytes.

Beim Transfer großer zusammenhängender Datenblöcke ist die Verwendung der bereits beschriebenen Zugriffsfunktionen ineffizient, da das Lesen und Schreiben der Daten in kleinen Blöcken von maximal zwei Bytes Größe geschieht. Aus diesem Grund wurde mit **copyData** eine weitere Funktion implementiert, die für das Kopieren größerer Datenblöcke von einer Adresse im externen Flash an eine andere Adresse im externen Flash verwendet werden kann. Die Funktion liest und schreibt möglichst große Datenblöcke am Stück, so dass die Anzahl der benötigten Initialisierungssequenzen minimiert wird.

### Verwendung statischer Objektsymbole

Auf bestimmte Objektsymbole muss der `FlexCup`-Linker während des Programmablaufs wiederholt lesend und schreibend zugreifen. Es handelt sich dabei um *_data_start* und *_data_end*, die den Beginn und das Ende des *.data*-Abschnitts im Datenspeicher markieren, um *_bss_start* und *_bss_end*, welche die Start- und Endadresse des *.bss*-Abschnitts im Datenspeicher repräsentieren, sowie um *_data_load_start* und *_data_load_end*, die für die Start- und Endadresse des Initialisierungsabschnittes der *.data*-Symbole im Programmspeicher stehen.

Da alle sechs Symbole in jedem Programm enthalten sein müssen, bekommen sie einen festen Symbolbezeichner am unteren Ende des Adressraums zugewiesen und bilden ge-

| Symbolname | Symbolbezeichner |
|---|---|
| _data_start | 0 |
| _data_end | 1 |
| _bss_start | 2 |
| _bss_end | 3 |
| _data_load_start | 4 |
| _data_load_end | 5 |

Tabelle 6.2: Statische Objektsymbole

meinsam die Menge der statischen Objektsymbole. Tabelle 6.2 gibt einen Überblick über die Symbolbezeichner der statischen Objektsymbole.

Der Vorteil dieses Vorgehens liegt darin, dass der `FlexCup`-Linker nun direkt auf den Inhalt dieser Symbole zugreifen kann, beispielsweise um Änderungen an der Größe des *.bss*-Abschnitts vermerken zu können. Würden die Symbole in jedem Programm andere Symbolbezeichner besitzen, so müsste diese Zuordnung dem `FlexCup`-Linker gesondert mitgeteilt werden, um ihm Anpassungen der Symbolwerte zu ermöglichen.

Da die statischen Objektsymbole weiterhin Teil der Symboltabelle sind, ist bei der Bearbeitung der Relocationtabellen keine Sonderbehandlung notwendig. Referenzen auf die statischen Objektsymbole können genauso aufgelöst werden wie Referenzen auf andere Objektsymbole.

### Überprüfen der Korrektheit

Ein wichtiger Teil der Implementierung des `FlexCup`-Linkers bestand in der Überprüfung der korrekten Arbeitsweise des Linkers. Es musste sichergestellt werden, dass die Semantik der durch den `FlexCup`-Linker erstellen Programme exakt der Semantik der Originalprogramme entspricht, die der Linker des AVR-GCC-Pakets erstellt.

Die erste wichtige Erkenntnis bestand darin, dass die Ausgaben der beiden Linker nur in wenigen Fällen komplett identisch sind, ohne dass dies zwangsläufig auf Fehler des `FlexCup`-Linkers hinweist. Da der `FlexCup`-Linker Änderungen inkrementell in den bestehenden Programmcode integriert, erzeugt er in vielen Fällen ein anderes Layout der Symbole im Datenspeicher als der AVR-GCC-Linker, der bei jedem Durchgang das Speicherlayout der Symbole komplett neu berechnet. So platziert der `FlexCup`-Linker neue Symbole des *.bss*-Abschnitts immer am Ende des Abschnitts, um andere Symbole nicht verschieben zu müssen. AVR-GCC kennt dagegen keine Unterscheidung neuer und bestehender Symbole.

Um das Ergebnis des `FlexCup`-Linkers und des AVR-GCC-Linkers vergleichen zu können, wurde jeweils der reine Binärcode der bearbeiteten Programme extrahiert. Mit Hilfe der GNU Diffutils [Freb] konnte anschließend ermittelt werden, an welchen Stellen Unterschiede auftraten. Auf Basis der Relocationtabellen des Programms ließ sich nun

ermitteln, bei welchen Relocationeinträgen die beiden Linker auf unterschiedliche Ziele verweisen. Da die Symboltabellen mit den jeweiligen Zieladressen beider Programmversionen vorlagen, konnte das korrekte Verhalten so durch einen Vergleich aller Unterschiede überprüft werden.

## 6.4 FlexCup-Bootloader

Der `FlexCup`-Bootloader hat eine relativ einfache Aufgabe zu erfüllen. Er muss den vom `FlexCup`-Linker im externen Flash vorbereiteten neuen Programmcode in den Programmspeicher des Prozessors übertragen und anschließend einen Neustart des Systems durchführen. Es existieren für den ATmega128L bereits zahlreiche Bootloader-Implementierungen mit ähnlichen Aufgaben, beispielsweise für Deluge [HC04] und Xnp [Cro03b]. Unterschiede zwischen den Bootloadern bestehen nur in Implementierungsdetails, wie dem Zugriff auf den externen Flash, sowie in Zusatzaufgaben, die die Bootloader in den verschiedenen Systemen zu erfüllen haben.

Der `FlexCup`-Bootloader muss zunächst auf Basis der Metadaten die Startadresse sowie die Größe der Kopie des Programmspeichers im externen Flash ermitteln. Anschließend liest er jeweils 256 Byte große Abschnitte der Programmdaten in seinen Speicher und schreibt diese dann in den Programmspeicher des Prozessors. Da der Bootloader dabei auf die Programmdaten im externen Flash nicht schreibend zugreift, kann er sie unter Umgehung der Flashpuffer direkt in den RAM einlesen.

Sobald die Programmdaten vollständig in den Programmspeicher übertragen sind, führt der Bootloader einen Sprung an die Adresse 0 im Programmspeicher durch und leitet dadurch einen Neustart des frisch installierten Systems ein.

Um den geänderten Programmcode der Anwendung in den Programmspeicher übertragen zu können, benötigt der Bootloader Schreibzugriff auf den Programmspeicher. Wie in Abschnitt 2.1.1 ausführlich dargelegt, ist ein solcher Schreibzugriff auf den Programmspeicher des ATmega128L-Prozessors nur aus dem so genannten Bootloader-Abschnitt des Programmspeichers möglich. Dessen Größe ist zwar konfigurierbar, jedoch auf maximal 8 Kilobyte beschränkt. Da der Programmcode des `FlexCup`-Linkers alleine bereits deutlich mehr als diese 8 Kilobyte umfasst, können der Linker und der Bootloader nicht in einem gemeinsamen Programm integriert und im Bootloader-Abschnitt platziert werden. Es muss stattdessen sichergestellt werden, dass sich zumindest der Programmcode des Bootloaders innerhalb des Bootloader-Abschnitts befindet. Der `FlexCup`-Linker kann dagegen an einer beliebigen anderen Position innerhalb des Programmspeichers untergebracht werden, zum Beispiel direkt vor dem Bootloader-Abschnitt.

## 6.5 Simulation des FlexCup-Linkers auf dem PC

Die Implementierung des `FlexCup`-Linkers für Mica2-Motes wurde in der Programmiersprache C geschrieben und mit AVR-GCC kompiliert. Nach einigen wenigen, einfachen Änderungen ließ sich das Programm auch für den PC kompilieren. Hierbei kam erneut der GCC-Compiler zum Einsatz, in diesem Fall eine Version für Cygwin.

Da auf dem PC kein externer Flashspeicher zur Verfügung steht, musste die Komponente für den Speicherzugriff auf dem PC neu geschrieben werden. Der externe Flash wird in diesem Fall durch eine Binärdatei simuliert: Ein Zugriff auf die Adresse $X$ innerhalb des externen Flash wird auf dem PC durch einen Zugriff auf die Position $X$ innerhalb der Binärdatei ersetzt. Um einen realen Programmablauf simulieren zu können, muss nun lediglich der entsprechende Startzustand des Flashspeichers in eine Binärdatei gespeichert und der PC-Zugriffskomponente übergeben werden.

Obwohl der Zugriff auf den externen Flash auf dem PC durch einem simplen Dateizugriff ersetzt wurde, implementiert die PC-Komponente doch sämtliche internen Berechnungen, die für den Zugriff auf den Flashspeicher benötigt werden. Bei jedem Speicherzugriff wird so aus der Speicheradresse die jeweilige Page- und die Byteadresse berechnet und die PC-Komponente überwacht, ob die angeforderte Page bereits in einem der Datenpuffer bereit liegt. Muss eine neue Page in einen der Puffer nachgeladen werden, so überprüft auch die PC-Komponente, ob der Inhalt des Puffers zuvor zurück in den Speicher geschrieben werden muss. Der Vorteil dieses Vorgehens liegt darin, dass sich auf diese Weise auch das Verhalten des Programms bezüglich der Zugriffe auf den externen Flash detailliert auf dem PC nachvollziehen und simulieren lässt.

Eine PC-Version des `FlexCup`-Linkers ist nützlich, um den Ablauf der Codeupdates genauer analysieren zu können. Die Mica2-Motes selber bieten hierfür nur eine sehr geringe Zahl an Interaktionsmöglichkeiten: Man kann sich einerseits bestimmte Ereignisse durch ein Blinken der integrierten Leuchtdioden anzeigen lassen, andererseits kann man das Ergebnis der Berechnungen aus dem externen Flash zurück auf den PC laden und mit einem vorgegebenen Soll-Ergebnis vergleichen. Detaillierte Auswertungen des Programmablaufs sind auf dem Mote nur sehr aufwändig möglich, auf dem PC lassen sich dagegen problemlos eine Vielzahl von Zählerwerten erfassen und dem Benutzer ausgeben.

Die folgenden Zählerwerte wurden bei Simulationsdurchläufen des `FlexCup`-Linkers auf dem PC erfasst:

- Anzahl der Lesezugriffe auf einen der Puffer des externen Flash
- Anzahl der Schreibzugriffe auf einen der Puffer des externen Flash
- Anzahl der „Speichere Puffer in den Flashspeicher“-Operationen
- Anzahl der „Lade Puffer aus dem Flashspeicher“-Operationen
- Anzahl der aus dem Puffer des externen Flash gelesenen Bytes

- Anzahl der in den Puffer des externen Flash geschriebenen Bytes
- Anzahl der „Lade Page direkt in den Mote-Speicher“-Operationen

Da die Zugriffe auf den externen Flash für einen bedeutenden Anteil des Zeit- und Energiebedarfs des `FlexCup`-Linkers verantwortlich sind, liefert die Auswertung dieser Zählerwerte wichtige Informationen für die Analyse des `FlexCup`-Linkers. Die Betrachtung der Zählerwerte einzelner Betriebsphasen hilft dabei, Programmabschnitte zu identifizieren, bei deren Ausführung besonders viel Aufwand anfällt. Dies konnte während der Implementierung des `FlexCup`-Linkers unter anderem bei der Suche nach Optimierungspotenzialen eingesetzt werden.

Da der Sourcecode beider Versionen bis auf wenige Teile identisch ist, kann man von den Ergebnissen der Auswertungen der PC-basierten Version auch direkt auf das Verhalten der Anwendung auf Mica2-Motes schließen. Dass sich beide Versionen auf logischer Ebene tatsächlich identisch verhalten, konnte zudem anhand zahlreicher Beispiele verifiziert werden: Ein Vergleich des Inhalts des Flashspeichers mit dem Inhalt der PC-Datei am Ende des Codeupdates ergab in allen untersuchten Fällen, dass die Berechnungen auf dem Mote und die Berechnungen auf dem PC identische Ergebnisse erzeugen.

## 6.6 Optimierungen

In diesem Abschnitt werden einige Optimierungen an der Implementierung von `FlexCup` präsentiert, durch deren Umsetzung die Effizienz des Codeaustauschs verbessert werden konnte. Alle im Folgenden geschilderten Optimierungen beziehen sich dabei auf Arbeitsschritte des `FlexCup`-Linkers. Dies hängt damit zusammen, dass an dieser Stelle das größte Einsparungspotential besteht: Verbesserungen der Implementierung des `FlexCup`-Bootloaders sind aufgrund seiner Einfachheit nur schwer zu finden. Verbesserungen am `FlexCup`-Analyzer führen dagegen zu keinen Einsparungen, da der zentrale Rechner keinen besonderen Ressourcenbeschränkungen unterliegt.

In Abschnitt 6.6.5 werden schließlich einige Ideen für mögliche weitere Optimierungen beschrieben, die in der vorliegenden Implementierung bisher noch nicht umgesetzt wurden.

### 6.6.1 Pufferung der Symboltabelle

Im Rahmen des Programmablaufs des `FlexCup`-Linkers muss an vielen Stellen auf die Funktions- und die Objektsymboleinträge in der Symboltabelle des Sensorknotens zugegriffen werden. Da jeder Zugriff auf die Symboltabelle immer auch mit einem Zugriff auf den externen Flash verbunden ist und diese Zugriffe mit hohen Kosten verbunden sind, bietet sich hier ein besonderes Optimierungspotential.

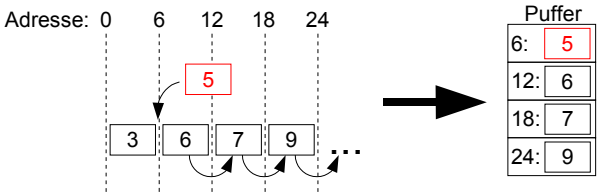


Abbildung 6.3: Einfügen eines Symbols in die Symboltabelle ohne und mit Puffer

Um die Kosten für Zugriffe auf den externen Flash zu senken, wurde folgende Optimierung implementiert: Änderungen an den Einträgen der Symboltabelle werden nicht mehr direkt in den externen Flash geschrieben, sondern zunächst im Hauptspeicher des Prozessors gepuffert. Erst vor dem Beenden des `FlexCup`-Linkers werden dann alle Änderungen am Stück zurück in den externen Flash übertragen. Da der `FlexCup`-Linker nur einen kleinen Teil der vorhandenen vier Kilobyte RAM benötigt, kann für diese Optimierung ein Puffer der Größe 3000 Bytes im Hauptspeicher reserviert werden.

Das Puffern der Änderungen an der Symboltabelle ist besonders beim **Einfügen von neuen Funktions- oder Objektsymbolen** sowie beim **Durchlaufen der Relocationtabellen** vorteilhaft.

Ohne Verwendung eines Puffers im RAM ist das **Einfügen von neuen Funktions- oder Objektsymbolen** eine besonders teure Operation, da die Sortierung der Symboltabelle nach Symbolbezeichnern auch nach dem Einfügen unbedingt erhalten bleiben muss. Aus diesem Grund müssen alle – bezogen auf den Bezeichner dahinter liegenden – Symbole um eine Position nach hinten verschoben werden, um das neue Symbol an der korrekten Position platzieren zu können. Dieser Vorgang ist beispielhaft auf der linken Seite von Abbildung 6.3 dargestellt: Um das Symbol mit dem numerischen Bezeichner 5 in die Symboltabelle einfügen zu können, müssen zunächst die Symbole 6, 7 und 9 jeweils um eine Position nach hinten verschoben werden. Ist die Symboltabelle auf mehrere Pages des externen Flash verteilt, so müssen während des Verschiebeprozesses unter Umständen mehrmals die Flashpuffer zurückgespeichert und frisch geladen werden. Werden zudem im Laufe der Zeit mehrere Symbole eingefügt, so muss der Verschiebeprozess für jedes Symbol wiederholt werden, so dass sich die Anzahl der benötigten Lese- und Schreibzugriffe auf den externen Flash vervielfacht.

Werden Änderungen der Symboltabelle im RAM gepuffert, so wird das neue Symbol in den Puffer geschrieben. Da sich die Adressen aller dahinter liegenden Symbole ändern, müssen sie auf jeden Fall auch in den Puffer übertragen werden. Dieser Vorgang ist auf der rechten Seite von Abbildung 6.3 dargestellt. Für jedes in den Puffer eingetragene Symbol wird gespeichert, an welcher Adresse des externen Flash es zukünftig gespeichert werden soll. Unabhängig von der Anzahl der in die Tabelle eingefügten Symbole müssen die Änderungen nur einmal am Stück zurück in den externen Flash geschrieben werden. Dies erfordert erheblich weniger Lese- und Schreibzugriffe auf den externen Flash als das

wiederholte Verschieben von Symbolen beim normalen Einfügen.

Beim **Durchlaufen der Relocationtabellen** während des Überprüfens und Aktualisierens der Referenzen kann ebenfalls von der Pufferung der Symboleinträge profitiert werden. Der Großteil der Relocationeinträge muss nur angepasst werden, wenn die referenzierten Symbole geändert wurden. Durch die Pufferung liegen genau diese geänderten Symbole bereits im Hauptspeicher des Prozessors bereit, so dass Zugriffe auf den externen Flash nur noch in wenigen Fällen notwendig sind.

Bei sehr großen Programmen mit entsprechend großen Symboltabellen kann es passieren, dass die reservierten 3000 Bytes Speicherplatz im RAM nicht mehr für die Zwischenspeicherung aller Änderungen an den Funktionssymbolen sowie zur Speicherung aller Objektsymbole ausreichen. In diesem Fall ist es erforderlich, die Änderungen schon zu einem früheren Zeitpunkt zurück in den externen Flash zu übertragen, um dadurch wieder Speicherplatz im RAM freizugeben. Dies hat entsprechende Auswirkungen auf die Verwendung der Symbolinformationen in den späteren Phasen: Zum einen stehen beim Durchlaufen der Relocationtabellen nicht alle geänderten Symbole im RAM zur Verfügung, so dass die Daten im RAM und die Daten im externen Flash parallel durchlaufen werden müssen. Zum anderen sind geänderte Symbole, die vorzeitig in den externen Flash zurückgeschrieben werden müssen, noch als geändert markiert. Diese Markierungen müssen am Ende des Programms wieder aufgehoben werden, dies erfordert zusätzliche Lese- und Schreibzugriffe auf den externen Flash.

Um eine solche Sonderbehandlung großer Symboltabellen nach Möglichkeit zu vermeiden, wäre es wünschenswert, über die vorhandenen 3000 Bytes hinaus weiteren Speicherplatz für die Pufferung von Symboldaten zur Verfügung stellen zu können. Eine Möglichkeit hierfür schien zunächst in der Verwendung des Prozessor-internen Configuration EEPROM zu liegen, das weitere 4096 Bytes Speicherplatz zur Verfügung stellt. Das Configuration EEPROM ist jedoch von seinen Eigenschaften her nicht auf regelmäßige Schreibzugriffe hin ausgelegt, sondern ist für die Abspeicherung nur selten geänderter Konfigurationsdaten gedacht. Während das Lesen eines Bytes aus dem EEPROM dementsprechend in wenigen Prozessorzyklen und mit geringem Energieverbrauch möglich ist, erfordert das Schreiben eines Bytes sehr viel mehr Zeit und – bedingt durch die lange Wartezeit – auch sehr viel Energie. Das Datenblatt des Prozessors [Atm04b] gibt für das Schreiben eines einzelnen Bytes eine Zeitdauer von 8,5 ms an. Diese Zeitangabe konnte in Experimenten als realistisch bestätigt werden. Vergleicht man diesen Wert mit typischen Zugriffszeiten auf den externen Flash, so erkennt man, dass der Zeitaufwand für das Schreiben eines einzelnen Bytes in das Configuration EEPROM in einer ähnlichen Größenordnung liegt wie das Schreiben einer kompletten Page in den externen Flash. Das Configuration EEPROM eignet sich demnach nicht für die Pufferung von Symboldaten, da die potentiellen Ersparnisse der Pufferung der Symboldaten durch den Zeit- und Energieaufwand für das Beschreiben des EEPROM mehr als aufgehoben würden.

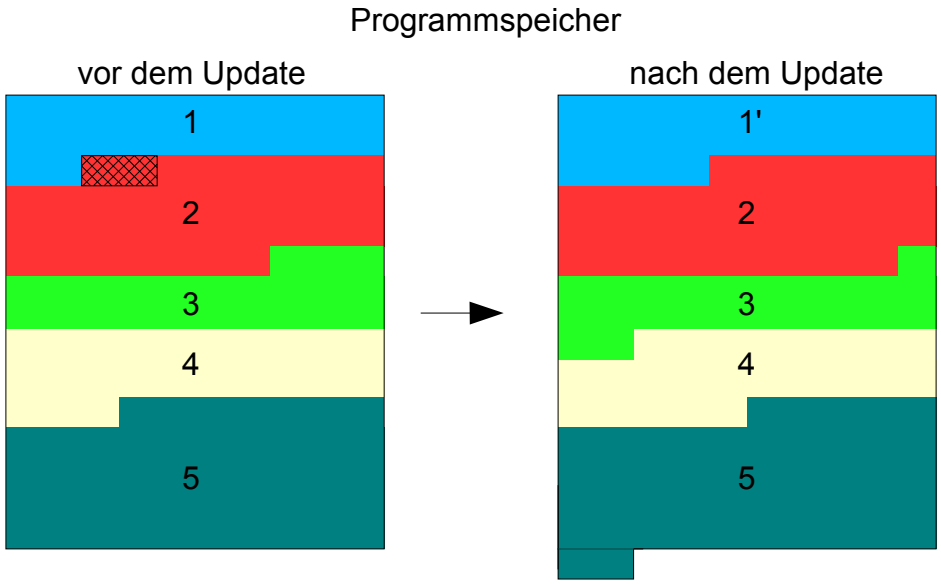


Abbildung 6.4: Codeupdate ohne Pufferspeicher

### 6.6.2 Verwendung von Pufferbereichen

Die Zieladressen der Funktionssymbole eines Programms geben an, an welcher Adresse innerhalb des Programmcodes die einzelnen Funktionsblöcke beginnen. Bei Referenzen auf diese Funktionssymbole handelt es sich dementsprechend um Aufrufe der entsprechenden Funktionen. Kommt es bei einer Änderung innerhalb des Programmcodes zu Verschiebungen, so müssen nicht nur die Zieladressen der Funktionssymbole angepasst sondern auch alle Referenzen auf diese Symbole aktualisiert werden.

Das Beispiel in Abbildung 6.4 demonstriert, dass schon eine vergleichsweise kleine Änderung am Programmcode große Auswirkungen auf das Gesamtlayout des Programms haben kann: Die Abbildung skizziert das Layout einer aus fünf Einzelkomponenten bestehenden Anwendung im Programmspeicher des Sensorknotens vor und nach einem Codeupdate der ersten Komponente. Durch die Änderung vergrößert sich der Umfang des Programmcodes der Komponente, das Ende der Komponente verschiebt sich im Programmspeicher dementsprechend nach hinten (angedeutet durch die schraffierte Fläche). Die Folge ist, dass alle vier dahinter liegenden Komponenten um den entsprechenden Betrag nach hinten verschoben werden müssen. Damit ändern sich die Zieladressen aller Funktionssymbole der Komponenten 2-4, so dass sämtliche Referenzen auf Funktionen dieser Komponenten angepasst werden müssen.

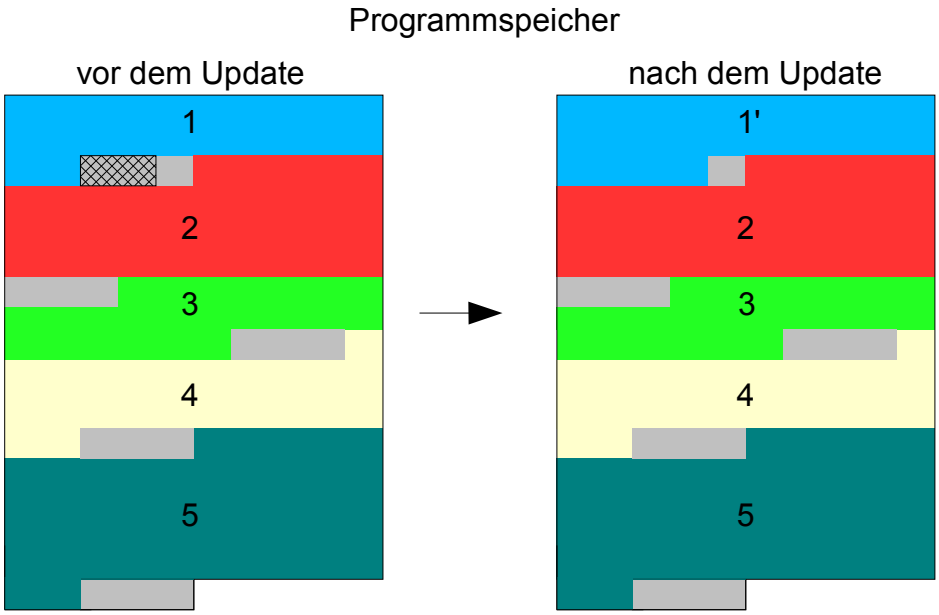


Abbildung 6.5: Codeupdate mit Pufferspeicher

### Pufferbereiche zwischen Komponenten

Eine Möglichkeit, das Verschieben von Codeabschnitten und damit den Aufwand zur Anpassung der Referenzen auf verschobene Funktionssymbole in vielen Fällen zu vermeiden, ist das Einfügen von Pufferbereichen zwischen den Komponenten: Anstatt den Binärcode der einzelnen Komponenten beim Linken direkt aneinanderzureihen, fügt man zwischen die Komponenten jeweils eine gewisse Menge Pufferplatz ein, der für die Funktionalität des Programms zunächst ohne Bedeutung ist. Größenänderungen haben nun keine Auswirkungen mehr auf die nachfolgenden Komponenten, solange sich Zuwächse auf die Größe des verfügbaren Puffers beschränken. Erst wenn dieser erschöpft ist, muss freier Platz durch ein Verschieben der dahinter liegenden Komponenten geschaffen werden.

Abbildung 6.5 demonstriert, wie ein vorhandener Puffer zwischen den Komponenten das Beispiel aus Abbildung 6.4 verändert. Der Puffer zwischen Komponente 1 und Komponente 2 ist ausreichend groß, um den zusätzlichen Platzbedarf der neuen Komponente 1' zur Verfügung stellen zu können. Die Komponenten 2-4 können aus diesem Grund an ihren angestammten Adressen verbleiben; eine Anpassung der Funktionssymbole dieser Komponenten wird nicht notwendig.

Das Beispiel in Abbildung 6.5 veranschaulicht auch den großen Nachteil des Konzepts: Der Programmcode der Anwendung verbraucht mit Pufferbereichen bereits von Anfang an deutlich mehr Speicherplatz. Je tief greifender die erwarteten Änderungen an den Komponenten sind, desto mehr Speicherplatz sollte zwischen den Komponenten freigehalten werden, um später in möglichst vielen Fällen auf ein Verschieben dahinter liegender

Komponenten verzichten zu können. Dies ist kein Problem, solange Programmspeicher in ausreichendem Umfang frei vorhanden ist. Sobald jedoch der Programmspeicher knapp wird, muss eine intelligente Zuteilung des Pufferplatzes gefunden werden, die möglichst viele Änderungen zulässt, ohne dabei übermäßig viel Pufferplatz zu benötigen.

Auf die Größe der zu übertragenden Codeabschnitte hat die Verwendung von Pufferbereichen dagegen keinen Einfluss, da grundsätzlich nur der reine Programmcode übertragen werden muss. Falls nach dem Einfügen des Programmcodes der neuen Komponente Speicherplatz freibleibt, so bildet `FlexCup` daraus automatisch einen Pufferbereich, ohne dass dies in der übertragenen Komponente explizit vorgesehen sein muss.

Verwendet man `FlexCup`, um das Verhalten der Sensorknoten durch den regelmäßigen Austausch einer Komponente an wechselnde Umweltbedingungen oder externe Anforderungen anzupassen, so kann die ideale Größe des Puffers unter Umständen direkt bestimmt werden: Ist die Menge der möglicherweise zum Einsatz kommenden Implementierungen einer Komponente im Voraus bekannt, so sollte der Pufferbereich so gewählt werden, dass die Gesamtgröße aus installierter Komponente und Puffer der Codegröße der größten Komponentenimplementierung entspricht.

### Pufferbereiche im Datenspeicher

Ein weiteres Einsatzgebiet für Pufferbereiche innerhalb von `FlexCup` bieten die *.data*-Abschnitte der Komponenten im Datenspeicher des Programms: Wie bereits in Abschnitt 2.4.1 dargelegt, muss die vom Assembler festgelegte Reihenfolge der Objekte innerhalb eines *.data*-Abschnitts grundsätzlich erhalten bleiben, da die Komponente ihre Datenobjekte nicht anhand des zugehörigen Symbolbezeichners, sondern relativ zur Startadresse des *.data*-Abschnitts referenziert. Durch ein Update neu hinzugekommene Objekte des *.data*-Abschnitts können demnach nicht an einer beliebigen Stelle des Datenspeichers platziert werden, sondern müssen in den *.data*-Abschnitt integriert werden. Dieser ändert dadurch zwangsläufig seine Größe. Dies macht ein Verschieben aller Datenobjekte erforderlich, die im Datenspeicher hinter dem entsprechenden *.data*-Abschnitt platziert sind.

Alle Referenzen auf derart verschobene Datenobjekte müssen im Anschluss aktualisiert werden, um die neue Lage der Objekte zu reflektieren. Dies hat besonders starke Auswirkungen, da die Datenobjekte des *.bss*-Abschnitts hinter den *.data*-Abschnitten platziert sind. Ändert sich die Größe des *.data*-Abschnitts einer Komponente, so müssen sämtliche Symboladressen des *.bss*-Abschnitts aktualisiert werden.

Durch das Einfügen von Pufferbereichen am Ende der einzelnen *.data*-Abschnitte im Datenspeicher können solche Verschiebungen größtenteils verhindert werden. Ändert sich nun die Größe eines *.data*-Abschnitts, so müssen dahinter liegende Symbole nur noch verschoben werden, wenn der vorhandene Pufferplatz nicht ausreicht, um die Größenänderung aufzufangen. In diesem Fall können dann auch die Pufferbereiche dahinter

```
.global SLOPSPACE_GenericCommunication
.data
.type SLOPSPACE_GenericCommunication, @object
.size SLOPSPACE_GenericCommunication, 4
SLOPSPACE_GenericCommunication:
.byte 1
.byte 1
.byte 1
.byte 1
```

Abbildung 6.6: Einfügen von Pufferspeicher in *.data*-Abschnitte

liegender *.data*-Abschnitte genutzt werden, um die Auswirkungen der Größenänderung auf möglichst wenig Symbole zu beschränken.

Innerhalb des *.bss*-Abschnitts werden keine Puffer benötigt. Kommen im Rahmen des Austauschs einer Komponente neue *.bss*-Symbole hinzu, so können die zugehörigen Datenobjekte am Ende des *.bss*-Abschnitts platziert werden, so dass keine Symboladressen verschoben werden müssen.

### Vorbereitung der Pufferbereiche auf dem PC

Für das Einfügen von Pufferbereichen zwischen den einzelnen Komponenten im Programmspeicher sowie zwischen den *.data*-Abschnitten im Datenspeicher gibt es noch keine automatisierte Lösung. Der Benutzer muss die entsprechenden Bereiche manuell hinzufügen. Dies geschieht sehr einfach im Assemblercode der einzelnen Komponenten, bevor mit dem Assembler die Objektdateien der Komponenten erstellt werden.

Zum Erstellen eines Pufferbereichs am Ende des Programmcodes einer Komponente müssen *NOP*-Anweisungen („**N**o **OP**eration“) hinter der letzten Assembler-Anweisung der Komponente eingefügt werden. Jede *NOP*-Anweisungen benötigt dabei 2 Byte Speicherplatz, so dass durch eine entsprechende Anzahl dieser Anweisungen ein beliebig großer Pufferbereich erzeugt werden kann.

Ähnlich erfolgt das Einfügen eines Pufferbereichs am Ende des *.data*-Abschnitts einer Komponente. Wie beispielhaft in Abbildung 6.6 gezeigt, fügt man der Assemblerdatei eine Variable hinzu, die einen Byte-Array beliebiger Größe repräsentiert. Es ist dabei wichtig, den Inhalt des Objekts mit Werten ungleich null zu initialisieren, damit das Objekt tatsächlich Teil des *.data*-Abschnitts und nicht des *.bss*-Abschnitts wird.

Durch die Verwendung des Schlüsselworts *SLOPSPACE* am Anfang des Variablennamens wird dem `FlexCup`-Analyzer signalisiert, dass es sich bei diesem Objekt um kein Datenobjekt handelt, sondern um Pufferplatz. Der `FlexCup`-Analyzer verzichtet in diesem Fall darauf, für das Datenobjekt ein Symbol in der Symboltabelle der Sensorknoten

zu erzeugen und speichert stattdessen die Größe des Pufferbereichs in den Komponenteninformationen.

#### Vergleich mit ähnlichen Ansätzen

Das Einfügen von Puffern ist kein neues Konzept dieser Arbeit. Reijers und Langendoen [RL03] schlagen die Verwendung von Padding vor, um bei Änderungen das Verschieben von Funktionen innerhalb des Codes zu verhindern. Auf diese Weise soll die Größe des nötigen Diff-Skriptes möglichst klein gehalten werden. Ähnlich argumentieren auch Koshy und Pandey [KP05], die für den Puffer zwischen Funktionen den Begriff Slop Space verwenden. Durch den Einsatz von Slop Space soll verhindert werden, dass geänderte Funktionen an eine andere Stelle im Programmspeicher verschoben werden müssen.

Puffer können im Programmspeicher nicht nur auf der Ebene der Komponenten eingefügt werden, prinzipiell lassen sich auch einzelne Funktionen innerhalb einer Komponente mit Hilfe von Slop Space voneinander separieren. Ändert sich dann die Größe einer einzelnen Funktion, so muss dies keine Auswirkungen auf die Platzierung der restlichen Funktionen einer Komponente haben. Diesen Ansatz wählen Koshy und Pandey [KP05], damit geänderte Funktionen nach Möglichkeit an ihrem angestammten Platz verbleiben können. Da hierbei potentiell an sehr vielen Stellen Pufferplatz eingefügt wird, ist es sehr wichtig die Größe der einzelnen Pufferbereiche intelligent zu wählen, um den zusätzlichen Platzbedarf im Programmspeicher in Grenzen zu halten. Koshy und Pandey schlagen vor, die Größe des Pufferspeichers einer Funktion abhängig von der erwarteten Veränderlichkeit der Funktion zu wählen, einer Funktion also nur dann viel Puffer zuzuweisen, wenn mit zahlreichen und tief greifenden Veränderungen gerechnet werden muss.

Da `FlexCup` den Codeaustausch auf Komponentenebene betreibt, sind momentan auch nur Pufferbereiche zwischen den Binärkomponenten eines Programms vorgesehen. Um Pufferspeicher auch innerhalb einer Komponente sinnvoll einsetzen zu können, muss das Layout einer Komponente nach Änderungen so weit als möglich erhalten bleiben. Um dies bei `FlexCup` jedoch sicherstellen zu können, müsste man bereits auf der Ebene des Compilers eingreifen und dadurch eine Art inkrementelles Kompilieren ermöglichen.

### 6.6.3 Umgehen der Flashpuffer

Das normale Vorgehen bei Lese- oder Schreibzugriffen auf Daten im externen Flash ist, zunächst zu überprüfen, ob die benötigte Page bereits in einem der beiden Flashpuffer für den Zugriff bereitliegt. Ist dies nicht der Fall, so muss zunächst die Page in den Puffer geladen werden, bevor anschließend lesend beziehungsweise schreibend darauf zugegriffen werden kann.

In bestimmten Fällen kann es sinnvoll sein, die beiden Flashpuffer zu umgehen und den Inhalt einer kompletten Page aus dem externen Flash direkt in den RAM des Prozessors zu laden. Der Atmel AT45DB041B bietet hierfür einen gesonderten Befehl an, der nach

dem Durchlaufen einer entsprechenden Initialisierungssequenz kontinuierlich die Bytes der Page ausgibt.

Sinnvoll ist das direkte Laden einer Page aus dem externen Flash nur, wenn auf die Daten lediglich lesend zugegriffen werden soll. Schreibzugriffe sind hardware-bedingt grundsätzlich nur über einen der zwei Flashpuffer möglich. Diese müssen jedoch zu diesem Zweck zuvor mit dem Inhalt der zu beschreibenden Page initialisiert worden sein, da beim Schreiben grundsätzlich die gesamte Page vom Flashpuffer in den Flash übertragen wird, und nicht nur die geänderten Bytes.

Das direkte Laden einer Page aus dem externen Flash in den Speicher des Prozessors kann in zwei Fällen vorteilhaft sein:

- **Paralleler Zugriff auf mehr als zwei Adressbereiche im Flash:** Muss innerhalb eines Programmteils immer wieder im Wechsel auf mehr als zwei unterschiedliche Bereiche des externen Flash zugegriffen werden, so kann dies dazu führen, dass der Flashpuffer bei jedem dieser Zugriffe neu geladen werden muss. Als Beispiel betrachte man den Fall, dass ein Programm zunächst ein Byte aus der Page X, danach ein Byte aus der Page Y und dann ein Byte aus der Page Z einliest. Im Anschluss daran liest es ein weiteres Byte der Page X, dann wieder ein Byte der Page Y und ein Byte der Page Z und so weiter. Immer wenn das Programm auf ein Byte der Page Z zugreifen möchte, befinden sich gerade die Page X und die Page Y in den zwei Puffern des externen Flash. Page Z muss daraufhin nachgeladen werden und ersetzt Page X, was dann beim nächsten Speicherzugriff wiederum dazu führt, dass Page X nachgeladen werden muss. Dieses zirkuläre Ersetzen macht jeden einzelnen Zugriff auf den externen Flash extrem teuer. Ist jedoch bekannt, dass auf eine der drei Pages, zum Beispiel Page Z, nur lesende Zugriffe erfolgen, so können die Daten dieser Page direkt in den Speicher des Prozessors geladen werden. In diesem Fall können Page X und Page Y ständig im Flashpuffer bleiben, während die Zugriffe auf Page Z direkt auf dem Puffer im RAM des Prozessors erfolgen.

- **Lesezugriff auf einen Großteil der Daten einer Page:** Werden nicht nur einzelne Bytes, sondern ein Großteil der Daten einer Page benötigt, so kann es sich lohnen die Page zunächst komplett in den Hauptspeicher zu laden und die einzelnen Datenzugriffe anschließend auf dem Puffer im RAM auszuführen. Dies hängt damit zusammen, dass auch für das Lesen von Daten aus dem Flashpuffer zunächst jeweils ein Flashbefehl mit entsprechender Initialisierungssequenz an den Flashbaustein übertragen werden muss. Werden die Daten der Page in kleinen Portionen (zum Beispiel in einzelnen Bytes oder Words) aus dem Puffer gelesen, so fällt hierbei eine Vielzahl solcher Initialisierungsschritte an. Für das Laden einer kompletten Page in den Hauptspeicher muss dagegen nur ein Befehl und eine Initialisierungssequenz an den Flashbaustein übertragen werden, anschließend werden die 264 Bytes der Page am Stück in den Hauptspeicher geladen.

Um der Anwendung einen einfachen, möglichst transparenten Direktzugriff auf den Speicher des externen Flash zu ermöglichen, wurden in der Flashkomponente die folgenden

zwei Funktionen implementiert:

- **external_flash_directMemory_read_byte**
- **external_flash_directMemory_read_word**

Bei jedem Zugriff auf eine dieser beiden Funktionen wird zunächst geprüft, ob die Page der angeforderten Speicheradresse bereits im RAM geladen ist. Ist dies der Fall, so kann der Lesezugriff direkt auf dem Datenarray im Hauptspeicher erfolgen, ansonsten muss zunächst die Page in den Hauptspeicher geladen werden.

Innerhalb des `FlexCup`-Linkers konnte die in diesem Abschnitt beschriebene Optimierung für das Durchlaufen der Relocationtabellen eingesetzt werden. In diesem Arbeitsschritt muss das Programm parallel auf drei unterschiedliche Stellen im externen Flash zugreifen: Zum einen müssen die Relocationeinträge aus den Relocationtabellen des Programms eingelesen werden. Parallel dazu muss auf die Daten der Symboltabelle zugegriffen werden (sofern diese nicht schon im Symboltabellenpuffer im RAM liegen). Ist eine Anpassung der jeweiligen Relocation notwendig, so muss als dritter Flashspeicherzugriff sowohl lesend als auch schreibend auf die Programmdaten zugegriffen werden.

Als Optimierung wird nun für die Zugriffe auf die Relocationeinträge der Direktzugriff auf den Flashspeicher unter Umgehung der Flashpuffer genutzt. Dadurch werden die Puffer nur noch für das Laden der Symboltabelleneinträge und für das Schreiben geänderter Relocations benötigt.

### 6.6.4 Spätes Rücksetzen der Änderungsmarkierungen

Während des Mergens der bestehenden Symboltabelle mit der Symboltabelle der geänderten Komponente werden Symbole, die geändert oder neu hinzugefügt wurden, als geändert markiert, indem das höchstwertige Bit des Symbolbezeichners auf eins gesetzt wird. Diese Markierung wird anschließend beim Durchlaufen der Relocationtabellen dazu verwendet, um festzustellen, ob eine Relocation angepasst werden muss. Als letzter Schritt des `FlexCup`-Linkers müssen diese Markierungen wieder zurückgesetzt werden, damit die Metadaten für spätere Durchläufe des Programms erneut genutzt werden können.

Das Zurücksetzen der Markierungen am Ende des Programms erfordert das Durchlaufen der kompletten Symboltabelle mit Lesezugriffen auf alle Symbolbezeichner und Schreibzugriffen auf alle zuvor als geändert markierten Symbolbezeichner. Je nach Größe der Symboltabelle ist hierfür wiederum ein mehrfaches Laden und Rückschreiben der Flashpuffer notwendig.

Für die Funktionssymboleinträge der Symboltabelle lässt sich dieses Vorgehen optimieren, indem man die Markierungen nicht am Ende des Programms in einem extra Schritt zurücksetzt, sondern damit bis zur nächsten Ausführung des `FlexCup`-Linkers wartet. Die Markierung der Symbole wird dann während des Mergens der Symboltabelle mit

der Symboltabelle der nächsten geänderten Komponente entfernt – und eventuell gleich wieder gesetzt, falls das betreffende Symbol erneut geändert wurde. Der Vorteil dieses späten Rücksetzens liegt darin, dass die Einträge der Symboltabelle während des Merge-Vorgangs in jedem Fall eingelesen werden müssen, so dass ein Teil des Aufwands für das Rücksetzen der Änderungen entfällt.

Für Objektsymboleinträge ist eine derartige Optimierung nicht so einfach möglich, da auf die Objektsymbole bereits bei der Berechnung des neuen Layouts des *.data*-Abschnitts im ersten Schritt des `FlexCup`-Linkers zugegriffen werden muss. Dabei könnte nicht unterschieden werden, ob ein Objektsymbol bereits aufgrund einer Änderung des Speicherlayouts als geändert markiert wurde, oder ob die Markierung noch aus einem früheren Durchlauf des `FlexCup`-Linkers stammt.

### 6.6.5 Mögliche weitere Optimierungen

Eine Reihe weiterer Optimierungen wurden für die Implementierung von `FlexCup` angedacht, aber bisher noch nicht umgesetzt. In diesem Abschnitt sollen vier dieser möglichen Optimierungen kurz vorgestellt werden.

#### Binäre Suche in der Symboltabelle

Zahlreiche Operationen des `FlexCup`-Linkers erfordern die Suche eines Symbols in der Symboltabelle. Momentan wird die Symboltabelle hierfür sequentiell durchlaufen, was zu keinen Performance-Nachteilen führt, wenn auf einen Großteil der Symboleinträge zugegriffen werden muss. Erfolgt dagegen ein Zugriff nur auf wenige Einträge der Symboltabelle, so entsteht ein merklicher Aufwand für das Suchen der Einträge in der Tabelle.

Da die Symboltabelle nach Symbolbezeichnern sortiert gespeichert ist, könnte alternativ als Suchalgorithmus die binäre Suche verwendet werden. Bei einer Gleichverteilung der Symboleinträge ist dann für das Auffinden der Symboleinträge im Allgemeinen nur noch ein logarithmischer Aufwand erforderlich. In der konkreten Implementierung auf den Mica2-Motes hätte die binäre Suche jedoch auch einen entscheidenden Nachteil: Da im schnellen Wechsel auf Einträge an verschiedensten Positionen der Symboltabelle zugegriffen wird (die Suche wird durch Halbierung des Suchraums konsequent eingegrenzt), müssen die Datenpuffer des externen Flash voraussichtlich häufiger mit neuen Daten geladen werden. Beim sequentiellen Zugriff auf die Symboltabelle muss dagegen jede benötigte Page nur genau einmal geladen werden.

#### Einsprungadressen für die Symboltabelle

Eine weitere Möglichkeit, die Suche nach Einträgen der Symboltabelle zu beschleunigen, könnte in der Verwendung einer Tabelle liegen, in der Einsprungadressen zu verschiedenen Bereichen der Symboltabelle gespeichert sind. Bei der Suche nach einem Symbol würde

der `FlexCup`-Linker dann zunächst, abhängig vom gesuchten Symbolbezeichner, einen Eintrag dieser Tabelle lesen und die Suche nach dem Symbol dann an der entsprechenden Adresse innerhalb der Symboltabelle beginnen.

Die Tabelle mit den Einsprungadressen der Symboltabelle sollte eine feste Größe besitzen und im RAM der Sensorknoten gespeichert werden. Jeder Eintrag könnte für einen bestimmten Bereich des Namensraums der möglichen Symbolbezeichner zuständig sein, und zwar idealerweise so, dass der `FlexCup`-Linker aus dem gesuchten Symbolbezeichner mit geringem Aufwand auf den passenden Eintrag in der Tabelle schließen kann.

Bei einer möglichen Umsetzung dieser Optimierung muss der zusätzliche Aufwand in Betracht gezogen werden, der für das Anlegen und die Pflege der Tabelle anfällt. Sobald es innerhalb der Symboltabelle zu Verschiebungen kommt, müssen die Tabelleneinträge mit neuen Werten angepasst werden.

### Puffereinträge in der Symboltabelle

Das Einfügen von Symbolen in die Symboltabelle der Sensorknoten ist eine teure Operation: Um die Sortierung nach Symbolbezeichnern zu erhalten, muss der neue Eintrag an einer festen Position in die Symboltabelle eingefügt werden. Zu diesem Zweck ist es wiederum erforderlich, alle dahinter liegenden Symbole zu verschieben. Dieser Effekt wird momentan durch eine Pufferung der geänderten Symboleinträge im Hauptspeicher der Sensorknoten abgefangen (siehe Abschnitt 6.6.1).

Eine alternative Möglichkeit zur Optimierung derartiger Änderungen der Symboltabelle ist das Einfügen von Puffereinträgen. In die Ursprungsversion der Metadaten werden hierfür an mehreren Stellen der Symboltabelle leere Symboleinträge mit einem Dummy-Bezeichner (zum Beispiel `0xffff`) eingefügt. Beim Lesen der Symboltabelle werden diese Einträge ignoriert, beim Einfügen eines neuen Symbols müssen die dahinter liegenden Symbole nur soweit verschoben werden, bis ein solcher leerer Puffereintrag gefunden und aufgefüllt werden kann.

Die Zahl der Puffereinträge sollte insgesamt jedoch klein bleiben, da jeder Puffereintrag einerseits den Speicherplatzbedarf der Symboltabelle im externen Flash vergrößert und andererseits zusätzlicher Aufwand für das Lesen der Puffereinträge bei jedem Einlesen der Symboltabelle notwendig ist.

### Selektives Übertragen in den Programmspeicher

Nach der Bearbeitung der Daten durch den `FlexCup`-Linker überträgt der `FlexCup`-Bootloader momentan grundsätzlich den gesamten Programmcode aus dem externen Flash in den Programmspeicher des Prozessors. Hierfür liest er jeweils 256 Bytes große Abschnitte aus dem externen Flash und schreibt diese dann als Page in den Programmspeicher.

Falls ein Codeupdate nur geringfügige Auswirkungen auf den Inhalt des Programmspeichers hat, zum Beispiel im Falle einer einzelnen geänderten Konstante, so ändert sich der Inhalt vieler Pages im Programmspeicher nicht. Eine mögliche Optimierung wäre demnach, wenn der `FlexCup`-Bootloader nur die tatsächlich geänderten Pages neu schreiben und alle anderen Pages unverändert lassen würde.

Um das selektive Übertragen von Pages in den Programmspeicher zu ermöglichen, müsste der `FlexCup`-Linker eine Page als geändert markieren, sobald er zum ersten Mal schreibend auf den darin enthaltenen Programmcode zugreift. Dies erfordert natürlich eine entsprechende Datenstruktur, die diese Änderungsinformationen speichert. Außerdem muss der `FlexCup`-Linker die Möglichkeit haben, die Adressen des Programmcodes im externen Flash der korrekten Page zuzuordnen.

Eine mögliche Implementierung bestünde darin, die Pages des Programmspeichers (256 Byte Größe) an den Pages des externen Flash (264 Byte Größe) auszurichten. Die übrig bleibenden 8 Bytes in jeder Page des externen Flash könnten dann unter anderem für die Speicherung der Änderungsinformation genutzt werden. Eine Zuordnung der Änderungsinformation wäre in diesem Fall trivial, da der `FlexCup`-Linker bei jedem Schreibzugriff lediglich das Änderungsbit der jeweiligen Page setzen müsste.

Der Nachteil dieser Lösung liegt im zusätzlichen Speicherplatzbedarf, sowie im nicht unerheblichen Aufwand für die Pflege der Änderungsinformationen. Es muss also sorgfältig geprüft werden, ob diese Optimierung tatsächlich zu Einsparungen führt, oder ob der anfallende Zusatzaufwand den Nutzen komplett zunichte macht.

## 6.7 Stand der Gesamtimplementierung

Mit dem in diesem Kapitel vorgestellten `FlexCup`-Analyzer, dem `FlexCup`-Linker sowie dem `FlexCup`-Bootloader stehen nun alle benötigten Einzelanwendungen zur Verfügung, die für den dynamischen Austausch von Komponenten innerhalb von `TinyCubus` benötigt werden.

Der nächste, bisher noch ausstehende Schritt ist die Integration der einzelnen Programme zu einer Gesamtanwendung. Insbesondere müssen die Implementierung des Codeverteilungsalgorithmus (siehe Abschnitt 4.2) und der `FlexCup`-Linker aufeinander abgestimmt werden. Zum einen verwendet der Codeverteilungsalgorithmus auf den Sensorknoten mit RAFS ein eigenes, speziell entwickeltes Dateisystem um die empfangenen Daten abzuspeichern. Da der `FlexCup`-Linker seine Daten momentan in einem zusammenhängenden Block im externen Flash erwartet, muss er bezüglich der Verwendung des Dateisystems noch angepasst werden. Ein weiterer Punkt ist der Beginn der Updateprozedur. Die nesC-Komponenten der Codeverteilungsimplementierung müssen zukünftig den `FlexCup`-Linker aufrufen, sobald neue Komponentendaten vollständig empfangen wurden.

Auch auf PC-Seite müssen die einzelnen Implementierungen noch miteinander verbunden werden. So sollte der `FlexCup`-Analyzer, der den Programmcode und Metadaten der

Komponenten zusammenstellt, mit dem Programm verbunden werden, das die Daten in das Sensornetz injiziert. Auf diese Weise könnten Codeupdates erstellt und anschließend direkt im Sensornetz verteilt werden.

# 7 Evaluation

Nachdem das Konzept und die Implementierung von `FlexCup` in den letzten zwei Kapiteln ausführlich vorgestellt wurden, soll in diesem Kapitel nun geklärt werden, welche Vorteile die Verwendung von `FlexCup` gegenüber dem Einsatz vergleichbarer Konzepte bietet.

Zunächst werden im nächsten Abschnitt einige theoretische Überlegungen zu dieser Fragestellung präsentiert. Um die Leistungsfähigkeit von `FlexCup` realistisch einschätzen zu können, wird der `FlexCup`-Ansatz anschließend in Abschnitt 7.2 anhand verschiedener Experimente mit zwei anderen Ansätzen verglichen.

## 7.1 Theoretische Betrachtungen

Bevor die Eigenschaften von `FlexCup` in Abschnitt 7.2 anhand praktischer Untersuchungen bewertet werden, sollen in diesem Abschnitt einige prinzipielle Merkmale von `FlexCup` zur Sprache kommen. Insbesondere wird auf einige prinzipbedingte Vor- und Nachteile des Ansatzes gegenüber anderen Lösungen eingegangen.

### 7.1.1 Bedeutung für TinyCubus

Unbeeinflusst von der generellen Leistungsfähigkeit des `FlexCup`-Ansatzes beim Austausch von Programmcode in Sensornetzen bleibt die Bedeutung von `FlexCup` für das `TinyCubus`-System. Für die Realisierung der Ziele von `TinyCubus` lässt sich `FlexCup` nicht ohne weiteres durch Alternativlösungen wie beispielsweise Deluge oder einen der Differenzansätze ersetzen.

Die Knoten eines Sensornetzes sollen mit `TinyCubus` zukünftig in der Lage sein, autonom über ihre Rolle im Sensornetz und über den Austausch einzelner Komponenten zu entscheiden. Dies ist nur möglich, wenn die Sensorknoten den Programmcode einer Komponente installieren können, ohne dass dies explizit durch einen zentralen Rechner gesteuert wird.

Bei Deluge muss der zentrale Rechner grundsätzlich den kompletten Programmcode an die Sensorknoten schicken, der Austausch einzelner Komponenten ist gar nicht möglich. Um auf verschiedenen Sensorknoten verschiedene Kombinationen der Komponenten betreiben zu können, müsste Deluge demnach alle möglichen Kombinationen jeweils als Gesamtprogramm im Netz verbreiten.

Die Differenzansätze beherrschen zwar den Austausch auch kleiner Teile des Programmcodes, jedoch benötigt der zentrale Rechner zu diesem Zweck detailliertes Wissen über die Konfiguration der einzelnen Knoten, um die Änderungsprotokolle erstellen zu können. Eine autonome Entscheidung der Sensorknoten ist damit nicht möglich.

`FlexCup` wurde dagegen mit Blick auf die Erfordernisse von `TinyCubus` entwickelt und bietet die erforderlichen Eigenschaften für den Austausch von Komponenten auf den einzelnen Sensorknoten. Wenn das Tiny Data Management Framework die Entscheidung zur Installation einer Komponente fällt, so muss es den Code und die Metadaten der Komponente lediglich an den `FlexCup`-Linker übergeben, der dann für die Installation der Komponente sorgt. Für das Erstellen der notwendigen Metadaten der Komponente benötigt der zentrale Rechner keine Informationen zur Konfiguration der einzelnen Sensorknoten.

### 7.1.2 Benötigte Ressourcen

Den potentiellen Vorteilen von `FlexCup`, allem voran den Einsparungen bei der Übertragung der Codeupdates, stehen Kosten durch den Verbrauch von Ressourcen gegenüber, die für den Einsatz von `FlexCup` benötigt werden. `FlexCup` benötigt Speicherplatz im Programmspeicher sowie im externen Flash der Sensorknoten und verursacht Berechnungsaufwand durch den Einsatz des `FlexCup`-Linkers.

**Speicherplatz im Programmspeicher:** `FlexCup` muss drei unterschiedliche Programmteile im Programmspeicher der Sensorknoten platzieren. Zum einen muss die Sensornetzanwendung einen `FlexCup`-spezifischen Teil enthalten, der für den Empfang der einzelnen Nachrichten eines Codeupdates zuständig ist und empfangene Daten im externen Flash des Sensorknotens zwischenspeichert. Daneben müssen im Programmspeicher der Code des `FlexCup`-Linkers und des `FlexCup`-Bootloaders abgelegt werden.

**Speicherplatz im externen Flash:** Im externen Flash der Sensorknoten müssen die Metadaten der Anwendung sowie eine Kopie des Programmcodes gespeichert werden. Zusätzlich muss Speicherplatz für die Daten der Komponentenupdates vorhanden sein, bei Bedarf auch für mehrere Einzelkomponenten gleichzeitig. Bei großen Programmen schlägt vor allem die benötigte Kopie des Programmcodes zu Buche. Zur Veranschaulichung eignet sich folgender Extremfall: Mit 128 Kilobyte Daten, die maximal in den Programmspeicher der Mica2-Sensorknoten passen, ließe sich bereits ein Viertel der Kapazität des externen Flash ausfüllen.

Prinzipiell kann auf das permanente Speichern der Kopie des Programmcodes auch verzichtet werden. In diesem Fall müssen die Daten dann jedoch bei Bedarf aus dem Programmspeicher nachgeladen werden, was zusätzlichen Arbeitsaufwand des Prozessors verursacht. Eine bestimmte Menge Speicherplatz müsste trotz allem freigehalten werden, um ein Zwischenspeichern der Programmdaten zu ermöglichen.

**Berechnungsaufwand auf den Sensorknoten:** Neben dem zusätzlichen Speicherplatz benötigt `FlexCup` vor allem Rechenzeit des Prozessors für das Ausführen des `FlexCup`-Linkers. Dieser Berechnungsaufwand gilt aus zwei Gründen als Ressourcenverbrauch: Zum einen steht der Prozessor nicht für andere Programme zur Verfügung, während der `FlexCup`-Linker ausgeführt wird; er kann in dieser Zeit seine normalen Aufgaben also nicht erfüllen. Zum anderen verbraucht der `FlexCup`-Linker in diesem Zeitraum Energie für den Betrieb des Prozessors und für den Zugriff auf Ressourcen wie den externen Flash.

Der größere Berechnungsaufwand für die Vorbereitung und die Installation des neuen Programmcodes auf den Sensorknoten lässt sich rechtfertigen, falls diesem Aufwand entsprechende Einsparungen bei der Übertragung des Programmcodes über die Funkschnittstelle gegenüberstehen. Die zu übertragende Datenmenge beim Austausch einer Komponente muss also klein genug sein, damit sich der Aufwand des `FlexCup`-Linkers zur Integration der Komponente in den Programmcode gegenüber der Übertragung des gesamten Programmcodes lohnt.

Dem durch die Verwendung von `FlexCup` benötigten Speicherplatz im externen Flash sowie im Programmspeicher stehen dagegen keine Platzeinsparungen an anderer Stelle gegenüber. Es ist auch nicht möglich, die Einsparungen bei der Übertragung einer neuen Komponente quantitativ mit dem dafür erforderlichen Speicherplatz auf den Sensorknoten zu vergleichen. Falls ausreichend freier Speicherplatz im Programmspeicher sowie im externen Flash der Sensorknoten vorhanden ist, so lässt sich `FlexCup` für den dynamischen Austausch von Komponenten einsetzen, ansonsten müssen andere Lösungen gewählt werden.

Auch andere Ansätze zum Codeupdate in Sensornetzen benötigen Ressourcen zur Erfüllung ihrer Aufgaben. Am wenigsten Aufwand für Berechnungen verursachen Ansätze wie Deluge oder Xnp (siehe Abschnitt 3.1), da sie nur den Programmcode empfangen, ihn im externen Flash zwischenspeichern und anschließend im Programmspeicher installieren. Im Gegenzug verbrauchen diese Ansätze viel Zeit und Energie beim Empfang des Programmcodes. Die Differenzansätze (siehe Abschnitt 3.2) benötigen dagegen wie `FlexCup` deutlich geringere Übertragungsvolumina, erkaufen diesen Vorteil aber mit zusätzlichem Aufwand für die Verarbeitung der Änderungsprotokolle.

### 7.1.3 Systemabhängigkeit der Lösung

Ein Nachteil des `FlexCup`-Ansatzes zum Austausch von Komponenten besteht in der inhärent engen Bindung der Implementierung an das System. Ein bedeutender Teil der Implementierung hängt von den Eigenschaften des Compilers, des Linkers und der verwendeten Hardware ab und lässt sich nur mit erheblichem Aufwand auf andere Plattformen übertragen.

Sowohl der `FlexCup`-Analyzer als auch der `FlexCup`-Linker nutzen an zahlreichen Stellen ihr Wissen zum genauen Verhalten des AVR-GCC-Compilers und des AVR-GCC-Linkers; schon die Verwendung eines anderen Compilers auf gleicher Hardware kann tief greifende

Veränderungen an der `FlexCup`-Implementierung notwendig machen. Ein Beispiel sind die in Tabelle 2.2 aufgeführten Relocationtypen, die der Linker beherrschen muss. Bei einem anderen Compiler muss zumindest mit einer anderen Aufteilung, wenn nicht sogar mit komplett unterschiedlichen Relocationtypen gerechnet werden. Ein weiteres Beispiel ist das Layout des Programms innerhalb des Programm- und innerhalb des Datenspeichers. Nur wenige Teile des in Abschnitt 2.4.1 beschriebenen Speicherlayouts sind durch die Hardware vorgegeben, der Rest wird durch den Compiler definiert und kann bei einem anderen Compiler komplett unterschiedlich aussehen.

Bei einer Portierung des Systems auf andere Hardwareplattformen muss `FlexCup` sowohl an Änderungen des Compilers als auch an geänderte Anforderungen der Hardware angepasst werden. Ein Extrembeispiel ist die Portierung auf ein System, das nicht auf der Harvard-Architektur aufbaut und dementsprechend die Trennung des Adressbereichs für Daten- und Programmspeicher nicht kennt. In diesem Fall müsste die ganze Architektur von `FlexCup` an die geänderten Voraussetzungen angepasst werden.

Neben der reinen Portierung der Anwendung für eine andere Sensorplattform müssen auch die Annahmen überprüft werden, die den realisierten Optimierungen zu Grunde liegen. In der vorliegenden Implementierung für Mica2-Sensorknoten wurde beispielsweise versucht, die Anzahl der Zugriffe auf den externen Flash zu minimieren und die notwendigen Zugriffe mit Hilfe der zwei Datenpuffer des Speicherbausteins möglichst effizient zu gestalten. Für Sensorplattformen, die einen anderen externen Speicherbaustein als den Atmel AT45DB041B der Mica2-Sensorknoten einsetzen, müssen unter Umständen andere Optimierungsansätze gewählt werden – zum Beispiel wenn in dem Speicherbaustein keine oder eine andere Zahl an Datenpuffern vorhanden sind. In einigen Fällen sind solche Optimierungen nicht auf einzelne Module beschränkt, sondern erstrecken sich über die gesamte Implementierung. So wurde im konkreten Beispiel für die Speicherzugriffe des gesamten Programms versucht, die Datenpuffer möglichst optimal auszunutzen.

Sollte ein System derart heterogen aufgebaut sein, dass sich die Hardware der einzelnen Knoten substanziell voneinander unterscheidet, so muss für jeden Knotentyp eine gesonderte Lösung implementiert werden. Dieses spezielle Problem beschränkt sich jedoch nicht auf den hier realisierten Ansatz. Lösungen wie Deluge bieten momentan noch keinerlei Unterstützung durch das System für Heterogenität im Netzwerk und müssten dementsprechend ebenfalls erweitert werden.

Die in diesem Abschnitt geschilderte enge Bindung der Implementierung an das jeweilige System ist sicherlich ein Hindernis für die schnelle Verbreitung der Lösung auf eine Vielzahl unterschiedlicher Sensorplattformen. Auf der anderen Seite enthält auch TinyOS selbst eine Vielzahl hardware-abhängiger Teile, die bei Portierungen auf eine andere Plattform angepasst werden müssen. Da die Anpassungen jeweils nur einmal pro Plattform und nicht für jede einzelne Anwendung erfolgen müssen, kann man argumentieren, dass der Aufwand bezogen auf den möglichen Nutzen gerechtfertigt ist.

### 7.1.4 Mangelnde Toleranz gegenüber Abstürzen

In der derzeitigen Implementierung ist `FlexCup` nicht in der Lage, den Absturz eines Sensorknotens während des Updateprozesses zu tolerieren. Dies hängt primär damit zusammen, dass während des gesamten Ablaufs des `FlexCup`-Linkers immer wieder die Metadaten des Programms geändert werden. Falls der Sensorknoten nun abstürzt, bevor der Updateprozess abgeschlossen ist, so wurde unter Umständen nur ein Teil der Metadaten geändert. In diesem Fall befinden sich die Metadaten ab diesem Zeitpunkt in einem inkonsistenten Zustand, der bei weiteren Updateversuchen zu einem nicht vorhersehbaren Verhalten führt.

Vollständig lässt sich diese Problematik in keinem System vermeiden, da ein Absturz während der Übertragung von Daten in den Programmspeicher grundsätzlich zu einem inkonsistenten Zustand führt. `FlexCup` ist jedoch über einen längeren Zeitraum anfällig, da bereits die Bearbeitung der Daten durch den `FlexCup`-Linker nicht durch Abstürze unterbrochen werden darf.

## 7.2 Praktische Untersuchungen

Da die Leistung von `FlexCup` alleine nur sehr schwierig zu bewerten ist, wurde die `FlexCup`-Implementierung in mehreren Experimenten mit Deluge [HC04] (siehe Abschnitt 3.1.2) und dem Diff-Ansatz von MOAP [SHE03][YYS03] (siehe Abschnitt 3.2.2) verglichen. Mit Deluge wurde ein Verfahren ausgewählt, das den Programmcode grundsätzlich komplett austauscht, während es sich bei MOAP um einen Vertreter der Differenzansätze handelt. Wichtig war bei der Auswahl, dass für beide Verfahren eine Implementierung für Mica2-Sensorknoten verfügbar ist, so dass direkte Vergleiche zu `FlexCup` angestellt werden können.

### 7.2.1 Untersuchte Anwendungen

Das Verhalten und die Performance von FlexCup im realen Einsatz wurde mit Hilfe dreier Anwendungen exemplarisch untersucht: **OscilloscopeRF**, **Surge** und **AcousticLocalization**. Alle drei Anwendungen sind Teil des TinyOS-Programmpakets und können über das TinyOS-CVS [1] bezogen werden. Sie repräsentieren ein Spektrum typischer TinyOS-Anwendungen unterschiedlicher Komplexität, von der sehr einfachen OscilloscopeRF-Anwendung bis hin zum relativ komplexen AcousticLocalization.

**OscilloscopeRF** ist eine vergleichsweise einfache Beispielanwendung, bei der ein Sensorknoten die Daten eines seiner Sensoren aufzeichnet und direkt per Funk an eine Basisstation überträgt. Teil des OscilloscopeRF-Programmpakets ist eine Java-Anwendung für den PC, die die Messwerte des Sensors entgegennimmt und graphisch – wie in einem

[1] http://cvs.sourceforge.net/viewcvs.py/tinyos/

Oszilloskop – darstellt. OscilloscopeRF unterstützt bei der Funkkommunikation keine Übertragung über Zwischenknoten, so dass der aufzeichnende Sensorknoten direkt mit der Basisstation kommunizieren können muss. In den folgenden Untersuchungen ist die OscilloscopeRF-Anwendung auf den Betrieb mit einem Helligkeitssensor ausgelegt.

**Surge** ist eine etwas aufwendigere Anwendung zur Anzeige der Netzwerktopologie in Multihop-Sensornetzen. Die PC-Anwendung empfängt per Funk Informationen von den einzelnen Sensorknoten, wobei bei Surge im Gegensatz zu OscilloscopeRF auch eine Übertragung der Nachrichten über Zwischenknoten möglich ist. Die Anwendung stellt auf Basis dieser Informationen die Topologie des Sensornetzes grafisch dar und gibt Informationen zu der Anzahl der gesendeten Nachrichten der einzelnen Knoten aus.

Bei **AcousticLocalization** [SBMkL04] handelt es sich um die anspruchvollste und komplizierteste der drei Beispielanwendungen. Ihre Aufgabe ist die Positionsbestimmung von Knoten in einem Sensornetz. Zu diesem Zweck bestimmen die Sensorknoten zunächst ihre jeweilige Distanz zueinander, indem sie die Laufzeit eines akustischen Signals messen. Anschließend werden die ermittelten Abstände der Sensorknoten an die Basisstation übermittelt, die daraus mit Hilfe der Triangulation die Position der einzelnen Sensorknoten bestimmen kann. Die Distanzmessungen erfolgen, indem einer der Sensorknoten zunächst ein Funksignal versendet, um den Beginn der Messung zu signalisieren. Direkt im Anschluss daran erzeugt er ein akustisches Signal, das von den anderen Sensorknoten über deren Mikrofon empfangen werden kann. Da sich das Funksignal fast $10^6$ mal so schnell ausbreitet wie das akustische Signal, können die Empfängersensorknoten aus der Differenz der Ankunftszeiten der beiden Signale auf die Laufzeit des akustischen Signals und daraus wiederum auf die Distanz zum Sender schließen.

Tabelle 7.1 gibt einen Überblick über die Programmgröße der drei Anwendungen sowie über die Anzahl der in den Anwendungen enthaltenen nesC-Komponenten.

Wie anhand der Größenangaben in der Tabelle zu erkennen ist, verursacht die Aufteilung der Anwendungen in Komponenten einen nicht zu vernachlässigenden Anstieg der Codegröße. Der Umfang von OscilloscopeRF steigt um stark 19,8%, der von Surge um zirka 19,4% und der von AcousticLocalization um ungefähr 15,4%. Der Größenanstieg geht dabei primär auf die zusätzlichen Funktionen zurück, die für das „Wiring“ der Komponenten benötigt werden.

Die nesC-Komponenten von OscilloscopeRF wurden auf 6 Binärkomponenten aufgeteilt, die Komponenten von Surge auf 10 und die nesC-Komponenten von AcousticLocalization auf 15 Binärkomponenten. Tabelle 7.2 gibt genauer Auskunft über die verschiedenen Binärkomponenten der drei Anwendungen. In der Tabelle ist auch erkennbar, welche Binärkomponenten von mehreren Anwendungen verwendet werden. Anhand ihres Namens kann man bei den meisten Binärkomponenten direkt auf ihre Funktion schließen. OscilloscopeRF besteht beispielsweise aus einer Komponente für den Zugriff auf den Lichtsensor (*PhotoSensorComponent*), einer Kommunikationskomponente (*GenericCommComponent*), einer Komponente für Analog-zu-Digital-Konvertierungen der Sensorsignale (*ADCComponent*), der Timer-Komponente (*TimerComponent*) und der

| | OscilloscopeRF | Surge | AcousticLocalization |
|---|---|---|---|
| **Größe unzerteilt (in Bytes)** | 11784 | 17096 | 24272 |
| **Größe zerteilt (in Bytes)** | 14116 | 20420 | 28020 |
| **Anzahl der nesC-Komponenten** | 39 | 53 | 69 |
| **Anzahl der Binärkomponenten** | 6 | 10 | 15 |

Tabelle 7.1: Komplexität der Beispielanwendungen

Systemkomponente (*SystemComponent*), sowie der Anwendungskomponente (*OscilloscopeApplication*), die das Anwendungsmodul enthält und alle Komponenten miteinander verdrahtet. In der Systemkomponente sind verschiedene einfache Module zusammengefasst, die in allen oder zumindest den meisten TinyOS-Programmen verwendet werden, beispielsweise das PowerManagement-Modul.

### 7.2.2 Untersuchte Codeänderungen

Für die Untersuchungen wurden insgesamt sechs Änderungen am Programmcode der drei Beispielanwendungen umgesetzt. Die Änderungen lassen sich grob in drei Klassen aufteilen, angefangen von **kleinen Änderungen** mit nur geringen Auswirkungen auf den Programmcode, über **interne Änderungen**, die sich nur auf den Programmcode einer einzelnen Komponente auswirken, bis hin zu **externen Änderungen**, die mehr als eine Komponente eines Programms betreffen.

#### Kleine Änderungen

Unter kleinen Änderungen am Programmcode versteht man Änderungen, die nur einen sehr kleinen Teil des Programms betreffen und die Struktur des Programms dabei nicht wesentlich verändern. Es handelt sich üblicherweise um einfache Fehlerkorrekturen oder kleine Erweiterungen des Programms.

Für die Experimente wurden drei Änderungen dieser Klasse an der OscilloscopeRF-Anwendung umgesetzt:

- **Austausch einer Konstante**: Die einfachste Änderung der OscilloscopeRF-Anwendung tauscht lediglich den Wert einer globalen Konstante gegen einen anderen Wert aus. Die Anweisungsfolge und die Struktur des Programms ändern sich dabei nicht.
- **Doppelter Aufruf**: In einer zweiten Änderung wird eine Initialisierungsfunktion innerhalb der Anwendungskomponente der OscilloscopeRF-Anwendung doppelt

| | **OscilloscopeRF** | **Surge** | **AcousticLocalization** |
|---|---|---|---|
| AcousticLocalizationApplication | | | × |
| AcousticBeaconComponent | | | × |
| AcousticMultiSamplerComponent | | | × |
| ADCComponent | × | × | × |
| BcastComponent | | × | |
| FloodRoutingComponent | | | × |
| GenericCommComponent | × | | × |
| GenericCommPromiscuousComponent | | × | |
| MicComponent | | | × |
| MultiHopRouterComponent | | × | |
| OscilloscopeApplication | × | | |
| OutsideRangingActuatorComponent | | | × |
| OutsideRangingSensorComponent | | | × |
| PeaceKeeperComponent | | | × |
| PhotoSensorComponent | × | × | |
| QueuedSendComponent | | × | |
| RemoteControlComponent | | | × |
| SounderComponent | | × | × |
| SurgeApplication | | × | |
| SystemComponent | × | × | × |
| TimerComponent | × | × | × |
| TimeSlotNegotiationComponent | | | × |

Tabelle 7.2: Binärkomponenten der Beispielanwendungen

aufgerufen. Der Programmcode der Komponente wird also um einen weiteren Funktionsaufruf ergänzt, wodurch die Größe der Komponente anwächst.

- **Durchschnittsbildung**: Die aufwendigste der drei Änderungen an der OscilloscopeRF-Anwendung ändert den Rückgabewert der Funktion, die den Wert des Helligkeitssensors ausliest. Anstatt den gelesenen Wert unverändert zurückzugeben, berechnet die Funktion nach der Änderung einen Durchschnittswert über die letzten vier gelesenen Sensorwerte und gibt diesen zurück. Die Änderung implementiert damit eine Art Glättungsfunktion.

### Interne Änderungen

Interne Änderungen haben tief greifendere Auswirkungen auf den Programmcode, ihr Einfluss beschränkt sich jedoch auf eine einzelne Komponente. Die externen Verbindungen dieser Komponente bleiben unverändert, intern kann es jedoch zu massiven Änderungen der Struktur des Programmcodes kommen.

Es wurden zwei interne Änderungen am Beispiel der Surge-Anwendung realisiert:

- **MintRoute**: Der Routingalgorithmus, der in der Lage ist, die Daten der einzelnen Sensorknoten über mehrere Zwischenknoten zur Basisstation zu übertragen, wurde gegen einen anderen Routingalgorithmus ausgetauscht. Dieser Algorithmus,

genannt MintRoute [WTC03], bezieht die Qualität der Funkverbindungen zwischen den einzelnen Knoten in seine Routingentscheidungen mit ein.

- **LED-Interface entfernen**: Die zweite interne Änderung modifiziert die Komponente, die für die Verbindungen („das Wiring“) zwischen den Komponenten der Surge-Anwendung zuständig ist. Es werden sämtliche Verbindungen zum LED-Interface entfernt und zur NoLeds-Komponente umgeleitet, so dass die Leuchtdioden des Sensorknotens nicht mehr verwendet werden. Da lediglich Verbindungen entfernt werden, beschränken sich die Auswirkungen dieser Änderungen auf den Programmcode der Verbindungskomponente.

#### Externe Änderungen

Für das Ausmaß externer Änderungen bestehen prinzipiell keine Beschränkungen. Sie können mehrere Komponenten betreffen und die Struktur des Programms beliebig verändern.

Es wurde eine externe Änderung implementiert, die das Verhalten der AcousticLocalization-Anwendung ändert:

- **Blattknoten**: AcousticLocalization wird so abgeändert, dass es keine Nachrichten anderer Sensorknoten mehr weiterleitet, sondern nur noch eigene Nachrichten in Richtung der Basisstation verschickt. Diese Änderung kann sinnvoll sein, um einen Teil der Knoten zu reinen Blattknoten zu machen und so Ressourcen einzusparen. Es handelt sich bei dieser Änderung um eine externe Änderung, da für ihre Umsetzung nicht nur der Programmcode einer Komponente, sondern auch die Verbindungen anderer Komponenten zu dieser Komponente geändert werden müssen.

### 7.2.3 Bewertungskriterien

Ein Kriterium, das bei der Bewertung von Codeupdate-Algorithmen häufig eingesetzt wird, ist die **Datenmenge, die im Rahmen eines Updates an die Knoten im Netzwerk übertragen werden muss**. Je geringer diese Datenmenge ausfällt, desto geeigneter scheint ein Algorithmus für diese Aufgabe zu sein, da jedes per Funkschnittstelle übertragene Byte nicht nur vom Empfängerknoten, sondern auch von allen Zwischenknoten auf dem Weg zum Empfänger, einen beträchtlichen Energieaufwand erfordert.

Als alleiniges Kriterium für die Bewertung von Codeupdate-Algorithmen eignet sich die zu übertragende Datenmenge allerdings nicht, da eine Reduktion des benötigten Übertragungsvolumens so gut wie immer mit zusätzlichem Berechnungsaufwand durch die Sensorknoten verbunden ist. Im Falle von `FlexCup` wird dieser Aufwand benötigt, um die empfangene Komponente unter Zuhilfenahme des `FlexCup`-Linkers in den Programmcode des Sensorknotens zu integrieren. Die Komplexität des `FlexCup`-Algorithmus lässt

bereits erahnen, dass eine Betrachtung dieses Berechnungsaufwands in Relation zu den eingesparten Kosten für die Datenübertragung nicht vernachlässigt werden darf.

Das mit am universellsten einsetzbare Kriterium für die Bewertung von Algorithmen in Sensornetzanwendungen ist der **Energieverbrauch**, den ein Verfahren während des gesamten Programmablaufs verursacht. Für die unterschiedlichen Codeupdate-Algorithmen gilt dabei natürlich, dass derjenige Algorithmus überlegen ist, der in verschiedenen Szenarien jeweils den geringsten Energieverbrauch verursacht. Für eine seriöse Betrachtung des Energieverbrauchs muss darauf geachtet werden, dass alle Faktoren in die Betrachtung einfließen. Neben dem Energieverbrauch für den Empfang der Daten ist dies der Energieverbrauch des Prozessors während der gesamten Bearbeitungszeit, sowie die benötigte Energie für Zugriffe auf externe Ressourcen, insbesondere auf den externen Flashbaustein.

Neben der Energie ist die **benötigte Zeit** ein weiteres wichtiges Bewertungskriterium. Obwohl der Zeit- und der Energieverbrauch eines Algorithmus untrennbar miteinander verbunden sind, so gibt es doch Unterschiede im Verhalten dieser beiden Werte. Dies hängt damit zusammen, dass der Energieverbrauch nicht nur von der Dauer der Prozessorbenutzung abhängt, sondern auch von der Art der verwendeten Zugriffe. Die Verwendung der Funkschnittstelle oder des externen Flash verbraucht im gleichen Zeitraum deutlich mehr Energie als rein interne Berechnungen des Prozessors.

Die Zeitdauer eines Codeupdates ist für den Anwender durchaus von großem Interesse, da der Sensorknoten während eines Teils dieses Zeitraums nicht für die normale Anwendung zur Verfügung steht. Bei `FlexCup` muss beispielsweise die Ausführung der normalen Anwendung gestoppt werden, bevor der `FlexCup`-Linker und der `FlexCup`-Bootloader den empfangenen Programmcode installieren können. Der Anwendungsprogrammierer ist daran interessiert, durch ein Update nur möglichst kurze Ausfallzeiten innerhalb des Sensornetzes zu verursachen. Ein weiteres Ziel ist eine möglichst kurze Gesamtdauer des gesamten Updateprozesses um (möglicherweise dringliche) Aktualisierungen des Programmcodes möglichst schnell auf allen Knoten des Sensornetzes zu installieren.

Bei der Evaluation von `FlexCup` werden im Folgenden alle drei Kriterien eingesetzt. Die zu übertragende Datenmenge wird dabei primär eingesetzt, um die Unterschiede der Algorithmen und deren Potential zu veranschaulichen. Die Eigenschaften der konkreten Implementierungen werden dann anhand des Energieverbrauchs und der Zeitdauer verglichen.

### 7.2.4 Vorgehen

Für die drei Anwendungen sowie die sechs abgeänderten Versionen der Anwendungen wurden mit dem `FlexCup`-Analyzer jeweils die Metadaten erstellt. Anhand dieser Daten konnte bereits ermittelt werden, welche Datenmenge für jedes Update mit `FlexCup` übertragen werden muss. Auf die gleiche Weise konnten mit MOAP-Diff die Änderungsprotokolle der sechs geänderten Anwendungen erstellt und ausgewertet werden. Für Deluge war eine derartige Vorbereitung der Daten nicht notwendig.

Um den Zeit- und Energiebedarf der unterschiedlichen Codeupdates zu ermitteln, wurde die Installation der sechs Programmänderungen auf einem Sensorknoten mit jedem der Update-Verfahren durchgeführt. Der Updatevorgang wurde dafür in drei Phasen aufgeteilt: die **Datentransferphase**, die **Verarbeitungsphase** und die **Installationsphase**.

Der Aufwand in der Datentransferphase hängt nicht nur vom Datenvolumen ab, das übertragen werden muss, sondern auch von Faktoren wie der Netzwerktopologie, der Fehlerrate bei der Übertragung und dem verwendeten Verteilungsalgorithmus. Um einen ausgewogenen Vergleich zwischen den drei Verfahren sicherzustellen, wurden für die Messungen perfekte Bedingungen angenommen: Es kommt zu keinen Kollisionen oder Übertragungsfehlern. Ebenfalls ignoriert wurde der Zusatzaufwand, der durch die jeweiligen Übertragungsprotokolle entsteht. Vereinfachend wurde zudem angenommen, dass jeder Sensorknoten in der Datentransferphase das benötigte Datenvolumen je einmal vollständig empfängt und einmal vollständig sendet, um die Daten weiterzuleiten.

Die Experimente wurden mit Hilfe von *atemu* [PBM+04] durchgeführt. *atemu* ist ein Emulator für eingebettete, auf einem AVR-Prozessor basierende Systeme. Der Emulator verarbeitet originären Binärcode und simuliert direkt die darin enthaltenen Prozessorbefehle. Durch dieses realitätsnahe Vorgehen ist es möglich, mit *atemu* sowohl den Zeitbedarf als auch den Energiebedarf eines Programms sehr genau zu ermitteln, wobei die Möglichkeit der Messung des Energiebedarfs eine Erweiterung der Originalimplementierung von *atemu* darstellt. Bei Bedarf können diese Werte auch für einzelne Programmteile getrennt gemessen werden.

Unter Zuhilfenahme verschiedener Erweiterungen, die die Peripherie der Mica2-Sensorknoten nachrüsten, lässt sich mit *atemu* auch ein vollständiger Mica2-Sensorknoten simulieren. Auf diese Weise kann das Zeit- und Energieverhalten beliebiger Anwendungen für Mica2-Sensorknoten sehr präzise simuliert werden.

Die Erweiterungen von *atemu* mussten zunächst kalibriert werden, um bei den Berechnungen des Zeit- und des Energieverbrauchs der einzelnen Operationen mit korrekten Werten zu arbeiten. Zu diesem Zweck wurden entsprechende Messungen an realen Mica2-Systemen durchgeführt.

Die Einrichtung und Kalibrierung von *atemu* erfolgte während der Vorbereitung einer Veröffentlichung [MLM+] zum Thema FlexCup. Insbesondere die notwendigen Erweiterungen und Änderungen der *atemu*-Software waren dementsprechend nicht Teil des Leistungsumfangs dieser Diplomarbeit. Im Rahmen dieser Veröffentlichung wurden auch bereits die Messungen durchgeführt, auf die der folgende Abschnitt Bezug nimmt.

| Anwendung | Updatetyp | Deluge | MOAP-Diff | FlexCup Meta | FlexCup Code | FlexCup Gesamt |
|---|---|---|---|---|---|---|
| OscilloscopeRF | Austausch einer Konstante | 23142 | 11 | 799 | 1198 | 1997 |
| | Doppelter Aufruf | 23142 | 1230 | 801 | 1202 | 2003 |
| | Durchschnittsbildung | 23142 | 2835 | 537 | 886 | 1423 |
| Surge | MintRoute | 28652 | 7684 | 1056 | 3258 | 4314 |
| | LED-Interface entfernen | 28652 | 375 | 1355 | 2142 | 3497 |
| Acoustic-Localization | Blattknoten | 34162 | 7802 | 2565 | 4773 | 7338 |

Tabelle 7.3: Übertragene Datenvolumina (in Bytes)

### 7.2.5 Performance-Ergebnisse

#### Übertragene und gespeicherte Datenmengen

Tabelle 7.3 zeigt, welche Datenmengen die drei untersuchten Ansätze jeweils übertragen müssen, um die sechs verschiedenen Änderungen auf einem Sensorknoten zu installieren.

Offenkundig zeigt sich, dass Deluge bezogen auf dieses Kriterium den beiden anderen Ansätzen bei weitem unterlegen ist. Unabhängig vom Umfang der Änderung muss Deluge immer den vollständigen Programmcode neu übertragen. Je geringer die Auswirkungen einer Änderung auf den Programmcode sind, desto gravierender ist der Nachteil, der Deluge daraus entsteht, da die anderen Ansätze hier Aufwand einsparen können. So muss Deluge für den einfachen Austausch der Konstante mehr als die 2100-fache Datenmenge im Vergleich zu MOAP-Diff übertragen, während beim sehr komplexen Komponententausch „nur“ in etwa die 4,38-fache Datenmenge anfällt.

Der Vergleich zwischen `FlexCup` und MOAP-Diff fällt hier uneinheitlich aus. In drei der sechs Fällen muss `FlexCup` weniger Daten übertragen, in den anderen drei Fällen ist MOAP-Diff überlegen. `FlexCup` kann eine gewisse Mindestgröße bei der Datenübertragung nicht unterschreiten, da grundsätzlich der vollständige Code der Komponente sowie die Metadaten übertragen werden müssen. Im Gegenzug wächst das benötigte Datenvolumen bei MOAP-Diff bei größeren Änderungen schnell an, während `FlexCup` nur dann mehr Daten benötigt, falls sich Änderungen über mehrere Komponenten erstrecken.

Tabelle 7.4 zeigt im Vergleich dazu den maximalen Speicherplatzbedarf der drei Verfahren im externen Flash. Hier ist Deluge im Vorteil, da es nur den reinen Programmcode im externen Flash zwischenspeichern muss, während MOAP-Diff und `FlexCup` deutlich mehr Speicherplatz benötigen. `FlexCup` benötigt dabei noch einmal zwischen 30% und 45% mehr Speicherplatz als MOAP-Diff. In Systemen, in denen der Speicher der Sensorknoten einen dominierenden Engpass darstellt, kann dies ein entscheidendes Argument gegen die Verwendung von `FlexCup` und auch MOAP-Diff sein.

| Anwendung | Updatetyp | Deluge | MOAP-Diff | FlexCup |
|---|---|---|---|---|
| OscilloscopeRF | Austausch einer Konstante | 23142 | 28538 | 37337 |
| | Doppelter Aufruf | 23142 | 28542 | 37343 |
| | Durchschnittsbildung | 23142 | 28608 | 36743 |
| Surge | MintRoute | 28652 | 33440 | 43561 |
| | LED-Interface entfernen | 28652 | 34272 | 42744 |
| AcousticLocalization | Blattknoten | 34162 | 40156 | 58014 |

Tabelle 7.4: Platzverbrauch im externen Flash (in Bytes)

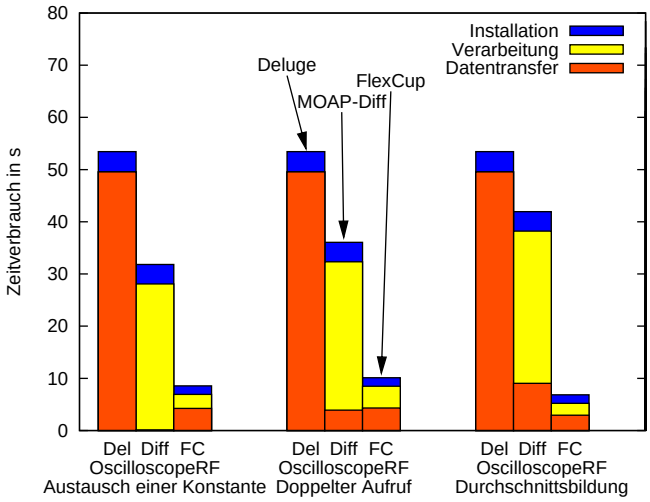


Abbildung 7.1: Zeitaufwand der Codeupdates (Teil 1)

### Zeit- und Energieaufwand

Wie bereits in Abschnitt 7.2.3 erwähnt, sind der Zeit- und vor allem der Energieaufwand deutlich aussagekräftigere Kriterien um die Effizienz der drei Verfahren beurteilen zu können. Abbildung 7.1 und Abbildung 7.2 zeigen den Zeitverbrauch in Sekunden für den vollständigen Durchlauf der drei Codeupdate-Verfahren. Die Darstellung ist aufgeteilt in die drei Phasen Datentransfer, Verarbeitung und Installation. Wie zu sehen ist, entfällt bei Deluge jedoch die Verarbeitungsphase, da die empfangenen Daten ohne weiteren Zwischenschritt direkt installiert werden können.

Die graphische Darstellung zeigt, dass `FlexCup` in allen sechs untersuchten Fällen deutlich bessere Ergebnisse erzielt als Deluge und auch merklich kürzer für die Aktualisierung des Codes benötigt als MOAP-Diff. Gegenüber Deluge sind die Einsparungen beim Zeitaufwand der Datenübertragungsphase entscheidend. `FlexCup` kommt hier im schlechtesten Fall mit 22% (9% im besten Fall) der Zeit aus, die Deluge für die Übertragung des kompletten Programmcodes benötigt. Diese Einsparungen rechtfertigen in allen Fällen den zusätzlichen Aufwand, den `FlexCup` für die Verarbeitung der Daten auf dem Sensorknoten investieren muss.

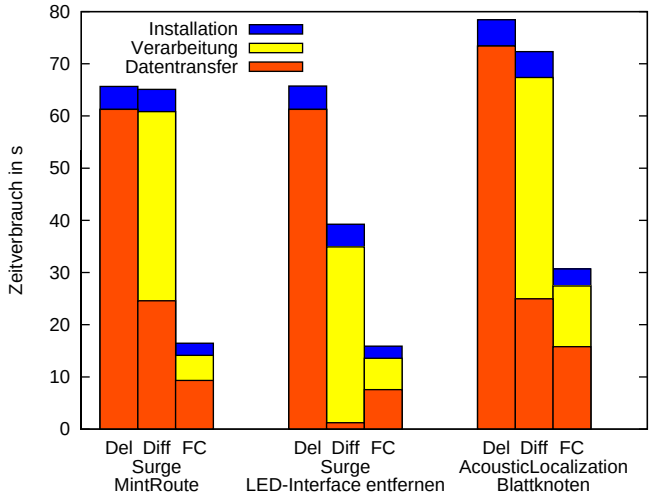


Abbildung 7.2: Zeitaufwand der Codeupdates (Teil 2)

Bei MOAP-Diff übersteigt die Zeitdauer der Verarbeitungsphase die Zeit, die für die Übertragung der Daten an den Sensorknoten benötigt wird, zum Teil um das Vielfache. Im Extrembeispiel „Austausch einer Konstante“ stehen 27.99 Sekunden Verarbeitungszeit nur 0,096 Sekunden Übertragungszeit (in der Abbildung nicht sichtbar) gegenüber. Der Vergleich der Ergebnisse von MOAP-Diff mit den Ergebnissen von Deluge zeigt, dass MOAP-Diff in einigen Fällen die Einsparungen bei der Übertragungszeit durch den Zeitaufwand für die Verarbeitung größtenteils wieder einbüßt. Bei `FlexCup` bleibt dagegen die Verarbeitungszeit in allen Fällen kleiner als die benötigte Übertragungszeit.

Da alle drei Ansätze während der Installationsphase prinzipiell die gleichen Aufgaben zu erledigen haben – alle müssen den kompletten Programmcode aus dem externen Flash in den Programmspeicher des Sensorknotens übertragen – sind die erkennbaren Unterschiede in der Verarbeitungszeit nur durch Details der jeweiligen Implementierung zu erklären und nicht durch prinzipbedingte Vorteile eines Ansatzes.

Die guten Ergebnisse von `FlexCup`, die der Vergleich des Zeitaufwands der drei Ansätze ergeben hat, bestätigten sich bei der Betrachtung des Energieaufwands, dargestellt in Abbildung 7.3 und Abbildung 7.4. Erneut erzeugt `FlexCup` in allen sechs Fällen bessere Ergebnisse als Deluge und MOAP-Diff.

Der Vergleich von Deluge und MOAP-Diff zeigt hier, dass es durchaus merkliche Unterschiede in der Bewertung geben kann zwischen der Betrachtung des Zeitaufwands und des Energieaufwands eines Verfahrens. War MOAP-Diff bei der Betrachtung der Zeitdauer der Codeupdates noch in allen Fällen (wenn auch teilweise knapp) Deluge überlegen, so kehrt sich dieses Verhältnis bei der Betrachtung des Energieverbrauchs teilweise um. Dies deutet darauf hin, dass MOAP-Diff während seiner Berechnungen einen größeren Anteil energieintensiver Operationen ausführt als Deluge. Vermutlich handelt es sich dabei um

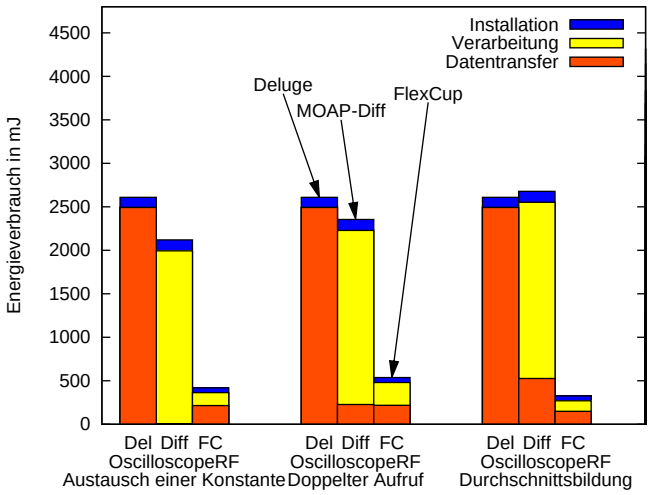


Abbildung 7.3: Energieaufwand der Codeupdates (Teil 1)

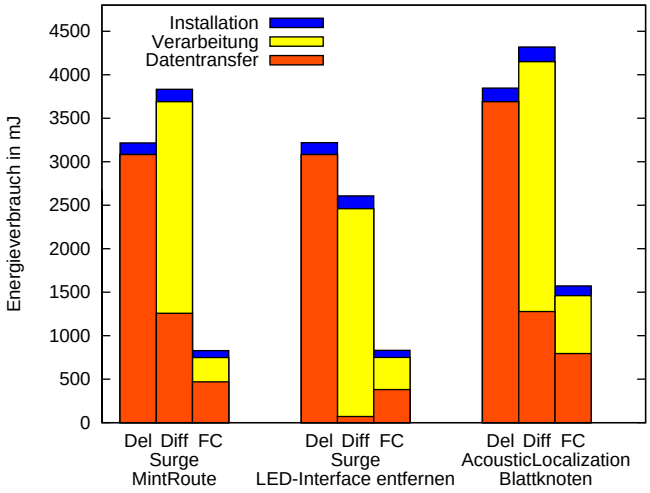


Abbildung 7.4: Energieaufwand der Codeupdates (Teil 2)

die zahlreichen Zugriffe von MOAP-Diff auf den externen Flash.

Der große Zeit- und Energiebedarf der Verarbeitungsphase wird bei MOAP-Diff durch die Verarbeitung des Änderungsprotokolls verursacht. Der Algorithmus muss die neue Version des Programmcodes mit Hilfe des Änderungsprotokolls und der aktuell installierten Programmversion konstruieren. Dies erfordert eine Vielzahl an Kopieroperationen sowie zahlreiche Lese- und Schreibzugriffe auf verschiedene Stellen des externen Flash. Trotz dieses nachvollziehbaren Aufwands sind die großen Unterschiede im Zeit- und Energieaufwand zwischen MOAP-Diff und `FlexCup` doch überraschend. Genau wie MOAP-Diff muss schließlich auch `FlexCup` den neuen Programmcode aus der alten Programmversion und den neu empfangenen Daten zusammensetzen und dabei an vielen Stellen lesend und schreibend auf den externen Flash zugreifen. Dies legt die Vermutung nahe, dass zumindest ein Teil des Leistungsunterschieds zwischen MOAP-Diff und `FlexCup` durch Ineffizienzen der MOAP-Diff-Implementierung verursacht werden.

Die Unterschiede im Zeit- und im Energiebedarf der Codeupdates zwischen Deluge und `FlexCup` entsprechen dagegen den Erwartungen: Deluge überträgt immer den vollständigen Programmcode und benötigt dementsprechend viel Zeit und Energie. `FlexCup` erkauft sich Einsparungen bei der Übertragung der Daten durch zusätzlichen Aufwand für die Verarbeitung der Updates. Die gezeigten Messungen bestätigen, dass dieser Verarbeitungsaufwand gegenüber den Einsparungen bei der Datenübertragung sowohl zeit- als auch energiemäßig so gering bleibt, dass `FlexCup` die Codeupdates deutlich effizienter bearbeiten kann als Deluge.

### Einfluss der Pufferbereiche

Bei den Messungen des Zeit- und Energiebedarfs der drei Ansätze wurde die in Abschnitt 6.6.2 beschriebene Optimierung eingesetzt, die Pufferbereiche zwischen dem Programmcode der einzelnen Komponenten einfügt, um bei Größenänderungen einer Komponente ein Verschieben der nachfolgenden Codesegmente zu verhindern. Im Folgenden soll nun anhand der Änderung „Doppelter Aufruf“ gezeigt werden, welchen Effekt der Einsatz von Pufferbereichen auf die Effizienz von `FlexCup` haben kann.

Durch den zusätzlichen Aufruf der Initialisierungsfunktion wächst die Codegröße der *OscilloscopeApplication*-Komponente um 4 Bytes an. Verfügt die installierte Anwendung nicht über einen ausreichend dimensionierten Puffer direkt hinter der geänderten Komponente, so müssen die im Programmspeicher nachfolgenden Komponenten verschoben werden, um Platz für die neue Komponente freizumachen. Im konkreten Beispiel bedeutet dies, dass die *ADCComponent*, die *GenericCommComponent*, die *PhotoSensorComponent*, die *SystemComponent*, die *TimerComponent* und der Binärcode einer Systembibliothek verschoben werden müssen, da sie im Programmspeicher alle hinter der *OscilloscopeApplication*-Komponente platziert sind. Dabei entsteht Aufwand für das Verschieben des Programmcodes innerhalb des externen Flash sowie Aufwand für die Anpassung der Referenzen auf sämtliche Funktionssymbole innerhalb der verschobenen Komponenten.

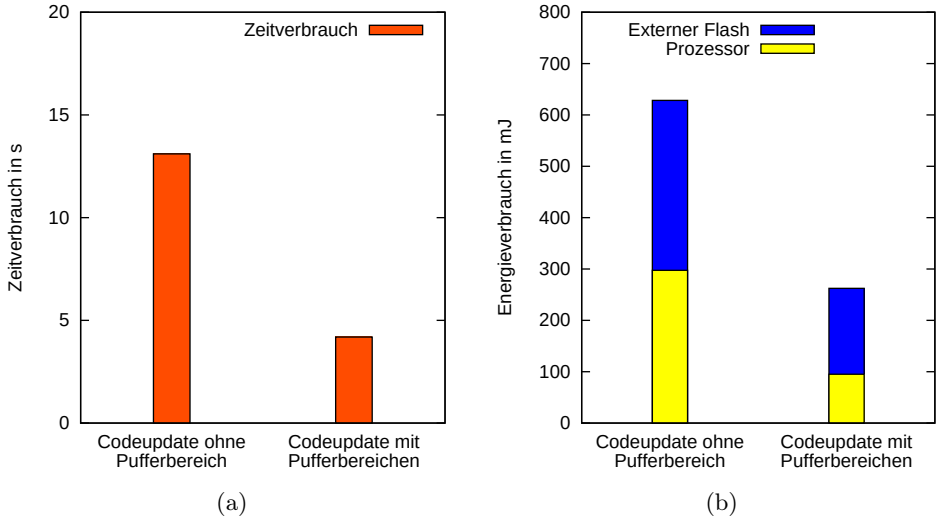


Abbildung 7.5: Zeit- und Energieaufwand mit Pufferbereichen und ohne Puffer für die Änderung „Doppelter Aufruf“

Abbildung 7.5 zeigt einen Vergleich des Zeit- und des Energieaufwands für die Bearbeitung der Codeaktualisierung durch den `FlexCup`-Linker mit und ohne ausreichend Puffer. Der Energieverbrauch ist zusätzlich aufgeschlüsselt in den Energieverbrauch des Prozessors, sowie den Energieverbrauch für Zugriffe auf den externen Flashbaustein. Ohne Puffer benötigt der `FlexCup`-Linker für die prinzipiell gleiche Aufgabe mehr als dreimal soviel Zeit sowie ungefähr 2,4-mal soviel Energie. Der zusätzliche Energiebedarf fällt dabei sowohl im Prozessor (3,1-facher Energieverbrauch) als auch für Zugriffe auf den externen Flashbaustein (2,0-facher Energieverbrauch) an.

Weiteren Aufschluss über die Ursachen des zusätzlichen Zeitbedarfs ergibt die Simulation des `FlexCup`-Linkers auf dem PC (siehe Abschnitt 6.5). Tabelle 7.5 zeigt die mit der Simulation erzeugten Auswertungen der Zugriffe auf den externen Flash, zum einen für den ersten Schritt („Austausch des Programmcodes der Komponente“) sowie für den gesamten Ablauf des `FlexCup`-Linkers.

Die Ergebnisse der Simulation bestätigen, dass ohne die Verwendung von Pufferbereichen vor allem im ersten Schritt des `FlexCup`-Linkers ein enormer Mehraufwand notwendig ist, um die Programmdaten der dahinter liegenden Komponenten zu verschieben. Die Optimierung zur Nutzung von Pufferbereichen zwischen den Komponenten sollte demnach in möglichst allen Fällen genutzt werden, um einen solchen Mehraufwand für den Großteil der Codeupdates zu vermeiden.

| | Austausch Programmcode | | Gesamtzahl | |
|---|---|---|---|---|
| | mit Puffer | ohne Puffer | mit Puffer | ohne Puffer |
| Anzahl der Lesezugriffe auf einen der Puffer des externen Flash | 45 | 12041 | 2076 | 13982 |
| Anzahl der Schreibzugriffe auf einen der Puffer des externen Flash | 13 | 12008 | 857 | 13281 |
| Anzahl der „Speichere Puffer in den Flash-Speicher“-Operationen | 7 | 60 | 206 | 327 |
| Anzahl der „Lade Puffer aus dem Flash-Speicher“-Operationen | 12 | 66 | 291 | 402 |
| Anzahl der aus dem Puffer des externen Flash gelesenen Bytes | 1259 | 13267 | 12375 | 24213 |
| Anzahl der in den Puffer des externen Flash geschriebenen Bytes | 1206 | 13212 | 3391 | 16251 |
| Anzahl der „Lade Page direkt in den Mote-Speicher“-Operationen | 0 | 0 | 25 | 25 |

Tabelle 7.5: Aufwand mit Pufferbereichen und Aufwand ohne Puffer für die Änderung „Doppelter Aufruf“

# 8 Zusammenfassung und Ausblick

Dieses Kapitel fasst das Vorgehen und die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Arbeiten in diesem Gebiet.

## 8.1 Zusammenfassung

In dieser Arbeit wurde mit `FlexCup` ein System entwickelt, das den dynamischen Austausch von Komponenten in TinyOS-basierten Sensornetzen ermöglicht. Die Grundidee des Konzeptes liegt in der Aufteilung der Anwendungen in mehrere Komponenten. Anstatt im Falle einer Aktualisierung den vollständigen Programmcode an die Sensorknoten übermitteln zu müssen, sendet `FlexCup` jeweils nur die geänderten Komponenten, die auf den Sensorknoten dann dynamisch in den existierenden Programmcode integriert werden.

Das Konzept von `FlexCup` wurde prototypisch für Sensorknoten vom Typ Mica2 implementiert. Die Funktionalität von `FlexCup` wird dabei von mehreren Teilen gemeinsam realisiert: Der `FlexCup`-Analyzer analysiert auf dem zentralen Rechner die TinyOS-Anwendungen und erstellt Metadaten, die für die Integration neuer Komponenten in die Programme benötigt werden. Der `FlexCup`-Linker ist auf den Sensorknoten installiert und hat die Aufgabe, empfangene, neue Komponenten in das bestehende Programm zu integrieren. Er verwendet zu diesem Zweck die Metadaten des Programms und der neuen Komponente. Das Übertragen der neuen Programmversion in den Programmspeicher der Sensorknoten ist die Aufgabe des `FlexCup`-Bootloaders, der nach der Installation das neue Programm direkt startet. Für die Aufteilung der Anwendungen in einzelne Komponenten wird ein neues Sprachkonzept von nesC verwendet, mit dessen Hilfe mehrere nesC-Komponenten zu einer so genannten binären Komponente zusammengefasst werden können.

Bei der Implementierung von `FlexCup` musste an vielen Stellen auf die systembedingten Ressourcenbeschränkungen der Sensorknoten Rücksicht genommen werden. Bei der Aufteilung der Anwendungen und beim Entwurf der Metadaten wurde darauf geachtet, bei einem Update möglichst geringe Datenmengen über die Funkschnittstelle übertragen zu müssen. Die Implementierungen des `FlexCup`-Linkers und des `FlexCup`-Analyzers sind jeweils darauf ausgerichtet, möglichst geringen Berechnungsaufwand auf den Sensorknoten zu verursachen. Durch verschiedene Optimierungen konnte die Effizienz von `FlexCup` im Laufe der Entwicklung noch weiter gesteigert werden.

Im Rahmen der Evaluation wurde `FlexCup` mit zwei weiteren Ansätzen verglichen, die ebenfalls den Austausch von Programmcode für Mica2-Sensorknoten implementieren und dabei zwei andere Herangehensweisen repräsentieren. Anhand von drei Beispielanwendungen und sechs beispielhaften Codeänderungen wurde untersucht, wie sich `FlexCup` in verschiedenen Situationen im Vergleich zu diesen Ansätzen verhält. Als Vergleichskriterien kamen hierbei der Zeit- und der Energieverbrauch der Verfahren zum Einsatz, außerdem wurde der jeweilige Speicherplatzbedarf im nicht-flüchtigen Flashspeicher betrachtet. Der Zeit- und Energieverbrauch konnte für alle drei Verfahren mit Hilfe einer präzisen Simulationen des Verhaltens der Mica2-Sensorknoten ermittelt werden.

Die Evaluation des Ansatzes und der Vergleich mit den zwei Alternativlösungen hat durchaus viel versprechende Ergebnisse ergeben. In allen sechs untersuchten Szenarien konnte `FlexCup` die ihm gestellten Aufgaben schneller erledigen als seine Konkurrenten und zeigte dabei jeweils auch den geringsten Energiebedarf. Konkret war `FlexCup` bis zu achtmal schneller und benötigte nur zwischen einem Achtel und der Hälfte der Energie der anderen Ansätze.

Den Einsparungen beim Zeit- und Energiebedarf steht ein deutlich höherer Bedarf an Speicherplatz im externen Flash der Sensorknoten gegenüber. `FlexCup` muss zusätzlich zu einer Kopie des Programmspeichers umfangreiche Metadaten abspeichern, die notwendiges Wissen zur Struktur der Anwendung für die Integration neuer Komponenten enthalten. Dieser Umstand kann den Einsatz von `FlexCup` verhindern, falls Speicherplatz ein begrenzendes Element einer Anwendungsdomäne ist.

Ein interessantes Ergebnis dieser Arbeit liegt in der Erkenntnis, dass die häufig propagierten Einsparungen bei der Datenübertragung zwischen den Sensorknoten nicht selbstverständlich zu effizienteren Lösungen führen. Obwohl das Senden und Empfangen von Daten sicherlich einer der größten Kostenfaktoren beim Betrieb eines Sensorknoten ist, so dürfen gleichzeitig die Kosten der lokalen Verarbeitung nicht ignoriert werden. In welchem Umfang sich zusätzlicher Aufwand für lokale Berechnungen als Ersatz für die Übertragung von Daten per Funk rentiert, ist stark hardware- und implementierungsabhängig und muss in jedem Einzelfall untersucht werden.

Eine weitere Erkenntnis der Experimente mit `FlexCup` war, dass bereits minimale Änderungen an der Implementierung große Auswirkungen auf die Performanceergebnisse der Anwendung haben können. So konnte die Implementierung der Zugriffe auf den externen Flash noch merklich verbessert werden, nachdem Messungen mit dem Oszilloskop die (auch logisch nachvollziehbaren) negativen Effekt bestimmter Zugriffssequenzen veranschaulicht hatten. Durch die sehr hardwarenahe Programmierung sind Sensornetzanwendungen für solche Effekte besonders anfällig. Diese Eigenschaft birgt natürlich auch die Gefahr in sich, dass der Vergleich unterschiedlicher Ansätze und Algorithmen in Sensornetzen durch mögliche Ineffizienzen der Implementierungen verfälscht wird.

Unabhängig von den Performanceergebnissen bleibt die Bedeutung von `FlexCup` als Lösung für den Austausch von Komponenten innerhalb des `TinyCubus`-Projektes. Während andere Ansätze grundsätzlich eine Steuerung der Updates durch einen zentralen Rechner

erfordern, können die einzelnen Knoten mit Hilfe von `FlexCup` prinzipiell autonom über die Installation neuer Komponenten entscheiden. Dies ist eine wichtige Voraussetzung für die Entwicklung des `TinyCubus`-Systems, das den dynamischen Austausch von Komponenten für Konfigurationsänderungen sowie für den Austausch der in der Anwendung verwendeten Algorithmen benötigt.

## 8.2 Ausblick auf zukünftige Arbeiten

Der nächste Schritt in der Entwicklung von `FlexCup` sollte in der Integration der einzelnen Anwendungen zu einem Gesamtsystem bestehen. Der wichtigste Schritt wird hierbei die Integration der `FlexCup`-Implementierung mit der Implementierung des Codeverteilungsalgorithmus von Robert Sauter [Sau05] sein.

Danach sind neben der Umsetzung der bereits in Abschnitt 6.6.5 beschriebenen Optimierungen eine Reihe von Ergänzungen und Erweiterungen vorstellbar, die im nächsten Abschnitt vorgestellt werden. Anschließend wird in Abschnitt 8.2.2 mit `FlexCup Diff` eine mögliche zukünftige Entwicklungsrichtung in mehr Detail vorgestellt.

### 8.2.1 Mögliche Ergänzungen und Erweiterungen

`FlexCup` wurde im Rahmen dieser Arbeit für die Hardware der Mica2-Sensorknoten von der Firma Crossbow [Croa] entwickelt. An einigen Stellen wurde das Vorgehen dabei speziell auf die besonderen Erfordernisse dieser Hardwareplattform zugeschnitten. Als wichtigstes Beispiel muss hier der Zugriff auf den externen Flash genannt werden, für den mehrere Optimierungen implementiert wurden. Um den Nutzen von `FlexCup` in unterschiedlichen Einsatzbereichen untersuchen zu können, wäre es nützlich, **das System auf weitere Hardwareplattformen zu portieren**. Zuallererst bieten sich hierfür weitere gebräuchliche TinyOS-Plattformen an, wie beispielsweise die Tmote sky-Plattform (Telos) [Mota], die BTnodes-Plattform [BTn], oder die IMotes-Plattform [Cor]. Im Falle der BTnodes-Plattform dürfte eine Portierung deutlich leichter fallen als für andere Plattformen, da sie auf dem selben Prozessor aufbaut wie die Mica2-Sensorknoten und dementsprechend den gleichen Compiler verwendet. Für die Portierung auf andere Plattformen müssen dagegen vermutlich sowohl der `FlexCup`-Analyzer als auch der `FlexCup`-Linker angepasst werden, um auf die speziellen Eigenschaften des Compilers und der Hardware-Architektur einzugehen.

Durch eine **engere Verknüpfung der Arbeit des `FlexCup`-Analyzers mit dem AVR-GCC-Compiler** könnten die Ergebnisse von `FlexCup` weiter verbessert werden. Der AVR-GCC-Compiler erzeugt für alle Funktionen einer Komponente Symboleinträge in der Symboltabelle, selbst dann, wenn die Funktionen im C-Quellcode als `static` definiert waren und dementsprechend gar nicht von außerhalb der Komponente zugegriffen werden können. Auf Basis der momentan zur Verfügung stehenden Informationen kann

der `FlexCup`-Analyzer jedoch nicht zwischen statischen und extern-zugreifbaren Funktionen unterscheiden und muss aus diesem Grund alle Funktionssymbole in die Symboltabelle der Sensorknoten übernehmen. Könnten diese Funktionssymbole eingespart werden, so würden die Symboltabellen auf den Sensorknoten weniger Speicherplatz erfordern. Verbunden damit wäre auch eine effizientere Bearbeitung, da weniger Symbole eingelesen, bearbeitet und aktualisiert werden müssten.

Eine weitere – noch stärker mit dem AVR-GCC-Compiler verbundene – Möglichkeit, den Umfang der Metadaten und den Aufwand des `FlexCup`-Linkers zu verkleinern, ist das **Eliminieren lokaler CALL-Anweisungen**. Die CALL-Anweisung gehört zum Befehlssatz der AVR-Prozessoren [Atm02] und wird dazu verwendet, Funktionen (Subroutinen) aufzurufen. Für jede CALL-Anweisung muss der Linker die korrekte Zieladresse anhand des jeweiligen Eintrags in der Relocationtabelle ermitteln und in den Programmcode eintragen. Viele CALL-Aufrufe, die Funktionen der gleichen Komponente aufrufen, könnten durch RCALL-Anweisungen ersetzt werden. RCALL-Anweisungen entsprechen in ihrer Funktionalität den CALL-Anweisungen, erwarten als Zielangabe jedoch keine absolute Adresse, sondern einen relativen Offset von der Quelladresse des Sprungs aus. Der Vorteil bestünde darin, dass lokale Sprünge mit einer RCALL-Anweisung ohne Beteiligung des Linkers realisiert werden könnten, da der einzutragende Offset von der späteren Position der Komponente im Programmspeicher unabhängig ist. Leider können nicht alle Funktionsaufrufe innerhalb einer Komponente mit RCALL-Anweisungen realisiert werden, da der Offset der Adresse maximal 4 Kilobyte (positiv oder negativ) betragen darf.

Durch die Verwendung von RCALL-Anweisungen könnten zahlreiche Einträge des „Internal Call“-Abschnitts der Relocationtabelle ersatzlos entfallen, was wiederum sowohl die Größe der Metadaten als auch den Aufwand des `FlexCup`-Linkers verkleinern würde.

Ein positiver Nebeneffekt der Verwendung von RCALL-Anweisungen bestünde darin, dass RCALL-Befehle einen Prozessorzyklus weniger benötigen als CALL-Befehle, wodurch die Anwendung minimal schneller würde. Eine solche Optimierung wurde von den Entwicklern des AVR-GCC-Compilers schon diskutiert, eine Umsetzung steht bisher allerdings noch aus.

Wie die Evaluation des `FlexCup`-Algorithmus gezeigt hat, ist die Verwendung von Pufferbereichen eine sehr wichtige Optimierung, die für manche Codeupdates eine sehr starke Verbesserung der Leistung ermöglicht. Wie bereits in Abschnitt 6.6.2 erwähnt, müssen die Pufferbereiche momentan beim Erstellen des Programmcodes noch manuell eingefügt werden. Einer der nächsten Schritte in der Weiterentwicklung von `FlexCup` sollte aus diesem Grund die Implementierung eines **Mechanismus** sein, **der es erlaubt, die Integration der Pufferbereiche in den Programmcode zu automatisieren**. Dieser Mechanismus sollte dabei einerseits den Aufwand des Benutzers minimieren, andererseits sollte es weiterhin möglich sein, die Größe der einzelnen Pufferbereiche selbst festzulegen.

Bei der Beschreibung der Implementierung des `FlexCup`-Linkers wurde bereits erwähnt, dass das Entfernen nicht mehr benötigter Objektsymboleinträge aus der Symboltabelle der Sensorknoten in der derzeitigen Version noch nicht implementiert ist. Die Aufgabe

zukünftiger Arbeiten besteht nun darin, hierfür ein Vorgehen zu implementieren, das einerseits **ungenutzte Einträge wirksam aus der Symboltabelle entfernt**, dabei andererseits jedoch auf die Ressourcenbeschränkung der Sensorknoten Rücksicht nimmt und möglichst wenig Zusatzaufwand generiert. Ein interessanter Aspekt wird hierbei sein, inwieweit zunächst nicht entfernte Symboleinträge bei späteren Aktualisierungen der Komponenten wieder genutzt werden können.

Die Implementierung des `FlexCup`-Linkers wurde in mehrere Einzelschritte aufgeteilt, in denen die einzelnen Aufgaben des Linkers bearbeitet werden. Ein interessanter Ansatz für zukünftige Verbesserungen könnte sein, die Aufgaben auf eine andere Art und Weise aufzuteilen, um zusätzliche Optimierungen realisieren zu können. Während beispielsweise momentan die Einträge der Symboltabelle aktualisiert werden, bevor anschließend die Relocationtabellen durchlaufen werden, könnte man alternativ auch die **Bearbeitung der Funktions- und Objektsymbole komplett voneinander trennen**. In diesem Fall würden zunächst die Funktionssymboleinträge in der Symboltabelle des Sensorknoten aktualisiert, dann die Referenzen auf Funktionssymboleinträge in den Relocationtabellen durchlaufen, bevor die gleichen Schritte anschließend für die Objektsymbole durchgeführt würden. Auf diese Weise stünde der Pufferspeicher im RAM zunächst nur für die Funktionssymbole und anschließend für die Objektsymbole zur Verfügung, wodurch der verfügbare Pufferspeicher auch für große Symboltabellen ausreichend sein dürfte. Auf ähnliche Weise sollte auch die Vorgehensweise an anderen Stellen der Implementierung bezüglich weiterer Verbesserungen auf den Prüfstand gestellt werden.

### 8.2.2 FlexCup Diff

Der Ansatz von `FlexCup`, eine Sensornetzanwendung in ihre Einzelkomponenten aufzuteilen und bei Änderungen am Code der Anwendung nur die geänderten Komponenten an die Sensorknoten zu übertragen, muss nicht in allen Fällen zu guten Ergebnissen führen. Als veranschaulichendes Beispiel kann hier die erste Codeänderung der Evaluation (siehe Abschnitt 7.2.2) dienen, bei der sich nur ein einziger konstanter Wert der Anwendung ändert. Trotz dieser minimalen Änderung ist `FlexCup` gezwungen, den vollständigen Programmcode plus der Metadaten der Komponente an die Sensorknoten zu übertragen. In diesem und ähnlichen Fällen scheinen die in Abschnitt 3.2 vorgestellten Differenzansätze besser geeignet zu sein, die zu übertragende Datenmenge zu minimieren. Sehr deutlich zeigt dies der Vergleich der zu übertragenden Datenvolumina in Tabelle 7.3: Den 1997 Bytes, die `FlexCup` übertragen muss, stehen nur 11 Bytes für den MOAP-Differenzansatz gegenüber.

Um auch beim Einsatz von `FlexCup` von diesen Vorteilen der Differenzansätze profitieren zu können, wird im Folgenden `FlexCup Diff` entworfen – eine Kombination des Ansatzes von `FlexCup` mit den Konzepten der Differenzansätze. `FlexCup Diff` könnte im Rahmen zukünftiger Arbeiten realisiert werden und könnte die Implementierung von `FlexCup` ergänzen.

### Grundidee

`FlexCup Diff` überträgt wie gewohnt nur den Programmcode geänderter Komponenten. Der Unterschied besteht nun darin, dass `FlexCup Diff` vor der Übertragung die neue Version des Komponentencodes mit der bisher auf den Sensorknoten installieren Version vergleicht und nur die Änderungen am Programmcode in Form eines Änderungsprotokolls an die Sensorknoten überträgt.

Um `FlexCup Diff` zu realisieren, muss auf dem zentralen Rechner zusätzlich zur Berechnung der Metadaten nun auch das Änderungsprotokoll des Programmcodes erstellt und anschließend an die Sensorknoten übertragen werden. Auf den Sensorknoten muss `FlexCup` um eine Komponente erweitert werden, die empfangene Änderungsprotokolle einliest und daraus den neuen Programmcode der Komponente rekonstruiert. Nach der Speicherung dieser Daten im externen Flash unterscheidet sich das restliche Vorgehen des `FlexCup`-Linkers dann nicht mehr von der beschriebenen Implementierung.

Auf die zu übertragenden Symboltabellen der geänderten Komponenten lässt sich der Differenzansatz nicht anwenden, da auf den Sensorknoten keine getrennten Symboltabellen für die Komponenten gespeichert werden, so dass eine Rekonstruktion durch den Vergleich mit der bisherigen Version nicht möglich ist. Alternativ könnte jedoch ein Änderungsprotokoll durch den Vergleich der gesamten Symboltabelle erstellt werden. Auf jeden Fall anwendbar ist der Ansatz für die Relocationtabellen. Es muss jedoch beachtet werden, dass die Differenzalgorithmen teilweise speziell für die Verarbeitung von Programmcode optimiert wurden und eventuell für andere Binärdaten keine optimalen Ergebnisse erzeugen.

### Vorteile gegenüber anderen Differenzansätzen

Ein wichtiger Unterschied zwischen `FlexCup Diff` und anderen Differenzansätzen besteht in der Bearbeitung einzelner Komponenten. Da `FlexCup` bekannt ist, in welchen Komponenten im Vergleich zur alten Programmversion Änderungen vorgenommen wurden, reicht es aus, Änderungsprotokolle für diese geänderten Komponenten zu erstellen anstatt eines Änderungsprotokolls für das Gesamtprogramm. Notwendige Änderungen am Code der anderen Komponenten, verursacht durch Verschiebungen referenzierter Funktions- oder Objektsymbole, werden auf den Sensorknoten automatisch durch den `FlexCup`-Linker durchgeführt.

Ein weiterer großer Vorteil liegt darin, dass der Differenzalgorithmus bei `FlexCup Diff` auf den ungelinkten Objektdateien der geänderten Komponenten arbeiten kann und so keine Änderungen durch Adressverschiebungen verarbeiten muss. Reijers und Langendoen [RL03] weisen darauf hin, dass ein bedeutender Anteil der Einträge ihrer Änderungsprotokolle auf solche Adressverschiebungen zurückgeht. Sie reagieren darauf mit aufwendigen Operatoren, die eine Anpassung solcher Adressverschiebungen ermöglichen. Bei `FlexCup Diff` sind solche Anpassungen nicht notwendig, wodurch die Skriptsprache

| Anwendung | Updatetyp | FlexCup | | | FlexCup Diff | | |
|---|---|---|---|---|---|---|---|
| | | Meta | Code | Gesamt | Meta | Code | Gesamt |
| OscilloscopeRF | Austausch einer Konstante | 799 | 1198 | 1997 | 530 | 15 | 545 |
| | Doppelter Aufruf | 801 | 1202 | 2003 | 760 | 5 | 765 |
| | Durchschnittsbildung | 537 | 886 | 1423 | 523 | 114 | 637 |
| Surge | MintRoute | 1056 | 3258 | 4314 | 1110 | 1587 | 2697 |
| | LED-Interface entfernen | 1355 | 2142 | 3497 | 1290 | 8 | 1298 |
| Acoustic-Localization | Blattknoten | 2565 | 4773 | 7338 | 2611 | 532 | 3143 |

Tabelle 8.1: Übertragene Datenvolumina für `FlexCup` und `FlexCup Diff` (in Bytes)

einfacher gestaltet werden kann und wodurch voraussichtlich auch kürzere Änderungsprotokolle entstehen.

### Einschränkungen

Da der zentrale Rechner für die Berechnung des Änderungsprotokolls den Programmcode der Komponenten kennen muss, der aktuell auf den Sensorknoten installiert ist, können die Sensorknoten nicht mehr unabhängig voneinander entscheiden, welche Komponenten sie installieren und wann sie eine Komponentenimplementierung austauschen. Bei der Verwendung von `FlexCup Diff` gibt der Benutzer damit einen Teil der von `FlexCup` gebotenen Flexibilität auf. Aufgrund dieser Einschränkung kann `FlexCup Diff` auch nur als Ergänzung zur Standardimplementierung von `FlexCup` dienen, diese jedoch nicht vollständig ersetzen.

### Evaluation des Konzeptes

`FlexCup Diff` wurde im Rahmen dieser Arbeit nicht implementiert, dies bleibt als mögliche Erweiterung des `FlexCup`-Systems zukünftigen Arbeiten überlassen. Mit Hilfe der MOAP-Implementierung des Differenzansatzes konnte jedoch zumindest das grundsätzliche Potential des Konzeptes untersucht werden. Zu diesem Zweck wurde für jede der sechs Codeänderungen aus dem Evaluationskapitel ein Änderungsprotokoll erstellt, indem der neue mit dem alten Programmcode der geänderten Komponenten verglichen wurde. Die Ergebnisse sind in Tabelle 8.1 aufgeführt. Gegenüber `FlexCup` erzielt `FlexCup Diff` deutliche Einsparungen bei der Datenmenge. Im besten Fall schrumpft die zu übertragende Datenmenge auf etwas mehr als ein Viertel der ursprünglichen Menge zusammen. Ein Vergleich mit den Ergebnissen von MOAP-Diff aus Tabelle 7.3 auf Seite 106 zeigt zudem, dass `FlexCup Diff` bei der Größe des Änderungsprotokolls für die Übertragung des Programmcodes MOAP-Diff teilweise dramatisch überlegen ist. Die zusätzlich zu übertragenden Metadaten gleichen diesen Effekt jedoch teilweise wieder aus.

Bei einer reinen Betrachtung des Übertragungsvolumens lohnt sich die Verwendung von `FlexCup Diff` auf jeden Fall. Ob eine Implementierung auch bezüglich der entscheidenden Kriterien Zeit- und Energieverbrauch Vorteile besitzt, dürfte vor allem davon abhängen, ob es gelingt, den Differenzansatz als Teil des `FlexCup`-Linkers effizient zu implementieren. Erste Versuche mit der MOAP-Diff-Implementierung ergaben diesbezüglich ein zwiespältiges Ergebnis.

## 8.3 Fazit

Das in dieser Arbeit entwickelte Verfahren für den dynamischen Austausch von Komponenten in TinyOS konnte die bisherigen Erwartungen weitgehend erfüllen. Wie die praktischen Untersuchungen gezeigt haben, ist `FlexCup` als reiner Codeupdate-Mechanismus anderen Lösungen in typischen Anwendungsszenarien überlegen. Für die zukünftige Entwicklung wird nun entscheidend sein, dass sich `FlexCup` als Alternative zu bestehenden Verfahren etablieren kann und dass eine Integration in das `TinyCubus`-Gesamtprojekt erfolgreich gelingt.

# Literaturverzeichnis


[Atm02] Atmel Corporation: *Atmel AVR Instruction Set*, August 2002. URL: http://www.atmel.com.

[Atm04a] Atmel Corporation: *Atmel AT45DB041B Datasheet*, April 2004. URL: http://www.atmel.com.

[Atm04b] Atmel Corporation: *Atmel ATmega128L Datasheet*, November 2004. URL: http://www.atmel.com.

[AVR] AVR Libc Project: *Avr libc home page*. URL: http://www.nongnu.org/avr-libc/.

[BCPR04] Butler, Z., P. Corke, R. Peterson, and D. Rus: *Dynamic virtual fences for controlling cows*. In *Proceedings of the Ninth International Symposium on Experimental Robotics*. Sage Publications, June 2004.

[BHS03] Boulis, Athanassios, Chih-Chieh Han, and Mani B. Srivastava: *Design and implementation of a framework for efficient and programmable sensor networks*. In *Proceedings of the The First International Conference on Mobile Systems, Applications, and Services*, pages 187–200, May 2003.

[BKM04] Baldus, Heribert, Karin Klabunde, and G. Müsch: *Reliable set-up of medical body-sensor networks*. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN 2004)*, pages 353–363, 2004.

[BTn] BTnodes: *A distributed environment for prototyping ad hoc networks*. URL: http://www.btnode.ethz.ch/.

[CEH+01] Cerpa, Alberto, Jeremy Elson, Michael Hamilton, Jerry Zhao, Deborah Estrin, and Lewis Girod: *Habitat monitoring: application driver for wireless communications technology*. In *SIGCOMM LA '01: Workshop on Data communication in Latin America and the Caribbean*, pages 20–41, New York, NY, USA, 2001. ACM Press.

[Con] Contiki: *The Contiki operating system*. URL: http://www.sics.se/~adam/contiki/.

[Cor] Corporation, Intel: *Intel motes*. URL: http://www.intel.com/research/exploratory/motes.htm.

[Croa] Crossbow Technology Inc. URL: http://www.xbow.com.

[Crob] Crossbow Technology, Inc.: *MICA2 Data sheet*. URL: http://www.xbow.com.

[Cro03a] Crossbow Technology, Inc.: *MIB - Mote Interface / Programming Board User's Manual*, December 2003. URL: http://www.xbow.com.

[Cro03b] Crossbow Technology, Inc.: *Mote In-Network Programming User Reference*, March 2003. URL: http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf.

[Cro04] Crossbow Technology, Inc.: *MPR / MIB User's manual*, August 2004. URL: http://www.xbow.com.

[DGV04] Dunkels, Adam, Björn Grönvall, and Thiemo Voigt: *Contiki - a lightweight and flexible operating system for tiny networked sensors*. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004*, November 2004.

[EYE] EYES Consortium: *EYES - energy efficient sensor networks*. URL: http://www.eyes.eu.org/.

[Frea] Free Software Foundation: *GNU Binutils*. URL: http://www.gnu.org/software/binutils/.

[Freb] Free Software Foundation: *GNU Diffutils*. URL: http://www.gnu.org/software/diffutils/.

[Frec] Free Software Foundation: *GNU GCC*. URL: http://gcc.gnu.org/.

[Fred] Free Software Foundation: *LIB BFD, the Binary File Descriptor Library*. URL: http://www.gnu.org/software/binutils/manual/bfd-2.9.1/.

[Gay03a] Gay, David: *Design of matchbox, the simple filing system for motes*, August 2003. URL: http://www.tinyos.net/tinyos-1.x/doc/matchbox-design.pdf.

[Gay03b] Gay, David: *Matchbox: A simple filing system for motes*, August 2003. URL: http://www.tinyos.net/tinyos-1.x/doc/matchbox.pdf.

[GLCB03] Gay, David, Philip Levis, David Culler, and Eric Brewer: *nesC 1.1 language reference manual*, 2003. URL: http://www.tinyos.net/tinyos-1.x/doc/nesc/ref.pdf.

[GLvB+03] Gay, David, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler: *The nesC language: A holistic approach to networked embedded systems*. SIGPLAN Not., 38(5):1–11, 2003.

[HC02] Hill, Jason L. and David E. Culler: *Mica: A wireless platform for deeply embedded networks.* IEEE Micro, 22(6):12–24, 2002.

[HC04] Hui, Jonathan W. and David Culler: *The dynamic behavior of a data dissemination protocol for network programming at scale.* In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

[HSW+00] Hill, Jason, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister: *System architecture directions for networked sensors.* SIGOPS Oper. Syst. Rev., 34(5):93–104, 2000.

[JC04] Jeong, Jaein and David Culler: *Incremental network programming for wireless sensors.* In *Proceedings of the First IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[Jeo04] Jeong, Jaein: *Incremental network programming for wireless sensors.* Master's thesis, University of California, Berkeley, 2004.

[JKB03] Jeong, Jaein, Sukun Kim, and Alan Broad: *Network reprogramming*, August 2003. URL: `http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf`.

[KP05] Koshy, Joel and Raju Pandey: *Remote incremental linking for energy-efficient reprogramming of sensor networks.* In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pages 354–365, 2005.

[LC02] Levis, Philip and David Culler: *Maté: a tiny virtual machine for sensor networks.* In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95. ACM Press, 2002.

[Lev99] Levine, John R.: *Linkers & Loaders.* Morgan Kaufmann, 1999.

[LLWC03] Levis, Philip, Nelson Lee, Matt Welsh, and David Culler: *Tossim: accurate and scalable simulation of entire tinyos applications.* In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137. ACM Press, 2003.

[LM03] Liu, Ting and Margaret Martonosi: *Impala: a middleware system for managing autonomic, parallel sensor systems.* In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 107–118. ACM Press, 2003.

[LPCS04] Levis, Philip, Neil Patel, David E. Culler, and Scott Shenker: *Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks.* In *Proceedings of the 1st Symposium on Networked*

*Systems Design and Implementation (NSDI 2004)*, pages 15–28. USENIX, March 2004.

[MCP+02] Mainwaring, Alan, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson: *Wireless sensor networks for habitat monitoring.* In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM Press.

[MLM+] Marrón, Pedro José, Andreas Lachenmann, Daniel Minder, Matthias Gauger, Olga Saukh, and Kurt Rothermel: *FlexCup: A flexible and efficient code update mechanism for sensor networks.* Submitted for publication.

[MLM+04] Marrón, Pedro José, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Kurt Rothermel, and Christian Becker: *Adaptation and cross-layer issues in sensor networks.* In *Proceedings of the 2004 Intelligent Sensors, Sensor Networks and Information Processing Conference (ISSNIP 2004)*, 2004.

[MLM+05] Marrón, Pedro José, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel: *TinyCubus: A flexible and adaptive framework for sensor networks.* In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pages 278–289, 2005.

[Mota] Moteiv: *Telos revision b / tmote sky.* URL: `http://www.moteiv.com/`.

[Motb] Motorola: *Motorola s-records.* URL: `http://www.amelek.gda.pl/avr/uisp/srecord.htm`.

[MSKG05] Marrón, Pedro José, Olga Saukh, Markus Krüger, and Christian Grosse: *Sensor network issues in the sustainable bridges project.* In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, 2005.

[PBM+04] Polley, Jonathan, Dionysys Blazakis, Jonathan McGee, Dan Rusk, and John S. Baras: *ATEMU: a fine-grained sensor network simulator.* In *Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON 2004)*, October 2004.

[RL03] Reijers, Niels and Koen Langendoen: *Efficient code distribution in wireless sensor networks.* In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67. ACM Press, 2003.

[Sau05] Sauter, Robert: *Entwicklung eines Systems zur dynamischen Installation von Komponenten in Sensornetzen.* Diplomarbeit, Universität Stuttgart, Stuttgart, Januar 2005.

[SBMkL04] Sallai, János, György Balough, Mikós Maróti und Ákos Lédeczi: *Acoustic Ranging in Resource Constrained Sensor Networks.* Technischer Bericht ISIS-04-504, Vanderbilt University, 2004.

[SGW01] Schwiebert, Loren, Sandeep K.S. Gupta und Jennifer Weinmann: *Research challenges in wireless networks of biomedical sensors.* In: *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, Seiten 151–165, New York, NY, USA, 2001. ACM Press.

[SHE03] Stathopoulos, Thanos, John Heidemann und Deborah Estrin: *A Remote Code Update Mechanism for Wireless Sensor Networks.* Technischer Bericht CENS-TR-30, University of California, Los Angeles, 2003.

[Too95] Tool Interface Standard Committee: *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*, Mai 1995.

[Tri99] Tridgell, Andrew: *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, The Australian National University, February 1999. URL: `http://samba.org/~tridge/phd_thesis.pdf`.

[TSS+97] Tennenhouse, David L., Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden: *A survey of active network research.* IEEE Communications Magazine, 35(1):80–86, 1997.

[UIS] UISP Project: *UISP - AVR In-System Programmer.* URL: `http://www.nongnu.org/uisp/`.

[WTC03] Woo, Alec, Terence Tong, and David Culler: *Taming the underlying challenges of reliable multihop routing in sensor networks.* In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 14–27, New York, NY, USA, 2003. ACM Press.

[YYS03] Yeh, Tom, Haru Yamamoto, and Thanos Stathopolous: *Over-the-air reprogramming of wireless sensor nodes.* UCLA EE202A Project report, December 2003.

**Erklärung**

Ich versichere, dass ich diese Arbeit selbstständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

---

(Matthias Gauger)