

Dynamische Re-Programmierung von Sensorknoten zur Laufzeit

Diplomarbeit im Fach Mechatronik

vorgelegt von

Jan Moritz Strübe

geb. am 18.09.82

in Hannover

angefertigt am

Department Informatik

Lehrstuhl für Verteilte Systeme und Betriebssysteme

und

Lehrstuhl für Rechnernetze und Kommunikationssysteme

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:

Dr. Falko Dressler

Dr. Rüdiger Kapitza

Betreuender Hochschullehrer:

Prof. Dr.-Ing. Habil. Wolfgang Schröder-Preikschat

Beginn der Arbeit: **14.02.2008**

Abgabe der Arbeit: **05.08.2008**

Kurzfassung

Bei der Betrachtung von Softwareaktualisierungen in Sensornetzwerken wird meist von einem homogenen System ausgegangen. Dies bedeutet, dass das Netzwerk aus gleich aufgebauten Sensorknoten besteht, auf welchen die selbe Software läuft. Des Weiteren ist es meist die Zielsetzung von Softwareaktualisierungen in Sensornetzwerken Fehler zu beheben oder Funktionen hinzuzufügen.

In vielen Situationen, zum Beispiel der Hausüberwachung, macht es Sinn Knoten mit unterschiedlichen Sensoren auszustatten. Dies hat zur Folge, dass jeder Knoten eine auf ihn zugeschnittene Softwareversion benötigt. Soll flexibel auf Ereignissen reagiert werden kommt erschwerend hinzu, dass bei einer Softwareaktualisierung der Knoten meist neu gestartet werden muss und nicht explizit gesicherte Daten verloren gehen.

Diese Arbeit untersucht zunächst unterschiedliche Möglichkeiten Software auf Sensorknoten zur Laufzeit zu aktualisieren und vergleicht verschiedene bekannte Mikrocontrollerbetriebssysteme darauf hin, ob sie sich für eine Softwareaktualisierungen zur Laufzeit auf BTnode Sensorknoten eignen.

Anschließend wird eine Modulunterstützung für das BTnut Betriebssystem implementiert, welche es ermöglicht Binärcode zur Laufzeit auf den Sensorknoten zu laden und auszuführen. Die Implementierung des mit Shared Memory vergleichbaren *Named Memory* ermöglicht es, auch über einen Reset des Knoten hinweg auf im Arbeitsspeicher liegende Daten zuzugreifen. Um die Handhabung zu vereinfachen, werden die Funktionalitäten abstrahiert und durch eine Bibliothek bereit gestellt.

Um die Funktionalität zu demonstrieren wurde ein Kernel entwickelt, welcher es erlaubt zur Laufzeit über Bluetooth Module zu laden, zu starten und wieder zu beenden. Das Modul wird dabei, ohne dass eine Benutzerintervention nötig ist, speziell für den auf dem Knoten installieren Kernel erstellt. Auch die Funktionalität des *Named Memory* wird mit diesem Modul demonstriert.

Die vorgestellte Lösung ermöglicht es in heterogenen Netzwerken Software zur Laufzeit nachzuladen, ohne dass der Knoten neu gestartet werden muss. Weiterhin ermöglicht die Lösung es über einen Neustart hinweg, welcher bei einer Aktualisierung der Kernels nötig ist, Daten im Arbeitsspeicher zu sichern.

Abstract

When considering software updates in Wireless Sensor Networks (WSN) a homogeneous network is presumed most often. Such a homogeneous network consists of identical motes, which run the same software, further on normally the purpose of software updates in WSNs is to fix bugs or add new functions.

In many situations, e.g. for building monitoring, it is reasonable to equip the motes with different sensors. This leads to different software versions on different motes. Further more most systems must be restarted after a software update and loose all data which have not explicitly been saved on non volatile memory. This makes it difficult to react flexible on events.

This Study starts with examining different possibilities to update the software running on the motes at runtime. Also popular operating systems for micro controllers are examined and compared in terms of their suitability for software updates at runtime.

Afterwards a module support for the BTnut operating system is implemented. It allows to load modules on to the node and execute them. The implementation of the *Named Memory*, which is comparable to shared memory, allows to save data in RAM using a name. Using this name, the data can then be accessed even after a reset. To make the use of theses features more simple, the features were abstracted and are provided as a library.

To demonstrate the implemented features, a kernel was developed which allows to load, execute and stop a module. The module is specifically created for the installed kernel and copied onto the mote without user interaction. In addition the module is used to demonstrate the features of the *Named Memory*.

The presented solutions allows to load software onto a mote at runtime without having to restart the node. It is also possible to update the kernel, which requires a restart, without loosing data which was saved in RAM.

Danksagung

An dieser Stelle möchte ich mich bei Dr. Falko Dressler und Dr. Rüdiger Kapitza für die gemeinsame Betreuung dieser Arbeit bedanken, welche sich ideal ergänzt hat, was durchaus nicht immer der Fall ist.

Bei Falko bedanke ich mich auch noch mal im Speziellen dafür, dass er mich mit nach Santorin auf die DCOSS mitgenommen hat und dort die Möglichkeit gegeben hat teile dieser Arbeit vorzustellen.

Auch gilt mein Dank den verschiedenen Mitarbeitern des *Lehrstuhl für Verteilte Systeme und Betriebssysteme*, insbesondere Meik Felser, die stets Zeit für ein Gespräch hatten und mir mit verschiedenen Fragestellungen weiter geholfen haben.

Mein Dank gilt natürlich auch all den andern Personen, die mir auf meine E-Mails in den verschiedenen Mailinglisten geantwortet habe, oder mich auf andere Weise unterstützt haben.

Ein Dank soll auch noch an Dipl.-Ing. Wolfgang Wolf gehen, dafür dass er so hohe Ansprüche an die Ausarbeitung meiner Studienarbeit hatte und mich mehr oder weniger drei mal hat schreiben lassen. Es hat das Schreiben dieser Arbeit erheblich erleichtert.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

Erlangen, den

Inhaltsverzeichnis

1 Einleitung.....	1
1.1 Erweiterung der Funktionalität von Sensorknoten.....	3
1.2 Anforderungen und Zielsetzung.....	4
1.3 Aufbau der Arbeit.....	5
2 Grundlagen.....	7
2.1 Sensorknoten.....	7
2.2 Der BTnode Sensorknoten.....	8
2.3 Reprogrammierung von Sensorknoten.....	10
2.4 Energieverbrauch.....	11
2.5 Speicherverwaltung.....	12
2.6 Portabilität.....	12
2.7 Threads, Prozesse und Tasks.....	13
2.7.1 Threadbasierte Betriebssysteme.....	13
2.7.2 Ereignisbasierte Betriebssysteme.....	14
2.7.3 Protothreads.....	15
2.8 Kommunikation.....	15
2.9 Funktionsweise von Compilern und Linkern.....	16
2.10 Benutzerdefinierte Sektionen.....	19
3 Nachladen und Aktualisieren von Code zur Laufzeit.....	21
3.1 Skriptsprachen.....	21
3.2 Virtuelle Maschinen.....	21
3.3 Aktualisieren des Binärcodes.....	22
3.4 Aktualisierung von Binärcode zur Laufzeit.....	23
3.5 Verwenden von Modulen.....	23
3.5.1 Pre-Linking von Modulen.....	24
3.5.2 Lookup Tables.....	25
3.5.3 Jump Tables.....	25
3.5.4 Linken von Modulen auf dem Knoten.....	26
3.5.5 PIC - Position Independent Code.....	27
4 Vergleich bekannter Betriebssysteme für Mikrocontroller.....	29
4.1 TinyOS.....	29
4.2 Contiki.....	29
4.3 SOS.....	30
4.4 Nut/OS und BTnut.....	31
4.5 Wahl des geeigneten Systems.....	32
5 Implementierung einer Modulunterstützung für BTnut.....	35
5.1 Anforderungen.....	35
5.2 Lösungskonzept.....	36
5.3 Implementierung.....	37
5.3.1 Reprogrammierung des ATmega128.....	38
5.3.2 Aufbau und Funktion des Bootloaders.....	39
5.3.3 Flash-Speicherverwaltung.....	40
5.3.4 Erstellung und Ersetzung des Kernels.....	42
5.3.5 Einbindung des Bootloaders in das Kernelabbild.....	44
5.3.6 Aufbau und Erstellung eines Moduls.....	45

5.3.7 Implementierung der Modulschnittstelle.....	47
5.3.8 Fehlerbehandlung der Modulschnittstelle.....	51
5.4 Starten und Stoppen von in Modulen enthaltenen Applikationen.....	52
5.4.1 Aufbau von Nut/OS Threads.....	52
5.4.2 Starten und Stoppen von Applikationen.....	53
5.4.3 Verhinderung von Speicherlecks.....	54
5.5 Einschränkungen der Implementierung.....	54
5.6 Nut/OS Quellcodemodifikationen zur Unterstützung der binären Threadsignalisierung.....	55
5.6.1 include/sys/heap.h.....	55
5.6.2 arch/avr/os/context_gcc.c.....	55
5.7 Nut/OS Quellcodemodifikationen um Speicher einem Thread zuzuordnen.....	56
5.7.1 include/sys/heap.h.....	56
5.7.2 include/sys/thread.h.....	56
5.7.3 arch/avr/os/context_gcc.c.....	56
5.7.4 os/heap.c.....	57
6 Named Memory zur Datenwiederherstellung.....	59
6.1 Anforderungen.....	60
6.2 Lösungskonzept.....	60
6.3 Umsetzung.....	62
6.3.1 Funktionsweise der Nut/OS Speicherverwaltung.....	63
6.3.2 Initialisierung.....	65
6.3.3 Named Memory verwalten.....	66
6.3.4 Einschränkungen.....	66
6.4 Nut/OS Codemodifikationen.....	67
6.4.1 os/heap.c.....	67
6.4.2 include/sys/heap.h.....	68
7 Anwendung der implementierten Funktionen.....	71
7.1 Einrichtung des Entwicklungssystems.....	71
7.2 Einbindung der Modulunterstützung in einen Kernel.....	72
7.3 Beispielimplementierung eines Kernels mit Bluetoothunterstützung.....	75
7.4 Erstellung und Laden einer Applikation.....	77
7.5 Echtzeitproblematiken bei der Verwendung von Modulen.....	79
7.6 Verwendung des Named Memory.....	79
7.7 Verwendung der Zuordnung des allokierten Speichers zu Threads.....	81
8 Zusammenfassung, Diskussion und Ausblick.....	83
8.1 Diskussion.....	83
8.1.1 Implementierung der Modulunterstützung.....	84
8.1.2 Implementierung des Named Memory.....	86
8.1.3 Sonstige Punkte.....	87
8.2 Ausblick.....	87
9 Sonstiges (Alles was mir gerade einfällt).....	89
10 Referenzen.....	91
11 Abbildungsverzeichnis.....	93
12 Bildnachweise.....	93
13 Tabellenverzeichnis.....	93

1 Einleitung

Soll die Umwelt elektronisch überwacht werden, zum Beispiel um die Temperatur eines Raumes zu regeln oder ein Erdbeben vorherzusagen, müssen die hierzu benötigten Messdaten zunächst mit Hilfe von Sensoren gewonnen werden. Die klassische Methode diese Messdaten zu sammeln besteht darin die Sensoren über Kabel anzubinden und zentral auszuwerten. Sind die Sensoren verteilt oder sollen die Messwerte in der Nähe der Sensoren ausgewertet werden, kommen Feldbusse zum Einsatz, welche es ermöglichen mehrere Sensoren an einen Bus anzubinden. Ist das Legen von Kabeln nicht möglich, weil der Aufwand zu groß ist, oder sich die Sensoren bewegen, können Datalogger verwendet werden, die die Messdaten aufzeichnen, um sie später auszuwerten. Werden die Messdaten zeitnah benötigt, oder ist das Einsammeln der Datalogger zu aufwändig, bietet es sich an, drahtlose Sensornetzwerke zu verwenden. Drahtlose Sensornetzwerke bestehen aus mehreren, meist batteriebetriebenen Sensorknoten, welche über Funk miteinander kommunizieren.

Kommerziell kommen drahtlose Sensornetzwerke bis jetzt vor allem in der Hausautomatisation zum Einsatz: Sie überwachen Temperatur, Luftfeuchtigkeit oder ob Fenster und Türen offen oder geschlossen sind. Auch sind sie mit Aktoren gekoppelt, welche die Heizung aufdrehen oder das Licht anschalten. Hier spielt vor allem der Vorteil der einfachen Nachrüstung eine entscheidende Rolle. Die Sensoren und Aktoren können an beliebigen Stellen platziert werden, ohne dass Kabel verlegt werden müssen.

Aber auch in der Industrie beginnen drahtlose Sensornetzwerke an Einfluss zu gewinnen. Bis jetzt ist ihr Einsatzgebiet vor allem die Logistik, zum Beispiel zur Überwachung von Containern¹ (Abb. 1.1). In den Containern enthaltene Sensorknoten können über GPS ihre Position, sowie die

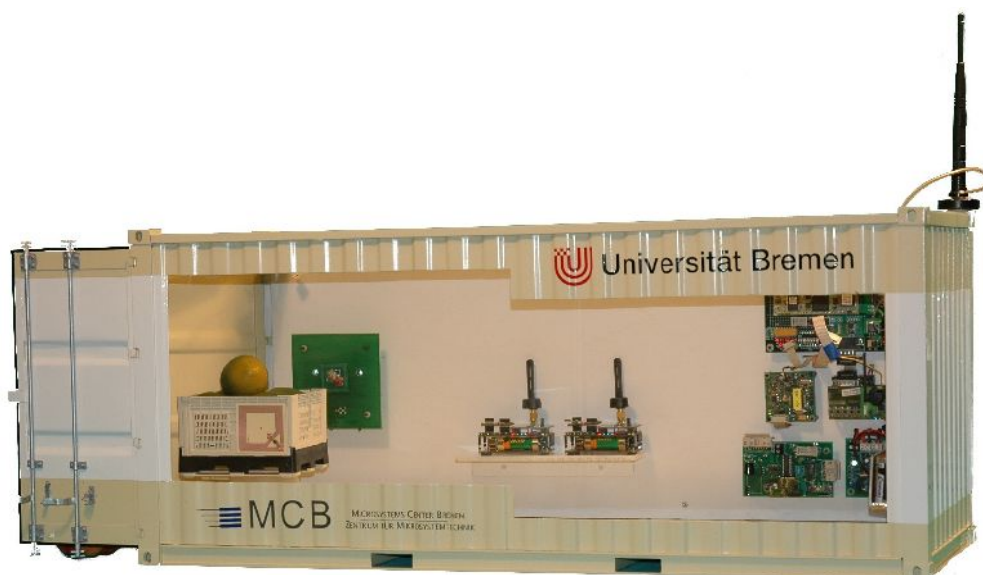


Abbildung 1.1: Demonstrator für einen intelligenten Container, welcher über ein Sensornetzwerk kommuniziert

¹ “--- The Intelligent Container ---”, <http://www.intelligentcontainer.com/>

Umwelt des Containers überwachen und gegebenenfalls sofort Alarm schlagen, sollte zum Beispiel verderbliche Ware zu warm werden, oder mit dem Verladekran Kontakt aufnehmen um sicher zu stellen, dass der Container richtig verladen wird.

In der Forschung werden drahtlose Netzwerke zum Beispiel im Zebronet Projekt zur Bewegungsüberwachung von Zebras eingesetzt [1]: Die Zebras wurden mit Sensorknoten ausgestattet, die regelmäßig die aktuelle Position über GPS auswerten (Abb. 1.2). Kommen zwei mit Sensorknoten ausgerüstete Zebras beispielsweise an Wasserlöchern in Funkreichweite, tauschen diese ihre gesammelten Daten aus. Auf diese Weise reicht es, in Funkreichweite einiger weniger Zebras zu kommen, um die Daten vieler einzusammeln. Ein aufwendiges Suchen der einzelnen Zebras entfällt.



Abbildung 1.2: Zebra mit Zebronet Sensorknoten um den Hals

Wird während des Betriebs ein Fehler in der Software entdeckt, ist es oft eine nicht triviale Aufgabe eine korrigierte Softwareversion auf die Knoten zu distribuieren. Hierbei treten mehrere Problemstellungen auf: Da die alte Softwareversion zum Betrieb des Netzwerkes benötigt wird, muss ausreichend Speicher vorhanden sein, um die neue Version vollständig zwischenspeichern zu können. Ist die neue Softwareversion installiert und in irgendeiner Weise fehlerhaft, zum Beispiel weil Übertragungsfehler aufgetreten sind, ist es ohne physikalischen Zugriff auf den Knoten meist unmöglich eine neue Softwareversion aufzuspielen. Ein weiteres Problemfeld ist die Übertragung an sich: Diese ist energieaufwändig, und kann die Laufzeit der meist batteriebetriebenen Sensorknoten erheblich verkürzen. Ein effizientes Verteilen einer neuen Softwareversion kann daher für den Erfolg eines Sensornetzwerkes eine entscheidende Rolle spielen.

1.1 Erweiterung der Funktionalität von Sensorknoten

Die Reprogrammierung von Mikrocontrollern ist nicht nur bei der Behebung von Fehlern, sondern auch zur Funktionserweiterung relevant. Interessant ist insbesondere, spezielle Applikationen situationsbedingt und damit temporär auf einen Sensorknoten auszuführen. Das Interesse an der Problemstellung zeigt auch das RUNES Projekt (Reconfigurable Ubiquitous Networked Embedded Systems), welches unter anderem von der Europäischen Union unterstützt wird¹.

Das Hauptszenario des RUNES Projektes ist ein Tunnelbrand. Der Tunnel ist mit vielen Sensoren ausgestattet, welche die Umweltbedingungen überwachen. Wird ein Grenzwert, zum Beispiel durch ein Feuer, überschritten, wird eine Alarmmeldung ausgegeben. Die eintreffende Feuerwehr ist jedoch wahrscheinlich weniger an dem Alarm an sich, sondern eher an der Temperatur, sowie deren Anstieg interessiert, um die Situation besser einschätzen zu können. Hierzu kann sie dann ihre eigene Software auf die Knoten laden.

Aber auch in anderen Szenarien spielt die Reprogrammierung von Sensorknoten eine entscheidende Rolle. Sensorknoten haben inzwischen Laufzeiten von mehreren Jahren, und mit regenerativen Energiequellen ist die Laufzeit praktisch unbegrenzt. Dies rechtfertigt die Positionierung der Sensorknoten an schwer zugänglichen Orten. Sind die Daten ausgewertet, muss die auf den Knoten laufende Software meist aktualisiert werden, um weitere Erkenntnisse gewinnen zu können. Hier spielt eine zuverlässige Reprogrammierung auf Grund der schlechten Zugänglichkeit eine wichtige Rolle.

Zusätzliche Anforderungen entstehen, wenn es sich um heterogene Netzwerke handelt. Diese entsteht bereits dann, wenn an den verwendeten Knoten unterschiedliche Sensoren angebunden sind. Dies führt dazu, dass bestimmte Funktionen nicht auf allen Knoten verfügbar sind.

Im Rahmen der Dissertation „Rekonfiguration von mobilen autonomen Diensten in heterogener Umgebung“ [2] wurde ein auf Profile Matching Verfahren aufbauendes Konzept zur Aktualisieren mobiler Geräte vorgestellt. Von einem Server wird zunächst ein Hardware- und Softwareprofil des mobilen Zielsystems erstellt oder ist bereits dort abgelegt. Besteht eine Netzwerkverbindung zwischen Server und Zielsystem, wird anhand dieses Profils überprüft, ob aktualisierte oder neu zu installierende Module zur Verfügung stehen. Diese werden anschließend an das Zielsystem übertragen.

Für TinyOS wurde ein solches Profile Matching System zur Softwareaktualisierung bereits im Rahmen einer Studienarbeit implementiert [3]. Abhängig von den gewünschten Anforderungen und den an Sensorknoten angeschlossenen Sensoren werden die benötigten Treiber in den Binärcode verlinkt und dann auf den Knoten übertragen.

Wie auch bei anderen Code-Aktualisierungssystemen [4-7] treten auch bei dieser Implementierung folgenden Einschränkungen auf:

- Obwohl sich nur ein kleiner Teil ändert, wird das komplette System ausgetauscht.

¹ “RUNES (Reconfigurable Ubiquitous Networked Embedded Systems) IST Project”, <http://www.ist-runes.org/>

- Nach einer Aktualisierung muss das System neu gestartet werden.
- Durch den Neustart des Systems gehen alle Daten, welche nicht explizit außerhalb des Arbeitsspeichers gesichert wurden, verloren.

Bisher wurde verschiedene Lösungen für die einzelnen Problemstellungen entwickelt, jedoch ist keines für das untersuchte Szenario geeignet. Vor allem der letzte Punkt, die Wiederherstellung von im Arbeitsspeicher gesicherten Daten, wurde bisher kaum untersucht.

In dieser Arbeit soll eine Infrastruktur geschaffen werden, welche es ermöglicht zum einen Teile des Codes, nach Möglichkeit zur Laufzeit, zu aktualisieren, und zum anderen nach einer solchen Aktualisierung auf Statusdaten zuzugreifen, ohne dass diese vorher explizit gesichert werden müssen.

Auf einer solchen Infrastruktur kann anschließend auch ein Profile Matching Verfahren aufgebaut werden, um Software auf verschiedene Module zu distribuieren.

Teile dieser Arbeit wurden bereits auf der „4th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS)“ vorgestellt [8].

1.2 Anforderungen und Zielsetzung

Die Zielsetzung dieser Arbeit kann in zwei Unterziele untergliedert werden. Zum einen in die Modulunterstützung und zum anderen die Wiederherstellung von Statusdaten nach einem Austausch eines Moduls oder des Gesamtsystems. Als Plattform sollen BTnode Sensorknoten dienen.

An die Reprogrammierung des Sensorknotens werden folgende Anforderungen gestellt:

- Es muss möglich sein das Gesamtsystem auszutauschen.
- Es muss möglich sein Code zur Laufzeit nachzuladen und auszuführen.
- Nach der Aktualisierung muss es möglich sein auf Statusdaten zuzugreifen.
- Als Betriebssystem soll BTnut oder ein anderes, vergleichbares Betriebssystem verwendet werden.

Die Verwendung der bereitgestellten Funktionen soll möglichst transparent geschehen.

Zur Realisierung sollen zunächst verschiedene Lösungskonzepte untersucht und verschiedene Betriebssysteme darauf hin analysiert und verglichen werden, in wie weit sie sich zur Lösung der Aufgabenstellung eignen. Anschließend soll ein Lösungskonzept entwickelt und umgesetzt werden.

1.3 Aufbau der Arbeit

Kapitel 2 „**Grundlagen**“ gibt einen Überblick über verschiedene, für diese Arbeit relevante Themen aus dem Bereich der Sensornetzwerke. Hierzu gehören die Vorstellung verschiedener Sensor-knoten, betriebssystemspezifische Themen wie Speicherverwaltung, Portabilität und Threads. Am Ende des Kapitels wird noch auf die Funktionsweise von Compilern und Linkern eingegangen, da dies wichtig für das Verständnis des nachfolgenden Kapitels ist.

In Kapitel 3 „**Nachladen und Aktualisieren von Code zur Laufzeit**“ werden verschiedene Konzepte zur Aktualisierung von Code auf Mikrocontrollern vorgestellt.

In Kapitel 4 „**Vergleich bekannter Betriebssysteme für Mikrocontroller**“ werden die Mikrocontrollerbetriebssysteme TinyOS, Contiki, SOS und Nut/OS vorgestellt und anschließend in Bezug auf ihre Eignung für das Lösen der Aufgabenstellung verglichen.

In den Kapiteln 5 „**Implementierung einer Modulunterstützung für BTnut**“ und 6 „**Named Memory zur Datenwiederherstellung**“ werden die Implementierung der Modulunterstützung beziehungsweise des *Named Memory* vorgestellt.

Kapitel 7 „**Anwendung der implementierten Funktionen**“ soll eine schnelle Einarbeitung in die implementierten Funktionen geben. Aus diesem Grund werden einige Themen der vorherigen Kapitel, sofern sie zum Verständnis benötigt werden, wiederholt.

Kapitel 8 „**Zusammenfassung, Diskussion und Ausblick**“ fasst die Arbeit kurz zusammen, diskutiert die vorgestellte Lösung und gibt einen Ausblick auf die Einsatzmöglichkeit der implementierten Funktionen.

2 Grundlagen

In diesem Kapitel soll ein Überblick über verschiedene Themengebiete der drahtlosen Sensornetze gegeben werden. Zunächst werden einige Sensorknoten kurz und anschließend der BTnode Sensorknoten genauer vorgestellt. Anschließend werden die speziell für drahtlosen Sensornetze relevanten Themen Kommunikation und Energieverbrauch angesprochen. Danach werden die eher betriebssystemlastigen Themen Speicherverwaltung, Portabilität und Threads thematisiert. Zum Schluss dieses Kapitel wird die Funktionsweise vom Compilern und Linkern kurz erklärt, da dies zum Teil zum Verständnis von Kapitel 3 benötigt wird.

2.1 Sensorknoten

Im Bereich der drahtlosen Sensornetze haben sich verschiedene Sensorknoten etabliert. Die meisten in der Forschung verwendeten Knoten sind so konzipiert, dass sie mit zwei AA-Batterien betrieben werden können. Aus diesem Grund orientieren sich auch die Abmessungen dieser Knoten an einer solchen Halterung. Im Englischen werden diese Sensorknoten auch Motes genannt, was übersetzt Staubkorn oder Partikel bedeutet.

In Tabelle 2.1 werden drei typische Sensorknoten verglichen. Sehr verbreitet sind die von UC Berkeley¹ entwickelten und von Crossbow² vertriebenen Mica und Telos Motes. Während die Mica Motes mit einem ATmega128 versehen sind, besitzen die neueren Telos Motes einen MSP430 Mikrocontroller von Texas Instruments. Auch die von Scatterweb³ vertriebenen ESB-430 und MSB-430 Motes verwenden den MSP430 Mikrocontroller.

Der für diese Arbeit verwendete BTnode wurde von der ETH Zürich entwickelt und baut auf einen Atmega128 auf. Neben der Unterstützung von Bluetooth unterscheidet sich der BTnode auch durch seine 180 KiB externen SRAM von den anderen Knoten. Von diesem können auf 60 KiB direkt und auf die weiteren 120 KiB durch so genanntes *Paging* zugegriffen werden. In Kapitel 2.2 wird der BTnode detailliert beschrieben.

Für die Funkübertragung ist vor allem der von Chipcon entwickelte CC1000 verbreitet. Bei diesem handelt es um einen Transceiver, welcher eine serielle Übertragung von Daten ermöglicht. Häufig wird auch der *ZigBee* Standard unterstützt. Bluetooth, wie beim verwendeten BTnode, ist auf Grund des vergleichsweise hohen Energiebedarfs weniger verbreitet, bietet jedoch eine einfache Interaktion mit Computern oder mobilen Geräten wie PDAs oder Handys.

1 "UC Berkeley Home Page", <http://www.berkeley.edu/>

2 "Crossbow Technology : Wireless Sensor Networks : Home Page", <http://www.xbow.com/>.

3 "ScatterWeb", <http://cst.mi.fu-berlin.de/projects/ScatterWeb/>




Knoten	BTnode	Mica 2	Tmote Sky / TelosB
			
CPU	Atmega 128		MSP430
Prozessertyp	8 Bit RISC Harvard		16 Bit RISC Von-Neumann
Programmspeicher	128 KiB		48KiB
RAM	64KiB + 180KiB ¹	4KiB	10KiB
Sonstiger Speicher	4KiB EEPROM	512KiB Flash 4KiB EEPROM	16KiB EEPROM 1024KiB Flash
Kommunikation	CC1000, Bluetooth	CC1000	IEEE 802.15.4
Abmessungen (mm)	58.15x33	58x32	65x31

Tabelle 2.1: Ausgewählte Sensorknoten

2.2 Der BTnode Sensorknoten

Als Entwicklungsplattform wird in dieser Arbeit der BTnode Sensorknoten verwendet (Abb. 2.1). Der BTnode baut auf dem Atmega128L auf (1). Dies ist ein RISC Mikrocontroller mit Harvardarchitektur, 128 KiB Flash als Programmspeicher und 4 KiB internem SRAM. Auf dem BTnode befinden sich weitere 180 KiB externer SRAM (2), von welchem 60 KiB direkt und die weiteren 120 KiB über Spezialfunktionen angesprochen werden können. Neben dem CC1000, einem seriellen Funkchip mit proprietärem Protokoll (11), steht dem BTnode auch Bluetooth als Kommunikationsmedium zur Verfügung. Die Bluetoothschnittstelle setzt sich aus dem Zeevo ZV4002 Bluetoothcontroller (6), einem zum Betrieb des Controllers benötigten Flash-Speichers (7) und einer Bluetoothantenne (5) zusammen. Um Energie zu sparen ist es möglich die beiden Funkchips über einen elektrischen Schalter von der Spannungsversorgung zu trennen. Wird zusätzlich der Mikroprozessor in den Power Down Modus versetzt, so sinkt der Energieverbrauch von ca. 45 auf 0,6 mW [9].

Die Spannungsversorgung kann über zwei AA Batterien (12) oder mit Hilfe des USB-Programmieradapters über USB zur Verfügung gestellt werden (13). Zwischen den beiden Spannungsquellen kann über einen Schalter hin und her gewechselt werden (10).

¹ Die 180KiB können über so genanntes Paging angesprochen werden. Hierbei werden unterschiedliche Teile des externen Speicher auf den internen Speicherbereich abgebildet.

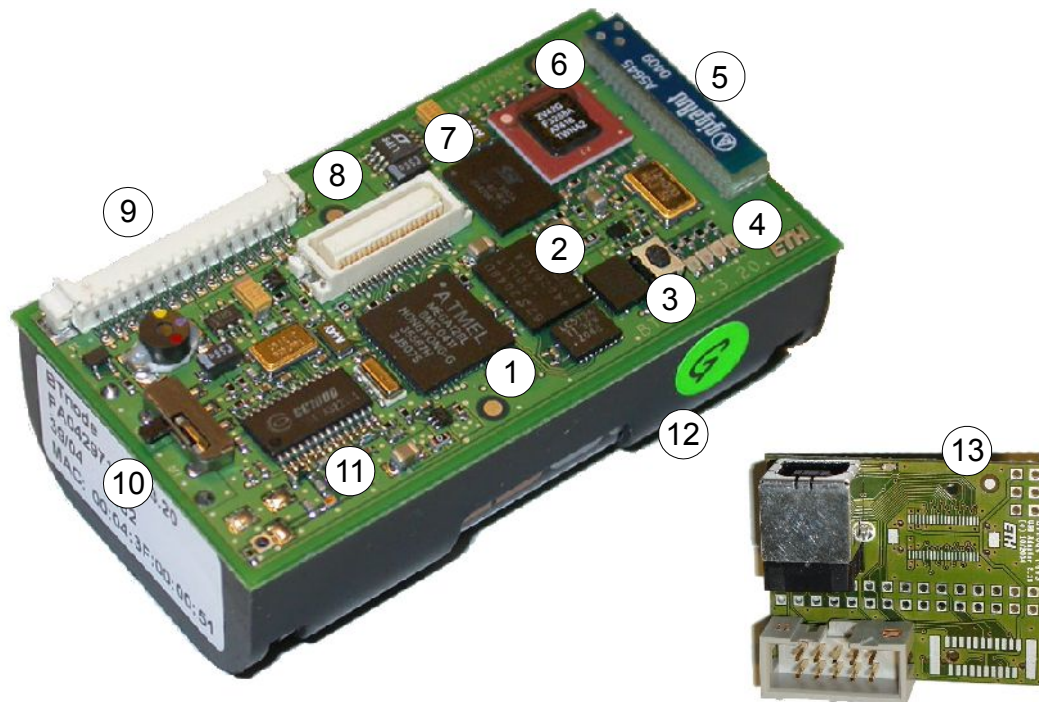


Abbildung 2.1: BTnode - ein drahtloser Sensorknoten

Ein Reset-Taster (3) ermöglicht das Zurücksetzen des Controllers von Hand. Des Weiteren stehen vier LEDs (4) zu Diagnosezwecken oder zur Anzeige von Statusinformationen zur Verfügung.

Der Programmieradapter (13) kann auf den BTnode aufgesteckt werden (8). Auf dem Programmieradapter befindet sich auch ein USB zu RS232 Umsetzer. So ist es über den USB-Anschluss nicht nur möglich den BTnode mit Spannung zu versorgen, sondern auch eine der beiden seriellen Schnittstellen über USB einzubinden. Neben der der USB-Schnittstelle stellt der Programmieradapter auch eine Buchse zum Anschluss eines *In System Programmiers* (ISP) zur Verfügung. Mit Hilfe des ISP ist es möglich, den Mikrocontroller ohne speziellen Bootloader zu reprogrammieren oder die so genannten Fuses¹ zu setzen. Des Weiteren kann über den Programmieradapter auf einen Großteil der Pins des Mikrocontrollers zugegriffen werden. So ist es zum Beispiel möglich einen Jtag² Debugger anzuschließen.

Sensoren können über die Steckleiste (9) angeschlossen werden. Es gibt verschiedene Sensoren, zum Beispiel Temperatursensoren, welche an diese Schnittstelle angeschlossen werden können.

¹ Fuses sind Schalter, welche es erlauben bestimmte Funktionen des Mikrocontrollers statisch ein- oder auszuschalten.

² Die Jtag Schnittstelle erleichtert die Fehlersuche, indem sie es ermöglicht den Mikrocontroller im Betrieb an bestimmten Stellen anzuhalten, sowie Daten aus dem Speicher auszulesen.

2.3 Reprogrammierung von Sensorknoten

Viele Sensorknoten, so auch der verwendete Atmega128, besitzen eine Harvardarchitektur. Bei dieser haben der Programm- und Datenspeicher getrennte Adressräume (Abb. 2.2), im Gegensatz zu der Von-Neumann-Architektur, in welcher nicht zwischen Bereichen für ausführbaren Code und Daten unterschieden wird.

Bei der Harvardarchitektur ist nicht möglich im Datenspeicher liegenden Binärcode auszuführen. Um auf die Daten des Programmspeichers zuzugreifen, müssen spezielle Befehle verwendet werden. Die Trennung der Adressräume hat den Vorteil, dass Programm und Datenspeicher über einen eignen Bus angebunden sind. Es ist daher möglich innerhalb eines Bustaktes sowohl einen Befehl als auch Daten aus dem Arbeitsspeicher zu laden. Des Weiteren kann die Datenwortbreite für beide Bereiche getrennt festgelegt werden. Dies ermöglicht die Erstellung von kompakterem Code, da zum Beispiel ein Bereich mit 16 und der andere mit 24 Bit adressiert werden kann.

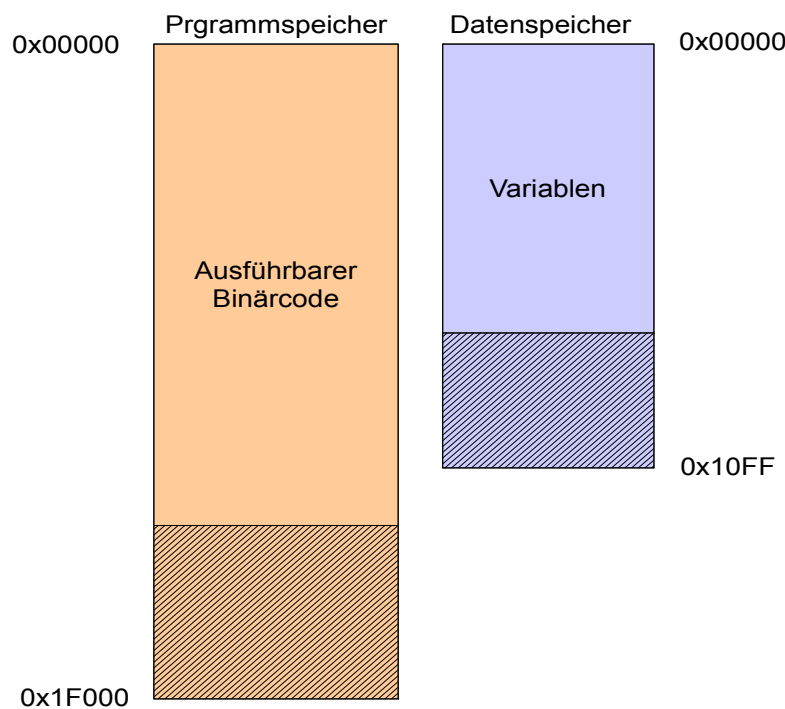


Abbildung 2.2: Aufbau einer Harvardarchitektur

Zur Speicherung von Daten wird in Sensornetzen meist SRAM verwendet, welcher sich vor allem durch geringen Stromverbrauch und unbegrenzte Schreibzyklen auszeichnet. Für den Programmspeicher wird meistens Flashspeicher verwendet. Dieser verliert auch nach dem Wegfall der Spannung seinen Inhalt nicht, jedoch sind Schreibzyklen vergleichsweise langsam.

Ist der Flash in den Mikrocontroller integriert, wird dieser meist von außen über einen so genannten In System Programmer (ISP) programmiert. Der ISP erlaubt es den Flash zu reprogrammieren ohne den Controller ausbauen und in ein spezielles Programmiergerät einbauen zu müssen. Die im Bereich der Sensornetze eingesetzten Mikrocontroller können sich aber auch selbst reprogrammieren. Je nach Ausführung kann dies mit unterschiedlichen Einschränkungen verbunden

sein. So kann es zum Beispiel sein, dass der Mikrocontroller während der Reprogrammierung keinen weiteren Code ausführen kann, nur bestimmte Bereiche programmiert werden können, oder sich der Reprogrammierbefehl in einem speziellen Bereich befinden muss.

Um den den Flash, welcher meistens zu Speicherung des Programmcodes verwendet wird, zu reprogrammieren, muss zunächst eine komplette Seite (Page) des Speichers gelöscht werden, bevor sie erneut geschrieben werden kann. Ein Lösch- und Schreibvorgang braucht bei dem verwendeten ATmega128 beispielsweise ca. 4 ms.

Die Nachteile des Flashspeichers wirken sich auch auf die Reprogrammierung aus. Soll eine Funktion ersetzt werden, ändert sich gegebenenfalls auch deren Einsprungsadresse. Aus diesem Grund muss nicht nur die Funktion selber in den Programmspeicher geschrieben werden, sondern auch alle Sprungverweise, welche auf die alte Adresse der Funktion zeigen, angepasst werden. Dies kann zur Folge haben, dass deutlich mehr Seiten des Flash geschrieben werden müssen, als die Funktion selber benötigt.

2.4 Energieverbrauch

Wie beim Betrieb, spielt auch bei der Reprogrammierung von Sensorknoten der Energieverbrauch eine entscheidende Rolle. Sensorknoten werden meist mit Batterien betrieben, um unabhängig von anderen Energiequellen zu sein. Auch bei regenerativen Energiequellen muss das Gesamtsystem stets so ausgelegt sein, dass es auch unter widrigen Umständen zuverlässig funktioniert. Eine Solarzelle muss zum Beispiel so ausgelegt sein, dass auch an einem bewölkten Tag ausreichend Energie zum Betrieb des Systems - und gegebenenfalls dem Laden eines Puffers für die Nacht - zur Verfügung steht.

Verschiedene Faktoren beeinflussen den Energieverbrauch von Sensorknoten. Die meisten Mikrocontroller unterstützen verschiedene Energiesparmodi. Der ATmega128L, eine besonders energiesparende Version des ATmega128, kann seine Stromaufnahme beispielsweise von bis zu 2,5 mA im Normalbetrieb auf unter 5 μ A im „Power-down“ Modus reduzieren. Aber auch die anderen verwendeten Bauelemente haben Einfluss auf die Energieaufnahme. Hierzu zählen auch die verwendeten Sensoren, Spannungsregler oder auch Aktoren wie LEDs.

Die Datenübertragung ist einer der wichtigsten Faktoren des Energiehaushalts. Hierbei spielt nicht nur die zu übertragende Datenmenge sondern vor allem die Übertragungstechnologie eine entscheidende Rolle. Daher werden im Bereich der Sensornetzwerke vor allem Funkstandards wie IEEE 802.15.4, Bluetooth oder der proprietäre CC1000 Funkchip eingesetzt. Diese sind speziell für den energiesparenden Datenaustausch entwickelt wurden.

Der Energieverbrauch kann aber auch durch die Reduzierung der zu übertragenden Datenmenge reduziert werden. Dies kann zum Beispiel durch Kompression der Daten oder dem Übertragen der Differenz zwischen dem bereits installierten und dem neuen Binärcode geschehen. Hierbei ist jedoch darauf zu achten, dass gerade auf leistungsschwachen Systemen die zuvor eingesparte Energie für die Dekompression der Daten benötigt wird [4].

2.5 Speicherverwaltung

Viele Mikrocontroller werden ohne Speicherverwaltung programmiert. Dies hat verschiedene Gründe: Zum einen ist die Speicherverwaltung im Normalfall Aufgabe des Betriebssystems. Soll eine Speicherverwaltung verwendet werden, muss entweder ein entsprechendes Betriebssystem oder einer Bibliothek verwendet werden. Da die meisten Mikrocontroller einen statischen Programmablauf haben, besteht selten Bedarf für eine dynamische Speicherverwaltung.

Zum anderen bringt dynamischer Speicher verschiedene Probleme mit sich. *Memory Leaks* sind ein klassisches Problem, welches auch auf dem PC bekannt ist. Hierbei wird Speicher reserviert, jedoch nicht wieder freigegeben. Wird der fehlerhafte Codeabschnitt erneut aufgerufen, reserviert er weiteren Speicher, bis kein Speicher mehr verfügbar ist.

Während sich *Memory Leaks* auf Grund des recht kompakten Codes, zum Beispiel durch Debugausgaben bei jedem Aufruf von `malloc()` und `free()`, vergleichsweise einfach finden lassen, ist der Zugriff auf Speicher, auf den nicht zugegriffen werden sollte, schwieriger aufzudecken. Dies ist um Beispiel bei einem so genannten *Buffer Overrun*, bei dem über den reservierten Speicher hinausgeschrieben wird, der Fall. Die Folgen der überschriebenen Daten treten nicht immer zeitnah auf. In einem solchen Fall ist es oft schwer die fehlerhafte Funktion zu finden. Auf leistungsfähigen Systemen stehen entsprechende Applikationen zum Finden solcher Fehler zur Verfügung (zum Beispiel Valgrind¹). Auf Mikrocontrollern stehen solche Werkzeuge, auf Grund der mangelnden Ressourcen, nicht zur Verfügung.

2.6 Portabilität

Wie in in Kapitel 2.1 beschrieben, ist die Anzahl der unterschiedlichen für Sensorknoten verwendeten Mikrocontroller auf eine überschaubare Anzahl begrenzt. Dennoch spielt die Möglichkeit Quellcode für verschiedene Knoten zu verwenden eine entscheidende Rolle. Dies ist besonders vorteilhaft, wenn in einer neuen Hardwareversion ein anderer Mikrocontroller verwendet werden soll, weil sich zum Beispiel die Anforderungen an Rechenleistung, Funktionsumfang oder Energieverbrauch geändert haben.

Die verbreiteten Betriebssysteme für Mikrocontroller versuchen daher die hardwareabhängigen Eigenschaften weitgehend zu abstrahieren und binden je nach Controller entsprechende Implementierung ein.

Gerade bei der Reprogrammierung von Mikrocontrollern ist eine portable Implementierung häufig schwierig, da es sich dabei um einen sehr hardwarenahen Vorgang handelt, welcher sich von Mikrocontroller zu Mikrocontroller stark unterscheidet. Beispielsweise muss bei dem verwendeten ATmega128 der Flash-Befehl zum Reprogrammieren des Programmcodes in einer speziellen Sektion stehen und ein Großteil des Programmcodes ist während der Reprogrammierung vom Zugriff gesperrt. Der Mikrocontroller ist daher während des Flash-Vorgangs ca. 4 ms blockiert und kann keinen Code ausführen, also auch keine Interrupts verarbeiten. Andere Mikrocontroller haben diese

¹ "Valgrind Home", <http://valgrind.org/>

Restriktionen nicht und können während der Reprogrammierung Code ausführen oder haben ihren Programmcode im Arbeitsspeicher gespeichert, welches eine sehr einfache Modifikationen des Programmcodes erlaubt.

2.7 Threads, Prozesse und Tasks

Ein Thread (Ausführungsfaden) bezeichnet die Abarbeitung eines Ausführungsstrangs. Wird einem oder mehreren Threads exklusiv ein Speicherbereich zugeteilt, so spricht man von einer Prozess. Ein Prozess kann im Normalfall nicht auf den Speicher eines anderen Prozesses zugreifen. Um dies zu realisieren, wird im Normalfall eine Hardwareunterstützung in Form einer MMU¹ benötigt, welche in Mikrocontrollern im Normalfall nicht zur Verfügung steht.

Der Begriff Task (Aufgabe) wird in unterschiedlichen Kontexten, für Threads, Prozesse, beide oder als eigenständiger Begriff verwendet [10]. Im folgenden Text wird versucht auf den Begriff zu verzichten, oder ihn im entsprechenden Kontext zu definieren.

Betriebssysteme für Mikrocontroller sind ereignis- oder threadbasiert. Letztere werden häufig auch als Multithreadingbetriebssystem bezeichnet (Werden mehrere Prozesse unterstützt spricht man von Multitasking). Bei threadbasierten Systemen können mehrere Threads gleichzeitig ausgeführt werden. Ein Thread ist ein Ausführungsstrang, welcher abgearbeitet wird. Indem das Betriebssystem zwischen den einzelnen Threads hin und her schaltet, findet eine quasiparallele Ausführung der verschiedenen Threads statt. Wird ein Thread unterbrochen, weil ihm die CPU entzogen wird oder er sie freiwillig abgibt, wird er zu einem späteren Zeitpunkt an der gleichen Stelle fortgesetzt.

Für jeden Thread müssen Statusinformationen und vor allem ein Stack zur Verfügung gestellt werden. Auf dem Stack werden lokale Variablen und Rücksprungadressen abgelegt. Für jeden Thread muss exklusiv so viel Speicher für den Stack reserviert werden, dass es nicht zu einem *Stack Overrun* kommt. Da eine Abschätzung des maximal benötigten Stack häufig schwierig ist, wird meist mehr Speicher für den Stack reserviert als nötig, was einen erhöhten Speicherbedarf zur Folge hat.

Ereignisbasierte (eventbasierte) Systeme verzichten auf Threads, beziehungsweise arbeiten mit nur einem Thread. Sie kommen daher mit weniger Arbeitsspeicher aus. Im Kontext der eventbasierten Betriebssysteme bezeichnet Task eine Aufgabe, welche abgearbeitet wird. Der Task wird durch ein Ereignis (Event) ausgelöst und dann vollständig ausgeführt. Sollen komplizierte Berechnungen angestellt werden, müssen diese in mehrere Untertasks unterteilt werden (siehe Kap. 2.7.2).

2.7.1 Threadbasierte Betriebssysteme

Bei threadbasierten Betriebssystemen wird zwischen den preemptiven und den non-preemptiven Betriebssystemen unterschieden. Preemptive, also verdrängende Betriebssysteme können einen laufenden Thread an einer beliebigen Stelle unterbrechen. Dies geschieht meist, indem das

¹ Memory Management Unit: Eine MMU erlaubt es virtuelle Adressen zu verwenden oder bestimmte Speicherbereiche vor dem Zugriff zu schützen.

Betriebssystem regelmäßig durch einen, von der Hardware ausgelösten, Timerinterrupt aufgerufen wird. Dieser überprüft, ob andere Threads mit einer höheren Priorität ausführungsbereit sind, und lastet diese gegebenenfalls ein.

Schreibt ein Thread eine größere Menge Daten in den Speicher und wird währenddessen durch einen anderen Thread verdrängt, so hinterlässt er die Daten gegebenenfalls in einem inkonsistenten Zustand. Soll ein anderer Thread auf die gleichen Daten zugreifen, muss zum Beispiel mit Semaphoren sichergestellt werden, dass dieser nicht auf diese Daten zugreift und gegebenenfalls wartet, bis der andere Thread das Schreiben beendet. Auf die Funktionsweise und die durch diese Synchronisationsmechanismen verursachten Probleme, soll an dieser Stelle nicht weiter eingegangen werden.

Im Gegensatz dazu muss der Punkt der Unterbrechung bei non-preemptive Multithreadingbetriebssystemen durch den Aufruf eines speziellen Befehls des Betriebssystems gezielt gesetzt werden. Es kann davon ausgegangen werden, dass andere Threads nur während dieses Betriebssystemaufrufs Daten manipulieren. Eine explizite Synchronisation der Threads untereinander kann dadurch in vielen Situationen entfallen. Während dies die Programmierung zum Teil erleichtert, hängt die Verfügbarkeit des resultierenden Systems stark vom Entwickler ab. Wird das Programm zu oft unterbrochen, wird ein Großteil der Zeit nicht mit Berechnungen sondern im Betriebssystem verbracht. Finden die Unterbrechungen zu selten statt, kann es passieren, dass andere Threads länger als gewünscht nicht ausgeführt werden.

2.7.2 Ereignisbasierte Betriebssysteme

Da auf Mikrocontrollern nur wenig Arbeitsspeicher zur Verfügung steht, verzichten ereignisbasierte Betriebssysteme auf Threads. Stattdessen werden Ereignisse in eine Schlange eingereiht und dann vollständig abgearbeitet. Wurde zum Beispiel ein Temperaturmesswert aufgenommen, wird das entsprechende Ereignis zur Verarbeitung in die Schlange eingereiht. Nachdem alle vor diesem Ereignis aufgetretenen, in der Schlange befindlichen Ereignisse abgearbeitet wurden, wird die entsprechende Funktion zur Verarbeitung der Temperaturmessung aufgerufen. Die Abarbeitung des Ereignisses wird auch als Task bezeichnet. Ereignisse können nicht nur durch externe Ereignisse, wie abgeschlossene Temperaturmessungen, Timer oder Taster, ausgelöst werden, sondern, im Sinne von Nachrichten, von einem Task an einen anderen geschickt werden.

Da Systeme, welche auf Mikrocontrollern laufen, meist auf gewisse Ereignisse reagieren und keine aufwändigen langen Berechnungen durchführen, ist dieser Ansatz für viele Systeme geeignet. Sollen aufwändigere Berechnungen durchgeführt werden, so muss der Task die Berechnung unterbrechen und ein Ereignis auslösen, welches ihn wieder aufruft, um die Berechnung fortzusetzen. So können andere Ereignisse abgearbeitet werden, bis der Task seine Berechnung fortsetzt. Der Nachteil dieses Verfahrens ist, dass komplizierte Tasks schnell unübersichtlich werden, da sie, zum Beispiel durch eine Switch-Anweisung, an die entsprechende Stelle zur Fortsetzung der Berechnung springen müssen. Eine Unterbrechung der Berechnung in einer, von dem Task aufgerufenen Funk-

tion, ist praktisch nicht möglich. In einem solchen Fall müsste die aufgerufene Funktion selbst als Task implementiert sein, und nach der Berechnung eine Nachricht an den Ausgangstask schicken, damit dieser mit dem Ergebnis weiterarbeiten kann.

2.7.3 Protothreads

Ein von Adam Dunkels entwickeltes Konzept, welches als eine Art Kompromiss zwischen den ereignis- und threadbasierten System gesehen werden kann, sind *Protothreads* [11]. Diese ermöglichen auch auf ereignisbasierten Systemen das Unterbrechen und Fortsetzen von Funktionen. *Protothreads* sind Funktionen ohne lokale Variablen. Sie legen daher keine Daten auf dem Stack ab. Die Verwendung von statischen oder globalen Variablen ist trotzdem möglich, da sie nicht auf dem Stack abgelegt werden. Die Unterbrechung ist jedoch nur innerhalb des *Protothreads* selber möglich, da bei dem Aufruf einer Funktion die Rücksprungadresse im Stack abgelegt wird.

Gibt ein Protothread die CPU ab, so wird der Programmzeiger gesichert und der Aufruf des *Protothreads* als Ereignis eingereiht. Sind alle anderen in der Zwischenzeit aufgetretenen Ereignisse abgearbeitet, wird der Protothread fortgesetzt, indem der Programmzeiger auf seinen vorherigen Wert zurückgesetzt wird. Trotz der Einschränkung, dass die Unterbrechungen nicht in Unterfunktionen sondern nur in der Protothreadfunktion selbst statt finden können, ist es auf diese Weise möglich, auch komplexere Berechnungen, welche auf ereignisbasierten Systemen nur schwer zu realisieren sind, auf eben diesen zu implementieren.

2.8 Kommunikation

Für die Kommunikation in drahtlosen Sensornetzwerken haben sich verschiedene Technologien zur Funkübertragung etabliert. Diese sind Bluetooth, IEEE 802.15.4, sowie verschiedene, auf dem Chipcon CC1000 Transceiver aufbauende Lösungen. WLAN spielt in drahtlosen Sensornetzwerken eine geringe Rolle. Hiervon ausgenommen sind Gateways, welche das Sensornetz mit anderen, übergeordneten Netzen verbinden. Dies liegt vor allem am hohen, für den Betrieb eines WLAN benötigten, Energieverbrauchs, welcher wesentlich höher ist als die eingesetzten Verfahren [12].

Der Chipcon CC1000 wird in vielen Projekten seiner Einfachheit wegen eingesetzt. Die Funktionalität beschränkt sich darauf, serielle Daten in Funkwellen und zurück zu wandeln. Ein MAC-Protokoll¹ wird durch die Hardware nicht bereit gestellt und muss gegebenenfalls in Software implementiert werden.

Bluetooth wurde ursprünglich für die Anbindung externer Geräte wie Tastaturen, Mäuse oder Drucker, an den Computer entwickelt. Durchgesetzt hat sich Bluetooth vor allem durch das Handy. Hier wird es für die Anbindung von Headsets, Freisprecheinrichtungen oder für die Datenübertragung zu Computern oder anderen Handys verwendet.

¹ Media Access Control Protokoll: Legt fest wann und wer zu einem bestimmten Zeitpunkt Daten senden darf.

Bluetooth baut so genannte *Piconetze* auf. Hierbei kann sich ein Master mit bis zu sieben Slaves verbinden. Sollen mehr Geräte in ein Netzwerk eingebunden werden, können Scatternetze aufgebaut werden. In *Scatternetzen* werden mehrere Piconetze verknüpft. Dabei ist zu beachten, dass ein Knoten nur Master eines Netzes jedoch Slave mehrerer Netze sein kann.

Bluetooth unterstützt verschiedene Sicherheits- und Verschlüsselungsmerkmale, welche im Bluetoothchip implementiert sind. Auf Grund der starken Abstraktion innerhalb des Chips ist die Übertragung verschlüsselter Daten über Bluetooth auch von Systemen mit geringer Rechenleistung problemlos möglich. Bei einem Zugriff auf die Hardware können diese jedoch zwischen Mikrocontroller und Bluetooth Chip abgehört werden.

IEEE 802.15.4 wurde speziell für Sensor- und Aktornetze entwickelt. Es ist deutlich energiesparender als Bluetooth, hat jedoch auch eine geringere Übertragungsrate. Wie Bluetooth unterstützt auch IEEE 802.15.4 verschiedene Sicherheits- und Verschlüsselungsmerkmale. Das auf IEEE 802.15.4 aufbauende *Zigbee* definiert darüber hinaus auch Routing und Netzwerkverwaltungsfunktionen. Häufig werden hierzu jedoch eigene Implementierungen verwendet, welche für den entsprechenden Einsatzzweck geeigneter sind.

Weitere Übertragungsstandards, welche jedoch kaum verbreitet sind, sind *Z-Wave*¹ und *Wibree*, welches in Bluetooth *low energy wireless technology* umbenannt wurde².

2.9 Funktionsweise von Compilern und Linkern

Zum Verständnis von Kapitel 3 wird ein Grundverständnis von Linkern benötigt. In diesem Kapitel soll daher das Zusammenspiel vom Compiler und Linker sowie die Funktionsweise eines Linkers genauer erläutert werden.

Das Zusammenspiel von Compiler und Linker wird in Abbildung 2.3 gezeigt. Der C-Quellcode wird an den Compiler übergeben. Dieser analysiert den Code, führt verschiedene Optimierungen durch und wandelt den Quellcode in Maschinencode um. Wichtig hierbei ist, dass noch kein Speicher zugeordnet wurde. Daher müssen die Namen der Funktionen und Variablen erhalten bleiben, um später die richtigen Adressen zuordnen zu können. Entsprechend kann der Code auch keine Speicheradressen enthalten. Stattdessen wird die Adresse `0x00` als Platzhalter eingetragen und die anzuspringende Funktion oder der Name der verwendeten Variablen in einer Tabelle gespeichert. Der Name einer Funktion oder Variable wird als Symbol bezeichnet.

Funktionen und Variablen werden innerhalb der Objektdatei in verschiedene Sektionen eingeteilt. Diese sind für die Verarbeitung durch den Linker wichtig. Die Standardsektionen sind `.text`, `.data` und `.bss`. In der `.text` Sektion wird der ausführbare Programmcode gespeichert. Die `.data` Sektion enthält alle statischen und globalen Variablen, die einen Initialwert besitzen. Laut ISO-C-Spezifikation müssen alle globalen und statischen Variablen, welche nicht explizit initiali-

1 “Z-WaveAlliance.org - Alliance – Start”, <http://www.z-wavealliance.org/>.

2 “Bluetooth.com | Bluetooth Low Energy Technology”, http://www.bluetooth.com/Bluetooth/Products/low_energy.htm.

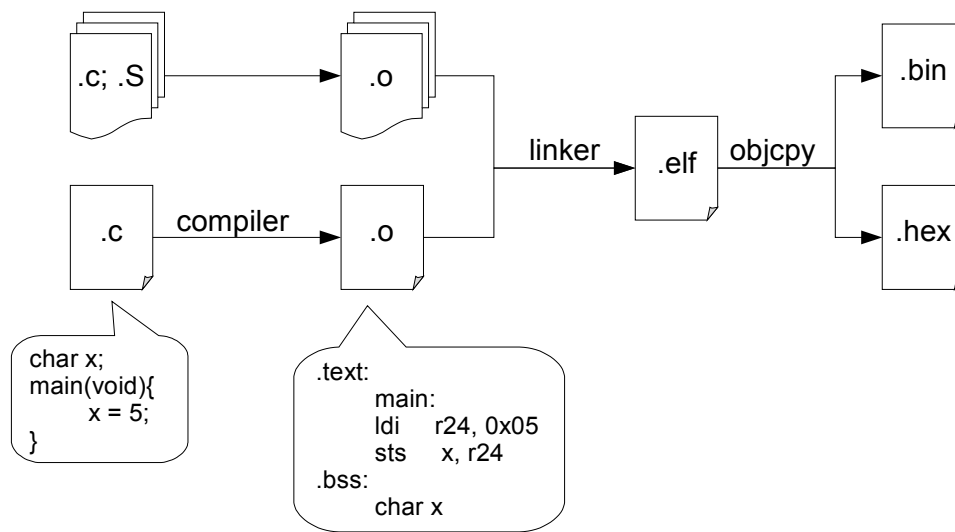


Abbildung 2.3: Zusammenspiel von Compiler und Linker

siert, mit 0 initialisiert werden [13]. Diese Variablen werden in der `.bss` Sektion abgelegt. Dies hat den Vorteil, dass nur noch der Name und die Größe der Variablen, nicht aber deren Inhalt gespeichert werden müssen.

Nach dem Compilieren werden die Objektdateien, meist zusammen mit Bibliotheken, an den Linker übergeben. Der Linker überprüft zunächst, ob er alle in den ihm übergebenen Objektdateien verwendeten Symbole auflösen kann. Ist dies nicht der Fall, werden die Bibliotheken nach Objektdateien mit dem fehlenden Symbol durchsucht und mit eingebettet. Hierdurch können weitere Abhängigkeiten entstehen, was zur Einbindung weiterer, sich in den Bibliotheken befindlicher, Objekte führen kann.

Nachdem der Linker alle benötigten Funktionen und Variablen zusammengesucht hat, werden diese im Speicher in den entsprechenden Sektionen (`.text`, `.data`, `.bss`) abgelegt. Anschließend werden die Platzhalter, welche bisher die Adresse `0x00` enthielten, durch die tatsächliche Adresse der Funktion oder Variablen ersetzt.

Bei der Erstellung eines Speicherabbildes für einen Mikrocontroller müssen hardwarebedingte Vorgaben beachtet werden. Abbildung 2.4 zeigt das Speicherlayout des BTnode Sensorknotens nach dem Linken. Am Anfang des Flash müssen sich die Interruptvektoren befinden. Dahinter befindet sich die benutzerdefinierte `.progmem` Sektion, welche in Kapitel 2.10 genauer erläutert wird. Dahinter befindet sich der Startup Code. In diesem wird der Mikrocontroller initialisiert. Der Initialisierungscode wird automatisch durch den Linker integriert. Des Weiteren zeigt der Resetvektor auf den Anfang des Startup Code. Nach der Initialisierung des Mikrocontrollers wird die Funktion `main()` aufgerufen, welche sich in der `.text` Sektion befindet.

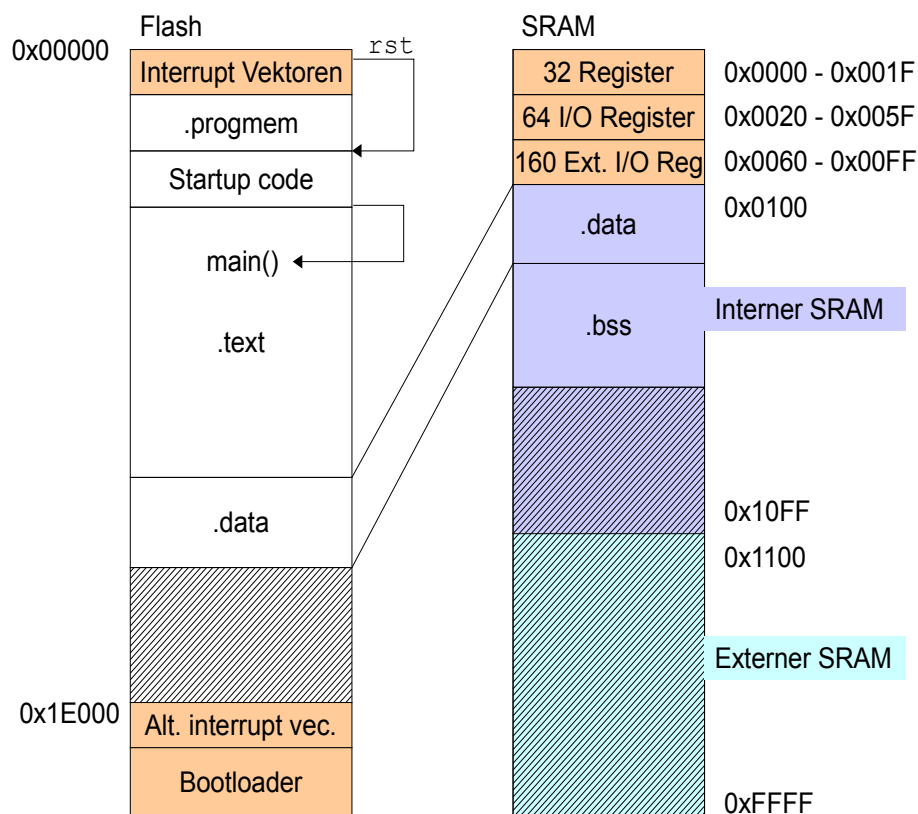


Abbildung 2.4: Aufbau des Speichers des BTnode Sensorknotens

Hinter der `.text` Sektion wird eine Kopie der `.data` Sektion abgelegt. Diese wird vom Startup Code während der Initialisierung in den SRAM kopiert. Auch die sich hinter der `.text` Sektion befindliche `.bss` Sektion wird von dem Startup Code mit `0x00` initialisiert.

Im Datenspeicher werden im Adressbereich von `0x000` bis `0x100` die verschiedenen Register des ATmega128 abgebildet. Hierdurch ist es möglich auf diese ohne weitere Spezialbefehle zuzugreifen.

Eine Besonderheit des ATmega128 ist der Bootloaderbereich, welcher sich am Ende des Programmspeichers befindet. Dieser kann 1, 2, 4 oder 8 KiB groß sein. In dem Beispiel ist er 8 KiB groß und beginnt daher bei der Adresse `0x1E000`. Soll der Flash durch den Mikrocontroller und nicht durch einen ISP reprogrammiert werden, muss sich der Reprogrammierungsbefehl im Bootloaderbereich befinden.

Am Anfang des Bootloaderbereichs befinden sich auch die alternativen Interruptvektoren. Es ist möglich den Resetvektor in den Bootloaderbereich zu legen, so dass nach einem Reset zunächst ein Bootloader ausgeführt werden kann, welcher gegebenenfalls ein neues Speicherabbild in den Flash schreiben kann. Alternativ können alle Interrupts in den Bootloaderbereich verschoben werden, um auf diese Weise auch während auf den Flash geschriebene Interrupts im Bootloaderbereich verarbeiten zu können (Siehe auch Kap. 5.3.1).

2.10 Benutzerdefinierte Sektionen

Häufig macht es Sinn große und selten verwendeten Daten, wie Zeichenketten, im Programmspeicher abzulegen und erst bei Bedarf in den Datenspeicher zu laden. Der GCC-Compiler kann durch Attribute angewiesen werden, Variablen oder Funktionen in spezielle Sektionen zu schreiben. Die AVR-Libc stellt beispielsweise das Makro `PSTR()` zur Verfügung. Der Befehl

```
PSTR("HELLO WORLD!")
```

legt die Zeichenkette „HELLO WORLD“ nicht in der `.data` sondern der `.text` Sektion ab. Vor allem bei Harvardarchitekturen muss hier aber darauf geachtet werden, dass der Zeiger auf diese Variable die Adresse im `.text` und nicht in der `.data` Sektion enthält und daher nicht direkt auf die Variable zugegriffen werden kann. Um im Programmspeicher liegende Zeichenketten zu verarbeiten gibt es spezielle Zeichenkettenfunktionen, wie `strcpy_P()`.

Des Weiteren ist es möglich benutzerdefinierte Sektionen anzulegen. Die Anweisung

```
void boot(void) __attribute__((section(".bootloader")));
```

legt die Funktion `boot()` in die Sektion `bootloader`. Der Parameter

```
--section-start=.bootloader=0x1E000
```

weist den Linker an, die Sektion `.bootloader` an der Adresse `0x1E000` beginnen zu lassen. Entsprechend wird die Funktion `boot()` an die Adresse `0x1E000` geschrieben.

Bei der Verwendung von benutzerdefinierten Sektionen muss aber beachtet werden, dass sich dies nur auf die Funktion mit dem entsprechenden Attribut bezieht. Andere Funktionen, auch solche, die durch `boot()` aufgerufen werden, werden auch weiterhin in die normale `.text` Sektion gelegt. Dies hat zur Folge, dass von Bootloader benötigte Funktionen nach dem Schreiben des Flash nicht mehr zur Verfügung stehen.

Das Schreiben aller vom Bootloader benötigter Funktionen in die `.bootloader` Sektion ist jedoch nicht immer hilfreich, weil es dem Linker offen steht Funktionen innerhalb eine Sektion beliebig zu sortieren. So kann es passieren, dass nicht die Funktion `boot()` sondern die Funktion `memcpy()` nach einem Reset ausgeführt wird.

Flexibler können Sektionen mit Hilfe eines benutzerdefinierten Linkerskripts definiert werden. Der Folgende Abschnitt zeigt einen vereinfachten Ausschnitt aus dem Standardlinkerskript des ATmega128:

```
.text :
{
  *(.vectors)
  /* For data that needs to reside in the lower 64k of progmem. */
  *(.progmem.gcc*)
  *(.progmem*)
  . = ALIGN(2);
[...]
  *(.init0) /* Start here after reset. */
  *(.init1)
```

```
*(.init2) /* Clear __zero_reg__, set up stack pointer. */
*(.init3)
[...]
*(.text)
. = ALIGN(2);
*(.text.*)
. = ALIGN(2);
[...]
_etext = . ;
} > text
```

Der obige Ausschnitt definiert, welche Sektionen in die endgültige `.text` Sektion kopiert werden sollen. Zunächst werden die Interruptvektoren im Programmspeicher platziert. Dies ist durch die Hardware vorgegeben. Dahinter werden alle Daten gespeichert, welche im Sektionsnamen `.progmem` enthalten, wobei `.progmem.gcc` bevorzugt wird. Diese Sektion wird genutzt um Variablen, welche im Programmspeicher abgelegt werden, möglichst an den Anfang des Adressraums abzulegen, da auf Grund der aktuellen Implementierung der AVR-Libc Probleme beim Zugriff oberhalb von 64 KiB entstehen können [14]. Anschließend werden verschiedene Initialisierungsfunktionen und dahinter die normalen Funktionen platziert. Der Befehl `„. = ALIGN(2);“` richtet die aktuelle Adresse, welche durch den Punkt repräsentiert wird, an einer geradzahligen Adresse aus. Dies ist nötig, da der Programmspeicher wortweise adressiert wird.

Am Ende wird die aktuelle Adresse dem Symbol `_etext` zugewiesen. Mit dem Befehl

```
printf("ET: %p\n", &_etext);
```

ist es zum Beispiel möglich sich die Endadresse der `.text` Sektion ausgeben zu lassen.

3 Nachladen und Aktualisieren von Code zur Laufzeit

Es gibt verschiedene Lösungsansätze, um Code auf Sensorknoten zur Laufzeit nachzuladen oder zu aktualisieren. Diese Kapitel soll einen Überblick über die gängigen Lösungsansätze geben. Hierbei werden zunächst Skriptsprachen und *virtuelle Maschinen* vorgestellt, bei welchen der Code nicht durch den Mikrocontroller, sondern durch einen in Software geschriebenen Interpreter verarbeitet wird.

Nachdem der Austausch des gesamten Binärabbildes erläutert wurde, wird auf die Möglichkeit der Codeaktualisierung zur Laufzeit eingegangen. Das Themengebiet wird mit einem Unterkapitel über Module abgeschlossen. In diesem wird erklärt, wie diese eingebunden werden können und welche Möglichkeiten es gibt, von einem Modul aus auf Funktionen des Kernels zuzugreifen.

3.1 Skriptsprachen

Skriptsprachen sind Programmiersprachen, welche zur Laufzeit durch einen Interpreter interpretiert werden. Im Falle des Sensorknoten bedeutet dies, dass der Quellcode auf den Knoten übertragen und dort ausgeführt wird, ohne dass er vorher in Binärcode umgewandelt wurde. Der Quellcode wird im Arbeitsspeicher gespeichert und kann so leicht ausgetauscht oder aktualisiert werden.

Die Verwendung von Skriptsprachen hat verschiedene Vorteile. Zum Beispiel ist es möglich den gleichen Code auf unterschiedlichster Hardware auszuführen. Kann ein Befehl, zum Beispiel weil keine Multiplikationseinheit zur Verfügung steht, nicht direkt von der Hardware ausgeführt werden, so kann dieser Befehl auch durch Software interpretiert werden.

Auf der Harvardarchitektur haben Skriptsprachen den Vorteil, dass der auszuführende Code im Datenspeicher liegt. Es kann also mit geringem Aufwand ausgetauscht, angepasst oder sogar generiert werden.

Skriptsprachen benötigen viele Systemressourcen, da der vollständige Programmcode auf dem Knoten gespeichert und analysiert werden muss bevor er ausgeführt werden kann. Auch müssen sehr viele Zustandsinformationen zwischengespeichert werden. Auf Grund der hohen Ressourcenanforderungen sind Skriptsprachen, außer in seltenen Ausnahmefällen, für die Verwendung auf Sensorknoten nicht geeignet.

3.2 Virtuelle Maschinen

Virtuelle Maschinen (VM) ähneln den Skriptsprachen. Im Gegensatz zu Skriptsprachen wird der Quelltext zunächst von einem Compiler in Bytecode umgewandelt. Der Bytecode wird jedoch nicht vom Mikrocontroller selbst, sondern durch die auf dem Knoten laufende VM interpretiert. Bei der VM handelt es sich um Software, welche die Befehle ausliest und interpretiert. Hierbei abstrahiert die VM sowohl den Prozessor als auch angeschlossene Hardware oder logische Schichten auf möglichst hoher Ebene und stellt hierfür Schnittstellen zur Verfügung.

Der auf dem Host-Rechner ausgeführte Compiler wandelt den Quellcode nicht nur in Bytecode um, sondern führt auch Optimierungen durch. Hierdurch entsteht ein, im Vergleich zum Quellcode, kompakter und optimierter Code. Dies, und die durch die starke Abstraktion kleine Codegröße sind von großem Vorteil, wenn der Code sehr oft geändert oder ausgetauscht wird.

Obwohl sie deutlich effektiver sind als Skriptsprachen, sind VMs deutlich ineffektiver als die Ausführung von Binärcode auf dem Controller. Die Ineffektivität entsteht vor allem dadurch, dass jeder Befehl erst geladen werden muss, um dann zu der entsprechen Stelle des VM-Codes zu springen, welcher für die Interpretation des entsprechenden Befehls zuständig ist. Ein zusätzlicher Bedarf an Arbeitsspeicher wird nicht nur durch den Programmcode, sondern auch durch verschiedene Statusinformationen, welche im Arbeitsspeicher abgelegt werden, verursacht.

Dunkels et al. haben gezeigt, dass eine VM bei häufigen Veränderungen des Codes durchaus effizienter sein kann als die anderen hier vorgestellten Lösungen [4].

Eine virtuelle Maschine für TinyOS, welche häufig als Referenz für diese Klasse der Sensorknoten verwendet wird, ist Maté [15]. Weitere VMs für Mikrocontroller sind DAViM [16] und SwissQM [17].

3.3 Aktualisieren des Binärcodes

Das einfachste Konzept der Codeaktualisierung besteht darin den gesamten Binärcode auszutauschen. Hierzu kann entweder ein ISP oder, wenn sich der Controller selbst programmieren kann, ein Bootloader verwendet werden. Letzterer wird nach einem Reset ausgeführt und aktualisiert den Programmspeicher. Das Binärabbild wird meist über die serielle Schnittstelle zum Knoten übertragen. Alternativ wird das Binärabbild zuvor in einem freien Bereich des internen oder eines externen Speicher gespeichert und von dort vom Bootloader an seine endgültige Position kopiert.

Diese Methode ist einfach, jedoch hat sie den Nachteil, dass sie nicht besonders effektiv ist. Werden die Daten über Kabel, zum Beispiel eine serielle Schnittstelle, übertragen, spielt dies kaum eine Rolle. Bei der Funkübertragung in Sensornetzwerken stellt Energieverbrauch jedoch einen entscheidenden Faktor dar.

Da immer nur ein Teil des Quellcodes verändert wird, weist ein neu erstelltes Speicherabbild im Normalfall Ähnlichkeiten mit der vorherigen Version auf. Es ist also auch möglich nur die Änderungen zwischen dem neu erstellten und dem sich bereits auf dem Sensorknoten befindlichen Binärcode zu übertragen. Dieser Ansatz wird auch als diff-basierter Ansatz bezeichnet [4].

Obwohl weniger Daten übertragen werden, bedeutet dies nicht unbedingt, dass dies effizienter ist. In [4] wird sowohl die Ausführungszeit, als auch der Energieverbrauch verglichen, welcher benötigt wird die Differenz, beziehungsweise das gesamte Speicherabbild, auf den Knoten zu übertragen. Obwohl der diff-basierte Ansatz bei der Übertragung dem vollständigen Speicherabbild überlegen ist, verbraucht er unter bestimmten Szenarien mehr Energie. Die Ursache für den hohen Energieverbrauch ist, dass der Speicher zum Teil mehrmals umkopiert werden muss. Da der SRAM meist nicht ausreicht, muss auf Speichermedien wie EEPROM oder Flash zurückgegriffen

werden, welche beim Speichern deutlich mehr Energie benötigen. Ob der diff-basierte Ansatz effizienter als die Übertragung des gesamten Binärbildes ist, hängt daher stark von der Übertragungsrate und Energieeffizienz des Netzwerkes sowie des verwendeten Speichers ab. In [6] wird eine Lösung vorgestellt, wie Binärcode erstellt werden kann, welcher möglichst effizient mit dem diff-basierten Ansatz übertragen werden kann.

3.4 Aktualisierung von Binärcode zur Laufzeit

Wird der Binärcode ausgetauscht, muss sichergestellt werden, dass der zu aktualisierende Code während der Aktualisierung nicht ausgeführt wird. Hiervon sind auch alle auf dem Stack liegende Funktionen betroffen, da die entsprechende Rücksprungadresse bei einer Codeaktualisierung ungültig wird. Aus diesem Grund wird meist die vorgestellte Lösung der Codeaktualisierung mit einem Bootloader verwendet.

Dass es möglich ist, einen Mikrocontroller zur Laufzeit zu aktualisieren, wird von Felser und anderen in [18] allgemein und von Oechslein in [19] für den BTnode gezeigt. Hierzu werden die aktiven Funktionen anhand einer Stackanalyse bestimmt und mit den zu ersetzenden Funktionen verglichen. Anschließend wird analysiert, ob eine Aktualisierung der Funktionen möglich ist. Ist eine Aktualisierung an der aktuellen Stelle nicht möglich, werden verschiedene Techniken eingesetzt, um das Programm an einer geeigneten Stelle zu unterbrechen.

Neben genauen Informationen über das laufende System müssen auch einige Informationen über den Systemzustand an den Host-Rechner zur Analyse übertragen werden. Bei diesem Ansatz bestehen zusätzlich verschiedene Beschränkungen bei umfangreichen Änderungen, zum Beispiel bei der Änderung von Datentypen.

Ein weitergehender Ansatz von Neamtii und anderen ermöglicht auch umfangreichere Codeaktualisierung zur Laufzeit [20]. Auf Grund der Anforderungen bezüglich der Indirektion von Variablen, Tests, und Speicher, der für eventuelle Erweiterungen frei gehalten wird, eignet sich dieses Konzept eher für leistungsfähige Systeme, denen ausreichend Systemressourcen zur Verfügung stehen.

3.5 Verwenden von Modulen

Der Code, welcher auf Sensornetzwerkknoten ausgeführt wird, besteht meist aus einem Betriebssystemkern (Kernel), welcher nur selten verändert wird, und einer oder mehreren Applikationen, welche auf dem Knoten ausgeführt werden. Bestehen keine Abhängigkeiten des Kerns zu der Applikation, kann diese in ein eigenes Modul gespeichert werden. So muss vor der Aktualisierung lediglich die in dem Modul enthaltene Applikation beendet und anschließend wieder gestartet werden.

Als Modul wird in diesem Kontext ein Block Binärcode bezeichnet, welcher auf den Knoten geladen und zur Ausführung gebracht werden kann. Das Laden von Modulen muss durch die Betriebssystem unterstützt werden. Dies liegt vor allem daran, dass zum Zeitpunkt der Kernelerstellung des Betriebssystems viele Informationen über das Modul nicht bekannt sind. Hierzu zählen die von

dem Modul angebotenen Funktionen, deren Adressen, sowie der benötigte Speicherbedarf. Einige Betriebssysteme reservieren den zum Betrieb benötigten Speicher bereits bei der Erstellung. Bei diesen ist eine Modulunterstützung nur schwer möglich.

Um auf die in den Modulen enthaltenen Funktionen zugreifen zu können, muss der Kernel Informationen über den Inhalt und Einsprungpunkte des Moduls haben. Dies kann auf verschiedene Weise geschehen. Zum Beispiel kann jedes Modul einen einleitenden Block enthalten, welcher diese Informationen enthält. Alternativ kann das Modul auch eine Funktion enthalten, welche das Modul registriert. Diese wird nach dem Laden des Moduls vom Kernel aufgerufen.

Es muss meist nicht nur vom Kernel aus auf die Module zugegriffen werden, sondern auch von den Modulen auf Funktionen des Kernels oder Funktionen anderer Module. Nachfolgend werden einige Konzepte vorgestellt, welche es Modulen ermöglichen auf Funktionen des Kernels zuzugreifen.

3.5.1 Pre-Linking von Modulen

Soll ein Modul gelinkt werden, ist es möglich die im Kernel abgelegten Funktionen aus dem Modul heraus aufzurufen. Die Funktionen müssen daher nicht nochmals in das Modul eingebettet werden. Hierzu muss lediglich die Speicheradresse der aufzurufenden Funktion bekannt sein. Geschieht dies auf dem Host und nicht auf dem Knoten selbst, wird dies in [4] als *Pre-Linking* bezeichnet. Die Tupel aus Funktionsname und Adresse können als Symboltabelle an den Linker übergeben werden, dieser trägt dann die entsprechende Adresse in das Modul ein. Dies funktioniert selbstverständlich auch für Variablen.

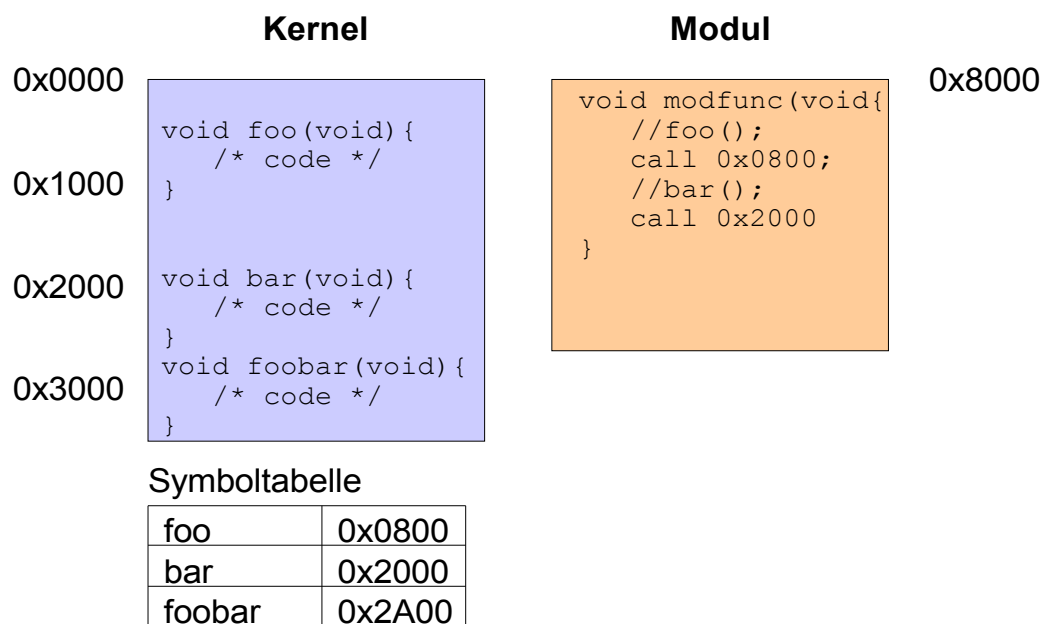


Abbildung 3.1: Verwendung einer Symboltabelle, um aus einem Modul auf Funktionen des Kernels zuzugreifen

Abbildung 3.1 zeigt die Verwendung der Symboltabelle des Kernels für ein Modul, welches die durch den Kernel zur Verfügung gestellten Funktionen `foo()`, `bar()` und `foobar()` verwendet. Der Linker bekommt die Symboltabelle des Kernels übergeben. Statt die Funktionen, welche bereits im Kernel enthalten ist, nochmal in das Modul einzubinden, trägt der Linker die ihm übergebenen Adressen ein. Bei der Ausführung des im Modul enthaltenen Codes wird die entsprechende Funktion im Kernel aufgerufen.

Soll das Modul auf einem anderen Computer erstellt werden, muss auch hier die Symboltabelle bekannt sein. Wird eine Quellcodeverwaltung verwendet, reicht gegebenenfalls die Versionsnummer des zur Erstellung des Kernels verwendeten Quellcodes aus. Anhand des Quellcode kann der Kernel nochmals erstellt werden und die Symboltabelle extrahiert werden.

In [4] wird jedoch auf die *Micro Heterogeneity* hingewiesen. Diese entsteht durch die Verwendung unterschiedlicher Versionen des Compilers oder Linkers. Obwohl der selbe Quellcode verwendet wird, unterscheidet sich der resultierende Code minimal. Ursache hierfür sind zum Beispiel Änderungen in der Optimierung oder eine andere Anordnung der einzelnen Funktionen durch den Linker. Infolgedessen stimmen die Adressen der Funktionen und Variablen nicht überein. Wird die falsche Adresse aufgerufen, ist das Verhalten des Knotens nicht mehr vorhersagbar.

3.5.2 Lookup Tables

Ein von SOS¹ verwendeter Ansatz ist eine *Lookup Table*. Während der Initialisierung eines Moduls kann dieses verschiedene Funktionen in der Tabelle registrieren. Soll eine Funktion aufgerufen werden, wird die entsprechend Speicheradresse über einen Zeichenkettenvergleich aufgelöst und anschließend aufgerufen.

Der Vorteil dieser Methode ist die hohe Flexibilität. Um den Overhead beim Aufruf zu minimieren, kann die Sprungadresse, nachdem sie aufgelöst wurde, zwischengespeichert werden. Neben dem Rechenaufwand welcher durch die zusätzliche Indirektion entsteht, hat die Verwendung von *Lookup Tables* einen entscheidenden Nachteil: Entweder muss der vollständige Funktionsname im häufig nur 4 KiB großen Arbeitsspeicher abgelegt werden, oder es muss mit IDs gearbeitet werden. Die Verwendung von IDs bedeutet einen zusätzlichen Verwaltungsaufwand, da sichergestellt werden muss, dass jede Funktion eine eindeutige ID bekommt. Diese Funktions ID darf zu einem späteren Zeitpunkt nicht mehr geändert werden, da die Funktion sonst entweder nicht zu Verfügung steht oder eine andere Funktion aufgerufen wird. Letzteres hat meist unvorhersehbare Folgen, da die Übergabeparameter für eine andere Funktion bereitgestellt wurden.

3.5.3 Jump Tables

Eine Methode, welche effektiver, jedoch weniger flexibel als die *Lookup Table* ist, ist die Verwendung von *Jump Tables*, beziehungsweise Sprungtabellen. Hierbei handelt es sich um eine Liste von Sprungbefehlen im Binärcode, welche jeweils zu einer Funktion springen. Soll eine Funktion auf-

¹ “SOS 2.x Home Page”, <https://projects.nesl.ucla.edu/public/sos-2x/doc/>.

gerufen werden, wird anstatt der Adresse der Funktion die Adresse des auf die Funktion springenden Sprungeintrags aufgerufen. Hierzu muss lediglich die Position der Tabelle und der Index der Funktion innerhalb der Tabelle bekannt sein, nicht aber die tatsächliche Adresse der Funktion. Der Sprungeintrags springt nach seiner Ausführung direkt weiter zu der tatsächlichen Adresse der auszuführenden Funktion. Da sich ein Sprung lediglich auf den Programmzähler, nicht aber auf die Rücksprungadresse oder die übergebenen Funktionsparameter auswirkt, ist die Verwendung von Sprungtabellen transparent.

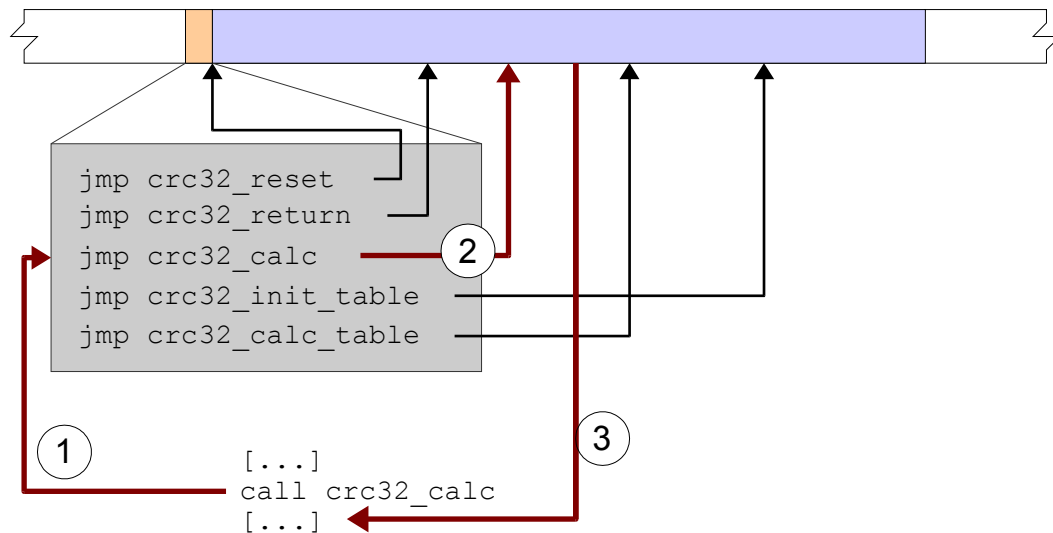


Abbildung 3.2: Funktionsweise von Jump Tables

Abbildung 3.2 zeigt, wie eine solche Sprungtabelle in einem *CRC32*-Modul aussehen kann. Am Anfang des Moduls befinden sich die Sprungbefehle zu der tatsächlichen Adresse der Funktion. Die Position der Funktion innerhalb der Sprungtabelle muss bekannt sein. Soll beispielsweise die Funktion `crc32_calc()` ausgeführt werden, wird die Adresse des Befehls `jmp crc32_calc` angesprungen (1). Dieser ändert den Programmzähler auf die tatsächliche Adresse der Funktion (2). Zurückgesprungen wird an die Adresse hinter dem ursprünglichen Funktionsaufruf (3).

3.5.4 Linken von Modulen auf dem Knoten

Module können nicht nur auf dem Host, sondern auch auf dem Knoten selbst gelinkt werden. Bei allen verbreiteten Mikrocontroller Betriebssystemen wurde, mit Ausnahme von Contiki, auf einen Linker verzichtet. Einer der Gründe hierfür ist, dass der Linker, vor allem aber die Symboltabelle, zusätzlichen Speicher benötigen.

In [4] und [5] wurde gezeigt, dass die Verwendung eines Linkers auf dem Knoten, vor allem in Hinblick auf den Energieverbrauch, effizienter ist als die Verwendung des diff-basierten Ansatzes. Dieser Vorteil verstärkt sich in großen Netzwerken, in denen unterschiedliche Kernelversionen vorkommen, und die Aktualisierungen nicht von einem zentralen Server direkt an den Knoten ver-

teilt, sondern von einem Knoten zum nächsten weitergegeben werden. Dieses „Fluten“ des Netzwerkes erlaubt vor allem in größeren Sensornetzwerken eine sehr schnelle und effiziente Verbreitung eines Moduls.

Einer der Hauptgründe, die Dunkels et al. für dieses Konzept aufführen, ist die so genannte *Micro Heterogeneity*, welche bereits in Kapitel 3.5.1 erwähnt wurde [4].

Im Gegensatz zum *Pre-Linking* von Modulen wird jedoch mehr Speicherplatz benötigt, da sowohl der Linker als auch die Symboltabelle des Kernels auf dem Knoten gespeichert werden müssen. Der dynamische Linker vergrößert Contiki um ca. 5,7 KiB, wobei die Symbol Tabelle ca. 4 KiB groß ist.

Nicht nur der Kernel ist deutlich größer, sondern es müssen auch mehr Daten übertragen werden, da in den Modulen nicht die zwei Byte große Speicheradresse, sondern der Name des Symbols gespeichert werden muss. Je nach Anzahl der verwendeten Symbole und der Länge der Symbolnamen kann dies einen nicht unerheblichen zusätzlichen Aufwand bedeuten.

3.5.5 PIC - Position Independent Code

Eine große Einschränkung beim *Pre-Linking* von Modulen ist, dass ihre Position im Speicher zum Zeitpunkt des Linkes feststehen muss. Soll das gleiche Modul auf mehreren Knoten installiert werden, muss sichergestellt werden, dass der Speicherbereich auf allen Knoten unbenutzt ist. Eine Möglichkeit, um zumindest das Text-Segment beim *Pre-Linking* beliebig zu platzieren, ist die Verwendung von Position Independent Code (PIC). Dieser verwendet innerhalb des Moduls nur relative Sprünge. Hierdurch kann er an einer beliebigen Position platziert werden.

Da der `rjmp`-Befehl, welcher relative Sprünge ausführt, auf dem ATmega128 auf 12 Bit begrenzt ist, ist es maximal möglich ± 2048 Wörter zu springen [21]. Hierdurch wird auch die Größe von PIC-Modulen auf 4 KiB begrenzt. Es wäre zwar möglich mehrfach zu springen und durch geschickte Anordnung der Funktionen die Anzahl der doppelten Sprünge auf ein Minimum zu reduzieren, jedoch gibt es hierfür bisher keine Werkzeuge.

4 Vergleich bekannter Betriebssysteme für Mikrocontroller

In diesem Kapitel werden verschiedene Betriebssysteme vorgestellt, welche im Bereich der Sensornetze eingesetzt werden. Anschließend wird diskutiert, ob sie sich für die Erfüllung der Aufgabenstellung dieser Arbeit eignen und wie groß der jeweilige Implementierungsaufwand ist.

4.1 TinyOS

TinyOS¹ ist ein ereignisbasiertes Betriebssystem, welches speziell für Sensornetze entwickelt wurde. Es ist im Bereich der Sensornetzwerkforschung das wohl am weitesten verbreitete Betriebssystem und wird häufig als Referenzsystem aufgeführt. Der Schwerpunkt von TinyOS liegt in möglichst kompaktem und robustem Code. Aus diesem Grund verzichtet TinyOS weitgehend auf Threads und dynamische Speicherverwaltung, auch wenn beides grundsätzlich möglich ist. Das Nachladen von Modulen ist auf Grund der verwendeten Strukturen nur schwer möglich.

TinyOS sowie die zugehörigen Applikationen werden in NesC programmiert. Hierbei handelt es sich um C-Erweiterungen, welche Portabilität, Flexibilität und die Robustheit erhöht. Der NesC-Code wird von einem speziellen Compiler zunächst in C-Code umgewandelt und anschließend kompiliert. NesC hat unter anderem ein Schnittstellen-Konzept. Jede C-Datei beinhaltet eine Liste der zur Verfügung gestellten sowie der benötigten Schnittstellen. Auf diese Weise ist es möglich für eine Schnittstelle verschiedene Implementierungen zu realisieren. Je nach zur Verfügung stehender Hardware kann auf diese Weise die passende Schnittstellenimplementierung gewählt werden.

Nachdem der NesC Compiler den benötigten Quellcode zusammengestellt hat, wandelt er ihn in normalen C-Code um, welcher von einem normalen C-Compiler kompiliert werden kann. Im Normalfall ist der endgültig Funktionsumfang erst nach der Ausführung des Linkers bekannt, da dieser die benötigten Funktionen aus den Objektdateien zusammen sucht. Da diese Aufgabe durch den NesC Compiler übernommen wird, ist der endgültige Funktionsumfang dem Compiler bereits bekannt. Dies wird dazu genutzt den Code zu analysieren und verschiedene Probleme wie mögliche Deadlocks automatisch zu erkennen.

Zum Aktualisieren von TinyOS steht Deluge[7] oder FlexCup[5] zur Verfügung, welches den diff-basierten Ansatz zur Aktualisierung verwendet.

Die aktuellen Entwicklungen bei TinyOS haben ihren Schwerpunkt vor allem in einer weiteren Stabilisierung und Vorhersagbarkeit des Laufzeitsystems.

¹ „TinyOS Community Forum || An open-source OS for the networked sensor regime.“, <http://www.tinyos.net/>

4.2 Contiki

Contiki¹ wurde größtenteils von Adam Dunkels am Swedish Institute of Computer Science entwickelt. Wie TinyOS arbeitet auch Contiki ereignisbasiert. Durch die Verwendung von *Protothreads* (siehe Kap. 2.7.3) werden Thread-ähnliche Strukturen zur Verfügung gestellt. Auf den *Protothreads* aufbauend wurde eine Bibliothek entwickelt, welche es ermöglicht auf dem ereignisbasierten System Threads auszuführen. Hierzu wird Speicher für den Stack reserviert. Gibt der Thread die CPU ab, so wird auf den System-Stack zurück gewechselt. Der Protothread wartet dann darauf, dass er durch ein Ereignis fortgesetzt wird und wechselt zurück auf den Stack des Threads. Durch die Verwendung von Timer-Interrupts ist auch eine Verdrängung (Preemption) des laufenden Threads möglich. Dies ermöglicht die Ausführung komplexer Berechnungen, ohne dass während der Entwicklung geeignete Punkte für die CPU-Abgabe bestimmt werden müssen.

Contiki bietet die Möglichkeit Module zur Laufzeit nachzuladen. Diese werden gegen den Kernel gelinkt. Mit Hilfe von Reallokationsinformationen ist es dem auf dem Knoten befindlichen Loader möglich das Modul an eine beliebige Stelle des Speichers zu positionieren. Nach dem Ablegen des Moduls im Programmspeicher wird eine Initialisierungsfunktion ausgeführt. Diese kann im Kernel Dienste (Service) und Prozesse registrieren. Dienste sind mit Shared Libraries zu vergleichen. Es handelt sich um Funktionen, welche durch andere Module aufgerufen werden können. Der Aufruf der Dienst-Funktion geschieht durch eine Zeichenkettenvergleich. Um weitere Aufrufe zu beschleunigen, wird eine Identifikationsnummer der Funktion gespeichert. Obwohl die Modulunterstützung schon für verschiedene Plattformen implementiert wurde, ist dies für die AVR-Plattform und somit für den ATmega128 noch nicht geschehen.

Durch *Micro Heterogeneity* kam es bei den Entwicklern von Contiki immer wieder zu Problemen bei der Erstellung von Modulen (vgl. Kap. 3.5.1). Aus diesem Grund entwickelten sie einen Linker, welcher auf dem Knoten läuft [4]. Dieser Linker unterstützt auch die AVR-Plattform, jedoch wird bis jetzt nur ein einziges Modul unterstützt, da es noch nicht möglich ist mehrere Module im Programmspeicher zu verwalten. Dennoch wurde damit gezeigt, dass es möglich ist einen Linker auf einem Sensorknoten laufen zu lassen.

Der Schwerpunkt liegt bei Contiki in der Erforschung der Kommunikation in Sensornetzwerken. Infolgedessen wurden unter anderem der sehr kleine TCP/IP Stack uIP [22], die Netzwerkabstraktion Rime [23] und COOJA, ein Simulator für Contiki [24] implementiert. Mit dem Simulator ist es möglich mehrere Knoten gleichzeitig zu simulieren, um so deren Kommunikationsverhalten zu analysieren.

Um die Leistungsfähigkeit von Contiki zu beweisen, wurden unter anderem ein Webserver, ein VNC-Server und ein Webbrowser implementiert, welcher auf, mit dem BNode vergleichbaren, Knoten läuft. Des Weiteren wurde Contiki als sehr portables System entwickelt und auf viele andere Systeme, unter anderem Apple II, portiert.

¹ „The Contiki Operating System – Home“, <http://www.sics.se/contiki/>

4.3 SOS

SOS¹ ist ein weiteres ereignisbasiertes Betriebssystem. Im Gegensatz zu TinyOS ist auch SOS in der Lage Module nachzuladen. Der Forschungs- und Entwicklungsschwerpunkt liegt im Vergleich zu Contiki weniger in der Netzwerkfähigkeit sondern darin ein möglichst robustes Betriebssystem zu entwickeln.

Um Module nachzuladen werden diese gegen eine *Jump Table* des Kernel gelinkt. Dies ermöglicht es Module vom Kernel unabhängig zu kompilieren, jedoch muss die Anzahl der vom Kernel zur Verfügung gestellten Funktionen stark limitiert werden (vgl. Kap. 3.5.3). Innerhalb des Moduls werden nur relative Sprünge verwendet. Dies ermöglicht es, das Modul an eine beliebige Stelle im Speicher zu positionieren, ohne dass ein Linker oder Loader benötigt wird. Auf der AVR-Plattform bedeutet dies jedoch eine Limitierung auf 4 KiB, da relative Sprünge bis maximal ± 4 KiB unterstützt werden.

Die einzelnen Module können sowohl über Nachrichten miteinander kommunizieren, als auch über das Registrieren von Funktionen. Ist eine Funktion im System registriert, so können andere Module direkt auf diese Funktion zugreifen, ohne dass sie eine Nachricht verschicken müssen. Hierzu erfragen sie die Adresse der Funktion anhand eines Tupels aus Modul-ID und Funktions-ID. Nach der ersten Abfrage wird ein Zeiger auf den Zeiger, welcher auf die Funktion zeigt, gespeichert. Dieser Dereferenzierungsschritt ermöglicht, dass nach dem Austausch eines Moduls nur der globale Zeiger auf eine Funktion aktualisiert werden muss, nicht jedoch die Funktionszeiger innerhalb anderer Module.

In SOS wurden verschiedene Mechanismen eingebaut, um einen stabilen Betrieb zu gewährleisten. Um die Verwendung von `malloc()` sicherer zu gestalten, muss allozierter Speicher immer einem Modul zugeordnet werden. Wird das Modul beendet, gibt ein Garbage Collector den, vom Modul nicht freigegebenen, Speicher wieder frei. Des Weiteren wird regelmäßig anhand von Guard-Bytes, welche sich nicht verändern dürfen, überprüft, ob ein Modul über den ihm zugewiesenen Speicher hinausgeschrieben hat. Da der Speicher einem Modul zugeordnet ist, kann das fehlerhafte Modul erkannt werden. Nach einem Neustart wird das Modul nicht mehr gestartet und so ein stabiler Betrieb gewährleistet.

Protothreads (siehe Kap. 2.7.3) wurden auch in SOS implementiert, jedoch wird die Implementierung aktuell nicht gewartet.

4.4 Nut/OS und BTnut

Nut/OS² ist ein non-preemptive multithreading Betriebssystem, welches hauptsächlich für die Ethernut Knoten entwickelt wurde. Der Schwerpunkt des Systems liegt darin, dem Entwickler eine ähnliche Umgebung zu bieten, wie er sie auch von größeren Betriebssystemen kennt. Neben der

1 „SOS 2.x Home Page“, <https://projects.nesl.ucla.edu/public/sos-2x/doc/>

2 „Ethernut Software“, <http://www.ethernut.de/en/software/index.html>

Möglichkeit mehrere Threads laufen lassen, werden von Nut/OS Events, Timer, Message Queues und Semaphoren zur Verfügung gestellt. Threads können auf Grund der Verwendung der dynamischen Speicherverwaltung auch mehrfach instantiiert werden.

Ein weiterer Schwerpunkt liegt in der Netzwerkfähigkeit und der Portabilität. Es wurden Treiber für verschiedene Netzwerkcontroller sowie verschiedene Protokolle wie zum Beispiel TCP/IP, DHCP, HTTP oder FTP implementiert. Des Weiteren wurde Nut/OS auf die AVR, ARM7, ARM9, H8/300H und m68k Plattformen portiert.

Bei BTnut¹ handelt es sich um eine für den BTnode Sensorknoten angepasste Version von Nut/OS. Neben kleinen Anpassungen und Erweiterungen wurde Nut/OS um einen Bluetooth-Stack erweitert. Dieser ermöglicht die einfache Verwendung des auf dem BTnode befindlichen Bluetooth-Controllers.

4.5 Wahl des geeigneten Systems

Die Hauptanforderungen für diese Arbeit waren die Möglichkeit Software zur Laufzeit nachladen zu können, die Unterstützung von Threads und die Beibehaltung von Zustandsdaten nach einer Softwareaktualisierung. In Tabelle 4.1 werden die untersuchten Betriebssysteme anhand der Anforderungen verglichen:

Anforderung	TinyOS	Contiki	SOS	BTnut
Modulunterstützung	Nein	Ja, muss angepasst werden	Ja	Nein
Threadunterstützung	Ja ²	Ja, mit Bibliothek	Nein	Ja
Wiederherstellung von Zustandsvariablen	Nein	Nein	Nein	Nein

Tabelle 4.1: Erfüllung der Anforderungen durch die untersuchten Betriebssysteme

Die Wiederherstellung von Zustandsvariablen wird von keinem der Systeme unterstützt. Da sich der Implementierungsaufwand zwischen den verschiedenen Systemen nicht signifikant unterscheiden dürfte, wird dieser Punkt für die Wahl des Systems außer Acht gelassen.

Weder TinyOS noch SOS kommen für die Lösung der in dieser Arbeit gestellten Aufgabenstellung in Frage. TinyOS, weil es nicht möglich ist, Threads nach dem Compilierzeitpunkt hinzuzufügen und SOS, weil es ein rein ereignisbasiertes OS ist. Zwar gibt es, wie bereits erwähnt, eine, wenn auch nicht gewartete, Implementierung von *Protothreads*. Eine Portierung von Contikis Thread-Bibliothek ist mit hoher Wahrscheinlichkeit möglich, jedoch ist der Aufwand sehr schwer abzuschätzen. Des Weiteren müssen höchstwahrscheinlich auch Anpassungen an der Modulunterstützungen vorgenommen werden, um Threads in Modulen zu unterstützen.

¹ „BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks : Main - Overview browse“, <http://www.btnode.ethz.ch/Documentation/BTnutOverview>

² Es werden Threads unterstützt, jedoch können nach dem Compilierzeitpunkt keine neuen Threads hinzugefügt werden.

Für die Verwendung von Contiki spricht, dass Module bereits unterstützt werden. Der größte Aufwand besteht in der Portierung des Bluetooth Stacks. Der Aufwand ist aber ohne eine sehr genaue Analyse des Bluetooth Stacks sowie der verwendeten Hardware nur schwer abzuschätzen. Auch die Modulunterstützung müsste für die AVR-Plattform portiert werden oder der Linker so erweitert werden, dass mehr als ein Modul unterstützt werden.

BTnut hingegen hat noch keine Modulunterstützung. Zum Start eines neuen Threads wird jedoch lediglich ein Name, Stackgröße und ein Einsprungpunkt benötigt. Für die Ausführung des Threads benötigter Arbeitsspeicher wird dynamisch allokiert. Dies ermöglicht eine einfache Implementierung der Unterstützung von Modulen.

Für die Aufgabenstellung scheint BTnut das am besten geeignete Betriebssystem. Der Aufwand ist überschaubar und der Quellcode sehr gut organisiert und dokumentiert. Dies ermöglicht eine rasche Einarbeitung in die Funktionsweise des Betriebssystems. Des Weiteren besitzt BTnut bereits einen Bluetoothstack, so dass auch Bluetooth ohne weiteren Aufwand als Kommunikationsmedium verwendet werden kann. Ein zusätzlicher Vorteil ist, dass aktuell auf dem BTnode laufende Projekte nicht auf ein anderes Betriebssystem portiert werden müssen, um die Modulunterstützung zu nutzen.

5 Implementierung einer Modulunterstützung für BTnut

Die beiden Hauptaufgabenstellungen sind die Softwareaktualisierung zur Laufzeit und die Wiederherstellung von Statusinformationen und Daten nach einer solchen Aktualisierung. Die in Kapitel 3.4 beschriebenen Verfahren zur Softwareaktualisierung zur Laufzeit des Threads sind sehr aufwändig und eignen sich nur bedingt für die Verwendung in drahtlosen Sensornetzwerken. Aus diesem Grund wurden die beiden Anforderungen getrennt: Zum einen in eine Modulunterstützung und zum anderen in die Möglichkeit Daten im SRAM auch über eine Softwareaktualisierung hinweg abzulegen. Letzteres wird in Kapitel 6 beschrieben.

Für die Softwareaktualisierung werden „pre-gelinkte“ Module verwendet (Kap. 3.5.1). Diese haben den Vorteil, dass sie sehr klein und im Vergleich zu VMs deutlich effektiver sind.

Im Folgenden werden die Anforderungen nochmals präzisiert und erweitert. Anschließend wird das Lösungskonzept vorgestellt um anschließend detailliert auf die Implementierung einzugehen.

5.1 Anforderungen

An die Modulunterstützung werden verschiedenen Anforderungen gestellt. Die in der Einleitung vorgestellten Anforderungen sollen an dieser Stelle nochmals präzisiert und erweitert werden.

Nachladen von Code zur Laufzeit

Es soll möglich sein, mehrere Module zur Laufzeit nachzuladen. Dies bedeutet, dass das System nach dem Laden des Moduls nicht neu gestartet werden muss und andere auf dem Knoten laufende Threads nicht beendet oder beeinflusst werden.

Einfache Anwendung

Die Lösung muss so implementiert werden, dass sie ohne umfangreiche Einarbeitung und Wissen über das zugrunde liegende System verwendet werden kann. Dies gilt nicht nur für die Programmierung, sondern auch für das Erstellen und Laden neuer Module auf den Knoten. Da das Kommunikationsmedium nicht vorgegeben ist, muss eine Schnittstelle entworfen werden, welche die Funktionalität soweit abstrahiert, dass eine einfache Portierung auf andere Übertragungsprotokolle möglich ist. Auch für die Aktualisierung des Kernels soll diese Schnittstelle verwendet werden. Hierdurch wird vermieden, dass je ein System zum Laden von Modulen und eines zum Aktualisieren des Kernels gewartet werden muss.

Praxistauglichkeit

Ein großes Problem in der Praxis ist, dass auf verschiedenen Knoten häufig verschiedene Kernelversionen installiert sind. Selbst bei der Verwendung des gleichen Quellcode können unterschiedliche Compiler- und Linkeroptionen sowie -versionen unterschiedlichen Binärcode zur Folge haben (siehe Kap. 3.5.1). Eine Versionsnummer ist daher zum Abgleich nicht ausreichend. Dieses Problem wird verschärft, wenn mehrere Personen mit dem selben Sensornetzwerk arbeiten. Es muss

also im Lösungskonzept sichergestellt werden, dass das Modul stets für den richtigen Kernel erstellt wird und es auch möglich ist, Module für einen Kernel zu erstellen, welcher von einer anderen Person auf einem anderen Rechner erstellt wurde.

Robustheit

Die Erkennung von Datenfehlern durch Prüfsummen ist bei der Reprogrammierung üblich. In diesem Fall ist die Erkennung von Fehlern jedoch besonders wichtig, da davon ausgegangen wird, dass die Knoten nur schwer zu erreichen sind und daher nur über Funk reprogrammierbar sind. Wird ein beschädigter Kernel installiert ist eine Kommunikation mit dem Knoten und somit die Aktualisierung des Knotens mit einer neuen, korrigierten Version meist nicht mehr möglich.

5.2 Lösungskonzept

Im Folgenden wird das für die Erfüllung des Anforderungen aufgestellte Lösungskonzept vorgestellt. Die genaue Implementierung wird im nachfolgenden Kapitel ausgeführt.

Einbindung in Nut/OS

Zum Starten eines neuen Threads benötigt Nut/OS lediglich die Stackgröße, den Namen des Threads, sowie die Einsprungadresse. Diese Informationen können im Header eines Moduls gespeichert werden. Die Problemstellung liegt also darin Module so zu linkern, dass diese die vom Kernel zur Verfügung gestellten Funktionen nutzen.

Verwaltung mehrerer Module

Um mehrere Module auf dem Knoten speichern zu können, muss deren Position sowie der für weitere Module freie Speicher bekannt sein. Hierzu wird eine einfache Flash-Speicherverwaltung verwendet. Diese ist aus aufeinander folgenden Blöcken aufgebaut, wobei im Header jedes Blocks die jeweilige Größe gespeichert ist. Ein Block kann entweder einen Kernel, ein Modul oder unbenutzten Speicher enthalten.

Erstellung des Moduls

Soll ein Modul gegen den Kernel gelinkt werden, werden verschiedene Informationen benötigt. Hierzu gehören die Symboltabelle des Kernels und freie Adressbereiche zur Positionierung der `.text`, `.data` und `.bss` Segmente. Freier Speicher für die Positionierung des `.text` Segments kann über die Flash-Speicherverwaltung bestimmt werden.

Positionierung der Datensegmente

Im Gegensatz zum `.text` Segment ist die Platzierung des `.data` und `.bss` Segments nicht unproblematisch. Da der gesamte ungenutzte Speicher der Speicherverwaltung zur Verfügung steht, müsste benötigter Speicher von der Speicherverwaltung allokiert werden. Auch muss ein von einem Modul verwendeter Speicher exklusiv für das Modul reserviert werden, da ein Verschieben der Segmente nach der Programmierung auf den Mikrocontroller nicht mehr, oder nur mit erheblichem Aufwand, möglich ist. Die Reservierung muss daher zum Zeitpunkt der Systeminitialisierung

passieren. Wird der Speicher anderweitig verwendet, ist es nicht mehr möglich das Modul zu starten. Zur Vereinfachung der Aufgabenstellung werden globale und statische Variablen, welche in den `.data` und `.bss` Segmenten gespeichert werden, in Modulen zunächst nicht unterstützt.

Identifikation der Symboltabelle des Kernels

Der Kernel und seine Symboltabelle werden über eine Prüfsumme identifiziert. Während der Erstellung des Kernels wird eine Prüfsumme für den Binärcode des Kernels berechnet. Die Symboltabelle wird in einer Datei, welche die Prüfsumme im Dateinamen enthält, abgelegt. Auf diese Weise ist es zu einem späteren Zeitpunkt möglich auf die Symboltabelle eines Kernels zuzugreifen.

Überprüfung der Integrität

Um die Integrität zu überprüfen, wird nicht nur für den Kernel, sondern auch für die Module eine Prüfsumme berechnet. Diese wird anschließend im Kernel, beziehungsweise in dem Modul, gespeichert. Zur Überprüfung wird die Prüfsumme auf dem Knoten erneut berechnet und mit der gespeicherten Prüfsumme verglichen. Bei der Berechnung muss die gespeicherte Prüfsumme maskiert werden.

Bootloader

Zur Reprogrammierung wird Code im Bootloader-Bereich benötigt. Dieser muss zwei Funktionen erfüllen: Erstens das Ersetzen des Kernels und zweitens das Bereitstellen von Funktionen um vom Kernel aus die Module in den Flash zu schreiben. Die Ersetzung des Kernels funktioniert wie auf vielen anderen Systemen auch. Der Mikrocontroller wird neu gestartet, der Bootloader überprüft ob ein neuer Kernel zu Verfügung steht und ersetzt gegebenenfalls den alten Kernel. Um vom Kernel aus Module speichern zu können, wird eine *Jump Table* mit drei Funktionen zur Verfügung gestellt: Eine Versionsabfrage, falls sich an der Funktionalität des Bootloaders etwas ändert, das Löschen sowie das Schreiben einer Seite.

5.3 Implementierung

In diesem Kapitel wird die Implementierung des in vorherigen Kapitel vorgestellten Konzepts beschrieben. Abbildung 5.1 zeigt das Speicherlayout, nachdem ein Modul installiert wurde. Am Anfang befindet sich der Kernel, dahinter ein Modul. Ab der Adresse `0x1E000` ist der Bootloader abgelegt. Nur der Bootloader kann den Flash reprogrammieren. Außerhalb des Bootloaders wird der Reprogrammierbefehl ignoriert. Die in der Abbildung gezeigten Header sind Teil der Speicherverwaltung. Diese wird benötigt, da die Positionen der installierten Module sowie die freien Bereiche bekannt sein müssen.

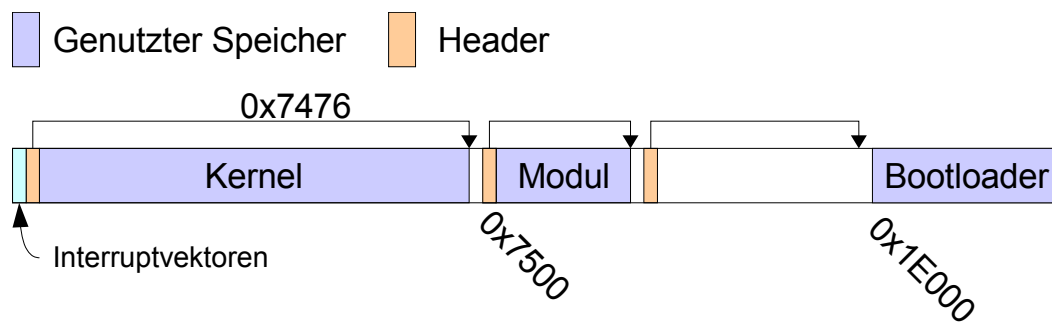


Abbildung 5.1: Übersicht des Speicherlayouts nach dem Laden eines Moduls

Zunächst wird nun nochmal sehr detailliert auf die Funktionsweise der Selbstprogrammierung des ATmega128 eingegangen und anschließend der Bootloader erklärt. Danach werden die Erstellung von Kernen und Modulen beschrieben. Abgeschlossen wird dieses Kapitel mit der Beschreibung der Schnittstelle um Module zu verwalten.

5.3.1 Reprogrammierung des ATmega128

Der ATmega128 Mikrocontroller besitzt 128 KiB Flash-Programmspeicher, in welchem der Programmcode gespeichert wird. Der nichtflüchtige Speicher kann vom Mikrocontroller reprogrammiert werden.

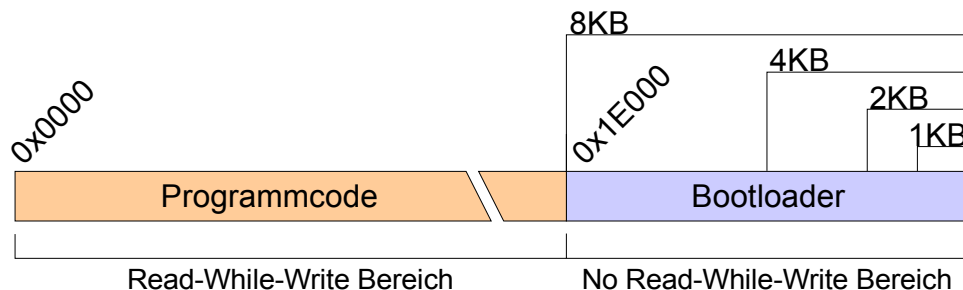


Abbildung 5.2: RWW, NRWW sowie Bootloaderbereich im Flash

Die verschiedenen Bereiche des Flashspeichers haben spezielle Funktionen und Eigenschaften. Der Speicher wird in den Read While Write (RWW) und den No Read While Write (NRWW) Bereich aufgeteilt, welcher sich in den letzten 8KiB des Speichers befindet (vgl. Abb. 5.2). Während in den NRWW Bereich geschrieben oder gelöscht wird, wird die CPU angehalten und es ist nicht möglich gleichzeitig weitere Befehle auszuführen. Wird hingegen in den RWW-Bereich geschrieben, ist es immer noch möglich im NRWW-Bereich liegenden Code auszuführen.

Des Weiteren gibt es einen Bootloaderbereich. Dieser liegt im NRW-Bereich. Der Anfang des Bootloaderbereichs kann jedoch über Fuses verändert werden. So ist es möglich die vollen 8 KiB des NRW-Bereichs oder nur die letzten 4, 2 oder 1 KiB für den Bootloader zu nutzen. Über eine weitere Fuse ist es möglich den Resetvektor an den Anfang des Bootloaderbereichs zu legen, so dass nach einem Reset nicht das normale Programm sondern der Bootloader ausgeführt wird.

Der Reprogrammierbefehl wird nur erfolgreich ausgeführt, wenn er sich innerhalb des Bootloaderbereichs befindet. Neben der Möglichkeit, den Reprogrammierbefehl nach einem Reset durch einen Bootloader auszuführen, kann eine entsprechende Funktion auch von außerhalb des Bootloaderbereichs angesprochen werden. Auf diese Weise ist es möglich den Flash zum Beispiel vom Kernel aus zu reprogrammieren.

Während der Reprogrammierung müssen die Interrupts im Normalfall abgeschaltet werden, da diese im RWW-Bereich liegen, auf welchen während des Schreibens nicht zugegriffen werden kann. Alternativ ist es möglich die komplette Interrupttabelle an den Anfang des Bootloaderbereichs zu verlegen. So können die Interrupts im NRW-Bereich behandelt werden, während der RWW-Bereich reprogrammiert wird.

Reprogrammiert wird der Flashspeicher seitenweise. Hierzu muss eine Seite zunächst gelöscht werden, bevor sie beschrieben werden kann. Während hardwareseitig sichergestellt ist, dass Löschen- und Schreibvorgang auch bei einem Wegfall der Versorgungsspannung stets abgeschlossen werden, ist es möglich, dass es zu einer Unterbrechung zwischen diesen beiden Schritten kommt und eine Seite gelöscht, jedoch nicht wieder geschrieben wird.

Dies ist dann zu beachten, wenn nur ein Teil einer Seite verändert werden soll, und der Inhalt im Arbeitsspeicher zwischengespeichert wird. In einem solchen Fall gehen nicht nur die Änderungen sondern auch der vorherige Inhalt verloren.

5.3.2 Aufbau und Funktion des Bootloaders

Der Bootloader erfüllt mehrere Funktionen. Dies sind die Überprüfung der Systemintegrität, das Kopieren des Kernels, die Bereitstellung von Funktionen, welche zum Schreiben auf den Flashspeicher benötigt werden, sowie das Starten des Kernels.

Sind die „Fuses“ des Mikrocontrollers richtig konfiguriert, wird nach einem Reset zunächst der Bootloader gestartet, welcher folgende Schritte ausführt:

1. Überprüfung und Wiederherstellung der Integrität der Flashverwaltung
2. Prüfen, ob ein aktualisierter Kernel gespeichert wurde und gegebenenfalls Überschreiben des alten durch den neuen Kernel
3. Starten des Kernels

Um die Berechnung der CRC32 Prüfsummen, welche zur Verifizierung von Kernel und Modulen verwendet werden, zu beschleunigen, wird zunächst eine *Lookup Table* berechnet. Anschließend wird die Prüfsumme des Kernels verifiziert. Ist diese fehlerhaft, wird der Flashspeicher nach einem

gültigen Kernel durchsucht und dieser gegebenenfalls installiert. Anschließend werden auch die installierten Module verifiziert. Hierzu werden deren Prüfsummen validiert und überprüft, ob diese für den aktuell vorhandenen Kernel kompiliert wurden. Fehler in der verketteten Liste der Flashspeicherverwaltung werden behoben und unvollständige, oder fehlerhafte Module entfernt.

Ist ein neuer Kernel vorhanden, wird der alte Kernel durch diesen ersetzt. Die Integrität des neuen Kernels wurde bereits während der Überprüfung der Flash-Speicherverwaltung sichergestellt. Erst wenn auch die Integrität der Kopie überprüft wurde, wird der Speicher hinter dem Kernel freigegeben und so alle dort befindlichen Module und die gepufferte Version des installierten Kernels gelöscht. Dies ist sinnvoll, da Module speziell für Kernel kompiliert werden müssen und mit dem neuen Kernel nicht mehr lauffähig sind. Schlägt der Kopiervorgang fehl, wird der Bootloader neu gestartet und der Kernel erneut kopiert.

Ist die Systemintegrität sichergestellt und gegebenenfalls ein neuer Kernel installiert, wird vom Bootloader die Adresse 0×00 angesprungen. Dies ist der Resetvektor des Kernels. Anschließend wird der Mikrocontroller durch den Kernel erneut initialisiert.

Um auf die Funktionen zum Schreiben auf den Flashspeicher zuzugreifen, wird eine *Jump Table* verwendet (vgl. Kap. 3.5.3). Über diese werden drei Funktionen zur Verfügung gestellt: `bootVersion()`, `erasePage()` und `flashPage()`. Mit diesen ist es möglich die Version des Bootloaders abzufragen, Seiten zu löschen und zu beschreiben. Die *Jump Table* befindet sich direkt hinter den alternativen Interruptvektoren, so dass die Position stets bekannt ist. Die Adressen der Funktionen innerhalb der *Jump Table* werden als Symboltabelle an den Linker übergeben.

5.3.3 Flash-Speicherverwaltung

Um Module nachladen zu können, muss das System die installierten Module und deren Position im Flashspeicher kennen. Hierzu wurde eine einfache Flashspeicherverwaltung entwickelt. Der Speicher wird in aufeinander folgende Blöcke unterteilt. Die Blöcke sind an den Seiten des Speichers ausgerichtet. Jeder Block beginnt mit Header-Informationen. Diese enthalten Informationen über die Art, den Inhalt und die Größe des Blockes.

In Abbildung 5.3 sieht man exemplarisch den Aufbau der Speicherverwaltung. Da die Position der Interruptvektoren durch die Hardware vorgegeben ist, muss der Header des Kernels hinter diesen gespeichert werden. Der Header enthält Informationen über die Endadresse des Blocks, in diesem Fall 0×7476 . Da die Elemente an den Seiten ausgerichtet sind, muss das nächste Element an der Adresse 0×7500 beginnen.

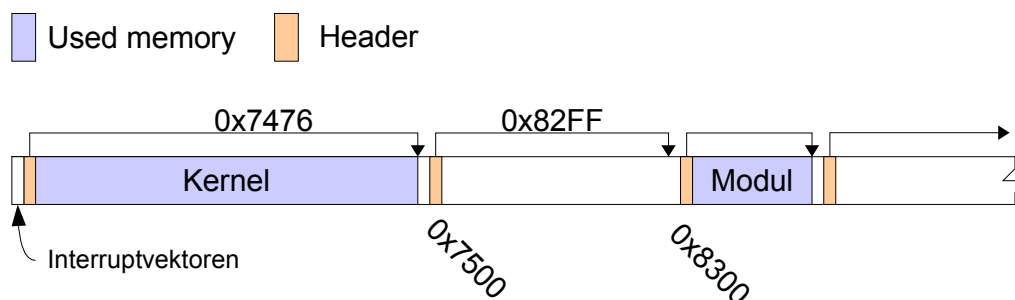


Abbildung 5.3: Aufbau der Flashspeicherverwaltung

Ein Block kann entweder leer sein oder ein Modul enthalten. Ein Modul ist eine zusammenhängende Menge Binärcode. Das Modul enthält eine Einsprungadresse, mit welcher eine Applikation gestartet werden kann. Diese kann ausgeführt werden, nachdem das Modul auf dem Knoten gespeichert wurde.

Der Header enthält verschiedene Informationen über den Block und ist folgendermaßen aufgebaut:

```
struct app_header_t {
    uint16_t endadd;
    char name[APP_NAME_LEN];
    void (*entryfn) (void *);
    uint16_t page;
    uint16_t stacksize;
    union mod_crc32 crc32;
    union mod_crc32 kernel_crc32;
    uint8_t deleted : 1;
    uint8_t kernel : 1;
};
```

In `endadd` wird die Endadresse des Blocks als Wort gespeichert. `name` enthält den Namen des Blocks. Dies ist auch gleichzeitig der Name des enthaltenen Moduls oder Applikation. Für freien Speicher wird diese Variable nicht ausgewertet. Da Module immer für eine spezielle Adresse kompiliert werden, wird in `page` die Seite gespeichert, in welcher der Block gespeichert werden muss. Enthält ein Modul eine Applikation, so wird die Einsprungadresse der Applikation in `entryfn` und die benötigte Stackgröße in `stacksize` abgelegt. Die beiden Flags dienen zur Erkennung eines Kernels (`kernel`) oder ob ein Speicherbereich frei ist (`deleted`). Des Weiteren werden zwei `CRC32` Prüfsummen gespeichert: In `crc32` wird die Prüfsumme des Blocks gespeichert. Diese berechnet sich über den gesamten Block, wobei die Prüfsumme selber mit 0 maskiert werden muss. In `kernel_crc32` wird die Prüfsumme des Kernels gespeichert, für welchen ein Modul kompiliert wurde. Hiermit wird vermieden, dass ein Modul auf einen Knoten mit einem Kernel, für welchen er nicht kompiliert wurde, installiert wird.

Um die Verwaltungsstrukturen leicht wiederherzustellen zu können und die Speicherverwaltung robuster zu machen, wird der Speicher zunächst, bei der letzten Seite eines Blocks beginnend, gelöscht und anschließend wieder vorwärts beschrieben. Die einzelnen Schritte werden in Abbildung 5.4 beschrieben.

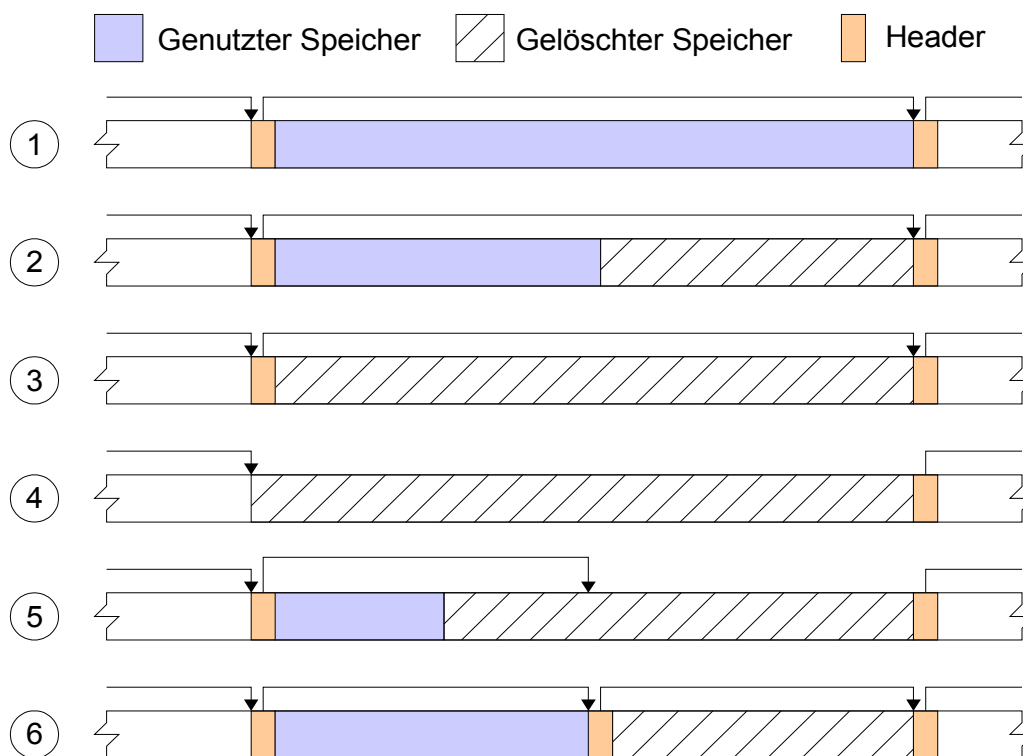


Abbildung 5.4: Löschen des Flashspeichers

- (1) Die Aneinanderreihung von Blöcken im Ausgangszustand.
- (2) Die Hälfte des Blocks ist gelöscht. Die Verwaltungsstrukturen sind immer noch gültig.
- (3) Solange die erste Seite des Blocks noch nicht gelöscht ist, bleibt die Verwaltungsstruktur gültig.
- (4) Ist auch die erste Seite gelöscht, kann die Verwaltung wiederhergestellt werden, indem die erste Seite gesucht wird, welche Daten enthält.
- (5) Die Daten werden von vorne nach hinten geschrieben. Auch wenn sich hinter dem Block noch keine Daten befinden, lassen sich die Verwaltungsstrukturen durch die Suche nach der ersten beschriebenen Seite wiederherstellen.
- (6) Nachdem auch der Header für den freien Bereich geschrieben wurde, sind die Verwaltungsstrukturen wieder gültig.

Wird ein Block freigegeben, wird dieser gegebenenfalls mit davor oder dahinter liegenden Blöcken zusammengefasst.

5.3.4 Erstellung und Ersetzung des Kernels

Die Erstellung des Kernels unterscheidet sich von der Erstellung eines normalen Speicherabbildes für einen Mikrocontroller vor allem dadurch, dass er den Header der Flashverwaltung enthalten muss. Der Header wird mit folgendem Code eingefügt:

```
const struct app_header_t kernel_header __attribute__((section
(".progmem"))) = {
    .endadd = (int16_t) &__endadd,
    .name = "kernel",
    .entryfn = 0,
    .crc32.split.crc32_lo = (int16_t)&_kernel_crc32_lo,
    .crc32.split.crc32_hi = (int16_t)&_kernel_crc32_hi,
    .page = 0,
    .deleted = 0,
    .kernel = 1,
};
```

Der Befehl `__attribute__((section(".progmem)))` lässt den Compiler die Struktur in die Sektion `.progmem` speichern. Diese befindet sich direkt hinter den Interruptvektoren. `__endadd` ist ein spezielles Symbol, welches folgendermaßen in einer Symboltabelle an den Linker übergeben wird:

```
__endadd = __data_load_end / 2;
```

`__data_load_end` ist ein vom Linkerskript generiertes Symbol und enthält die erste Adresse hinter der `.data` Sektion auf dem Flash. Für die Endadresse wird ein Wort¹ als Einheit verwendet, um den Adressraum auf 16 Bit zu reduzieren. Aus diesem Grund muss die Adresse halbiert werden. Dies ist problemlos möglich, da alle Sektionen immer auf zwei Byte aufgerundet und gegebenenfalls mit `0x00` gefüllt werden. Symbole entsprechen immer einer im Speicher liegenden Variablen. Da jedoch die Adresse und nicht deren Inhalt in `endadd` abgelegt werden soll, muss der `&`-Operator verwendet werden.

Die Prüfsumme der Binärdatei wird auch durch Symbole übergeben. Da Zeiger auf Atmega128 Systemen maximal 16 Bit groß sind, müssen jedoch zwei Symbole verwendet werden um die 32 Bit große `CRC32` Prüfsumme zu übergeben. Zunächst wird der Kernel mit der Prüfsumme 0 gelinkt und das Binärabbild erstellt. Da die Prüfsumme 0 ist, muss sie nicht mehr maskiert werden und die Prüfsumme kann direkt über das Abbild berechnet werden. Anschließend wird der Kernel nochmals mit der richtigen Prüfsumme gelinkt. Es wäre zwar möglich, die richtige Prüfsumme direkt in das Binärabbild zu schreiben, jedoch hat das angewandte Vorgehen den Vorteil, dass die richtige Prüfsumme auch in der ELF-Datei steht. Dies ermöglicht die Verwendung der ELF-Datei mit einem Debugger oder die problemlose Extraktion des Binärabbildes im Intel-Hex Format, welches sich zum Beispiel für die Übertragung über die serielle Schnittstelle eignet.

¹ Ein Wort sind zwei Byte

Um später Module gegen den Kernel linken zu können, muss die Symboltabelle des Kernels extrahiert werden. Dies ist zum Beispiel mit `objdump` möglich:

```
avr-objdump -t kernel.elf
```

Das folgende Beispiel zeigt einen Ausschnitt aus der Ausgabe für einen Kernel zur Reprogrammierung über die serielle Schnittstelle:

```
kernel.elf:      file format elf32-avr

SYMBOL TABLE:
[.]
00000ef6 g      F .text      00000040 pageToByte
00002a14 g      F .text      00000058 NutRegisterDevice
00800400 g      O .bss       00000002 ds
00800385 g      O .data      00000006 sig_UART0_DATA
0080022e g      O .data      00000001 idle_sleep_mode
00002742 g      F .text      000000e4 crc32_init_table
000039cc g      F .text      000000ce NutDumpHeap
00006a2c g      F .text      00000012 memcpy
[.]
```

Man kann an dieser Tabelle zum Beispiel erkennen, dass die Funktion `pageToByte()` in der `.text` Sektion auf dem Flash an der Adresse `0x0ef6` liegt. Eine Besonderheit stellen Variablen dar. Da die `gcc` Toolchain von sich aus keine Harvardarchitekturen unterstützt, wird hier ein Trick angewandt: Die `.data` und `.bss` Sektionen werden in den Adressbereich oberhalb von `0x80100` gelegt – die ersten 255 Byte sind durch Register belegt. Da der Mikrocontroller jedoch nur einen 16bit Adressraum besitzt, werden die ersten beiden Byte der Adresse nicht im extrahierten Speicherabbild gespeichert. Die Variable `ds` hat daher nicht die Adresse `0x800400` sondern `0x0400`. Die weiteren zur Verfügung gestellten Informationen spielen für die Erstellung der Symboltabelle keine Rolle und werden daher nicht weiter erläutert.

Der Linker benötigt die Symboltabelle in der Form:

```
pageToByte      = 0x00000ef6
NutRegisterDevice = 0x00002a14
ds              = 0x00800400
sig_UART0_DATA  = 0x00800385
idle_sleep_mode = 0x0080022e
crc32_init_table = 0x00002742
NutDumpHeap     = 0x000039cc
memcpy          = 0x00006a2c
```

Diese Umwandlung wird von einem PHP-Skript übernommen. Die Symboltabelle wird in ein Linkerskript integriert und in einem Unterverzeichnis abgespeichert. Die `CRC32` Prüfsumme des Kernels wird in den Dateinamen integriert, um zu einem späteren Zeitpunkt auf die richtige Symboltabelle zugreifen zu können.

5.3.5 Einbindung des Bootloaders in das Kernelabbild

Der Kernel und der Bootloader sind voneinander unabhängig. Es ist daher möglich, diese nacheinander auf dem Mikrocontroller zu installieren. Einige Debugger, wie das AVR-Studio, löschen jedoch den gesamten Speicher bevor sie das Abbild des Kernels auf den Knoten schreiben. Dies bedeutet, dass nach jeder Korrektur des Kernels auch der Bootloader erneut auf den Knoten geladen werden muss, obwohl sich dieser nicht geändert hat. Soll der Bootloader mit dem Kernel verknüpft werden, müssen verschiedene Dinge beachtet werden:

- Der Bootloader muss mit in die `.elf`-Datei gespeichert werden, da diese vom Debugger geladen wird
- Der Bootloader darf erst beim zweiten Linken des Kernels hinzugefügt werden, da die Prüfsumme nur über den Kernel, nicht aber über den Bootloader berechnet werden darf.
- Das Hinzulinken des Bootloaders darf keinen Einfluss auf das Speicherlayout des Kernels haben.

Da sich die Speicherbereiche von Kernel und Bootloader zwar nicht im Programmspeicher, wohl aber im Datenspeicher überschneiden, ist es möglich, dass das Hinzulinken des Bootloaders das Speicherlayout der `.data` und `.bss` Sektionen beeinflusst. Als einfachste Lösung hat sich das Einbinden des Bootloaders als Binärdaten herausgestellt.

Der hierzu verwendete Befehl lautet:

```
avr-objcopy --rename-section
.data=.btl.text,contents,alloc,load,readonly,code -I binary -O elf32-avr
bootloader.bin bootloader.bin.o
```

Dieser Befehl wandelt Binärdaten (`-I binary`) in ein AVR-Objekt um (`-O elf32-avr`), welches wiederum an den Linker übergeben werden kann. Standardmäßig werden Binärdaten in die `.data` Sektion gespeichert. Um auf diese zuzugreifen werden die beiden Symbole `_binary_foo_bin_start_` und `_binary_foo_bin_end_` vom Linker bereitgestellt. Dies kann zum Beispiel auch dazu genutzt werden, um Binärdaten, wie Bilder, in Programmcode zu integrieren. In diesem Fall soll jedoch kein Speicher im `.data` Segment, also Arbeitsspeicher, sondern im Programmspeicher reserviert werden. Aus diesem Grund werden die Daten über den `-rename-section` Parameter in die die Sektion `.btl.text` gespeichert. Dies ermöglicht es die Position der Binärdaten beim Linkvorgang des Kernels genauer zu spezifizieren (siehe auch Kap 2.10). Die weiteren übergebenen Parameter sind die für die `.text` Sektion verwendeten Standardparameter.

Wird der Kernel zum zweiten Mal gelinkt, wird dem Linker sowohl das Objekt mit den Binärdaten des Bootloaders, als auch der Parameter `-section-start=.btl.text=0x1E000` übergeben. Dieser bewirkt, dass der Bootloader an die Position `0x1E000` gespeichert wird. Der Nachteil

dieser Lösung ist, dass beim Debuggen nur der disassemblierte Code, jedoch keine Symbole zur Orientierung zur Verfügung stehen. Dennoch wird der Programmiervorgang durch die Einbettung des Bootloaders erheblich erleichtert.

5.3.6 Aufbau und Erstellung eines Moduls

Sind Kernel und Bootloader auf dem Knoten installiert, können Module erstellt werden. Module enthalten eine Applikation, welche auf dem Knoten gestartet werden kann, sowie die nötigen Strukturen, um das Modul als Block auf dem Flash speichern zu können. Module, die keine Applikation sondern zum Beispiel eine Funktionsbibliothek enthalten, werden noch nicht unterstützt.

Eine Beispielapplikation kann folgendermaßen aussehen:

```
APPLICATION("PrintU", 192, arg){
    for (;;) {
        if(app_gotShutdown() == 1) {
            putchar('E');
            NutThreadExit();
        }
        putchar('U');
        NutSleep(125);
    }
}
```

Hierdurch wird eine Applikation mit dem Namen PrintU mit der Stackgröße 192 erstellt. Der Parameter arg ist der Name eines void Zeigers, mit welchem es möglich ist Daten an die Applikation zu übergeben. Diese Funktionalität wird zunächst nicht unterstützt, kann aber bei Bedarf implementiert werden. Die Beispielapplikation gibt in einer Endlosschleife U auf die Standardausgabe aus und schläft dann für 125 ms. Hat die Applikation den Befehl bekommen sich zu beenden, liefert app_gotShutdown() den Wert 1 zurück. Als Folge wird der Buchstabe E ausgegeben und die Applikation beendet.

Hinter dem Schlüsselwort APPLICATION versteckt sich ein umfangreiches Makro, welches die benötigten Informationen einfügt.

```
#define APPLICATION(_name, _stacksize, arg) \
void APP_FUNC(void *arg) __attribute__((noreturn)); \
const struct app_header_t app_header __attribute__((section \
(".block_header"))) = { \
    .endadd = (uint16_t) &__endadd, \
    .name = _name, \
    .entryfn = &APP_FUNC, \
    .crc32.split.crc32_lo = (uint16_t)&__mod_crc32_lo, \
    .crc32.split.crc32_hi = (uint16_t)&__mod_crc32_hi, \
    .kernel_crc32.split.crc32_lo = (uint16_t)&__ker_crc32_lo, \
    .kernel_crc32.split.crc32_hi = (uint16_t)&__ker_crc32_hi, \
    .page = (uint16_t)&__mod_page, \
    .stacksize = _stacksize, \
    .deleted = 0 \
}; \
void APP_FUNC(void *arg)
```

Das Makro definiert zunächst einen Prototypen der Funktion `APP_FUNK()`. Der Parameter `__attribute__((noreturn))` sagt dem Compiler, dass diese Funktion niemals zurückkehren darf. Dies ist wichtig, da die Funktion vom Betriebssystem initialisiert wird und keine Rücksprungsadresse hat. Sie muss durch den Befehl `NutThreadExit()` beendet werden.

Anschließend wird der Header, der für die Flashspeicherverwaltung benötigt wird, definiert. Dieser wird in der Sektion `.block_header` abgelegt. In dem Linkerskript, welches bei der Erstellung des Kernels erstellt wird, ist diese Sektion direkt am Anfang des Speichers definiert.

Zusätzlich zu den Parametern, die bereits vom Kernel verwendet werden, werden bei Modulen zusätzlich die Parameter `name`, `entryfn`, `stacksize` und `kernel_crc32` verwendet. Der Name des Blocks und somit der Applikation wird in `name` gespeichert. `entryfn` zeigt auf die Funktion `APP_FUNK`, weshalb zuvor deren Prototyp definiert werden musste. Die benötigte Stackgröße wird in `stacksize` gespeichert. In `kernel_crc32` wird die Prüfsumme des Kernels geschrieben. Die entsprechenden Symbole sind bereits im Linkerskript enthalten, so dass eine falsche Zuweisung praktisch ausgeschlossen ist. Durch die Speicherung der Prüfsumme des Kernels wird sichergestellt, dass das Modul nur von dem Kernel verwendet wird, für welchen es kompiliert wurde. Schließlich wird die Funktion, welche mit `APPLICATION` definiert wurde, als `APP_FUNC` eingeleitet. Die statische Verwendung dieses Funktionsnamens ist unproblematisch, da maximal eine Applikation je Modul gespeichert werden kann.

Die Erstellung eines Moduls geschieht in mehreren Schritten. Zunächst muss die Prüfsumme des Kernels vom Knoten erfragt werden. Ist diese bekannt, kann das Modul das erste Mal kompiliert werden. Nach der ersten Compilation ist die Größe des Moduls bekannt. Mit diesem Wissen kann nun Speicher auf dem Knoten allokiert werden. Die vom Knoten zurückgelieferte Anfangsadresse für das Modul wird mit dem Parameter `-Text=` an den Linker übergeben. Damit wird der Linker angewiesen, die Textsektion an der angegebenen Adresse beginnen zu lassen.

Nach dem Linken des Moduls an die richtige Stelle kann, wie beim Kernel, die `CRC32` Prüfsumme berechnet werden und in einem dritten Linkvorgang in das Modul integriert werden. Wird das Modul zum dem Knoten übertragen und dort in den Speicherbereich geschrieben, für welches es kompiliert wurde, kann es ohne weitere Modifikation des Binärcodes ausgeführt werden.

Bei der Programmierung des Flashspeichers zur Laufzeit muss beachtet werden, dass der Flashvorgang ca. 4 ms dauert. Während dieser Zeit ist es nicht möglich Interrupts zu behandeln (siehe auch Kapitel 5.3.1).

5.3.7 Implementierung der Modulschnittstelle

Zur Speicherung von Kernen und Modulen wurden verschiedene Funktionen zur Abstraktion der zugrundeliegenden Verwaltungsstrukturen geschrieben. Diese sind in der Lage einen Kernel oder Modul als Bytestrom zu verarbeiten. Alle hierzu benötigten Daten wie Name, Größe oder ob es

sich um ein Modul oder Kernel handelt, werden automatisch aus den Daten extrahiert. Hierdurch wird ein maximaler Abstraktionsgrad erreicht, welcher eine schnelle Implementierung auf einem beliebigen Übertragungsmedium ermöglicht.

Während der Erstellung von Modulen wird sowohl die Prüfsumme des Kernels als auch ein freier Speicherbereich benötigt. Zum Erfragen der Prüfsumme steht die Funktion `mod_kernelCRC()` zur Verfügung. Diese liest die Prüfsumme aus dem Header des Kernels und liefert sie als 32bit Integerwert zurück.

Die Funktion `app_loadmod_requestMem()` durchsucht den Flashspeicher nach einem freiem Block, an welchem das Modul gespeichert werden kann. Hierbei wird die *first fit* Methode verwendet. Des Weiteren wird auch Speicher, an welchem sich bereits eine Version des Moduls befindet, als frei interpretiert. Aus diesem Grund benötigt die Funktion nicht nur die Größe, sondern auch den Namen des zu speichernden Moduls. Der zurückgegebene Speicher wird nicht explizit für dieses Modul reserviert. Findet gleichzeitig eine zweite Abfrage statt, bevor das Modul installiert wurde, kann es zu Überschneidungen kommen.

Wurde ein neuer Kernel oder Modul erstellt, kann es zum Knoten übertragen und in den Flash geschrieben werden. Hierzu stehen drei Funktionen zur Verfügung: `app_loadmod_init()`, `app_loadmod_data()` und `app_loadmod_finish()`.

Mit `app_loadmod_init(block)` werden die benötigten Strukturen und ein Puffer initialisiert. Mit dem Parameter `block` wird konfiguriert, ob der Flash beschrieben wird, sobald ausreichend Binärdaten zum Schreiben einer Seite zur Verfügung stehen, oder nach dem Aufruf von `app_loadmod_finish()`. An die `app_loadmod_data()`-Funktion werden die Binärdaten als Datenstrom übergeben. Ist die Datenübertragung beendet, wird der Vorgang mit `app_loadmod_finish()` abgeschlossen. Im Folgenden wird nun zunächst der fehlerfreie Programmiervorgang beschrieben. Die Fehlerbehandlung wird im nachfolgenden Kapitel erläutert.

Bei der Initialisierung wird zunächst Speicher für die Verwaltungsstruktur allokiert:

```
struct app_loadmod_data_t{
    struct {
        unsigned block : 1;
        unsigned kernel : 1;
        unsigned allocated : 1;
    } flags;
    uint8_t error;
    uint32_t pos;
    uint32_t buffered;
    uint8_t * data;
    uint32_t totalsize;
    uint16_t page;
};
```

In den Flags werden verschiedene Zustände festgehalten. In `kernel` wird gesichert, ob es sich bei den Daten um einen Kernel oder ein Modul handelt. Des Weiteren besteht die Möglichkeiten, die Daten seitenweise auf den Flash zu schreiben oder erst im Arbeitsspeicher zwischenspeichern und anschließend als Block zu schreiben (`block`). Die Variable `allocated` wird auf eins gesetzt, nachdem der Flash für das Schreiben des Moduls vorbereitet wurde.

Die an die Modulschnittstelle übergebenen Daten werden zunächst in einem Puffer zwischengespeichert. Der Puffer ist mindestens so groß, dass die Einheitsvektoren, sowie zwei Header gespeichert werden können. Dies ist die Mindestspeichermenge, welche zum Speichern eines Kernels benötigt wird. Soll im Block geschrieben werden, wird der Puffer gegebenenfalls nach der Analyse des Headers so vergrößert, dass der gesamte Block hineinpasst. Die Integerwerte `pos`, `buffered` und `totalsize` geben Auskunft darüber, bis zu welcher Position geschrieben wurde, wie viele Daten im Speicher liegen und wie groß der zu schreibende Block insgesamt ist. `page` speichert die Seite, an welcher der Block anfängt.

Tritt während des Speichervorgangs ein Fehler auf, so wird die entsprechende Fehlernummer in `error` gespeichert, welche mit der Funktion `app_loadmod_geterror()` abgerufen werden kann.

Der Funktion `app_loadmod_data` können die Daten blockweise übergeben werden. Die übergebenen Daten werden zunächst in den vorhandenen Puffer kopiert. Wurden ausreichend Daten übergeben um einen Block-Header zu enthalten, wird zunächst überprüft, ob es sich um ein Modul, welches mit einem Header oder einen Kernel, welcher mit den Interruptvektoren beginnt, handelt. Hierzu wird analysiert, ob jedes zweite Wort den Wert `0x0C94` hat. Dies entspricht dem `jmp`-Befehl, aus welchem jedes zweite Wort der Interrupttabelle besteht. Handelt es sich bei den Daten um einen Kernel, so wird das entsprechende Flag in der Verwaltungsstruktur gesetzt. Je nachdem, ob es sich um einen Kernel oder ein Modul handelt, muss der Header unterschiedlich analysiert und behandelt werden.

Handelt es sich um ein Modul, wird zunächst eine Plausibilitätsüberprüfung des Headers durchgeführt. Hierbei wird zum Beispiel überprüft, ob die Endadresse hinter der Startseite oder `entryfn` innerhalb des Moduls liegt. Nachdem sichergestellt ist, dass es sich um einen Header handelt, wird anhand der Prüfsumme überprüft, ob das Modul für den installierten Kernel gelinkt wurde. Trifft auch dies zu, werden die Startseite sowie die Größe des Blocks in die Verwaltungsstruktur kopiert.

Handelt es sich bei den Daten um einen Kernel, so ist die Behandlung aufwändiger. Auch beim Kernel wird, wie bei einem Modul, zunächst der Header auf Plausibilität überprüft. Die Binärdaten

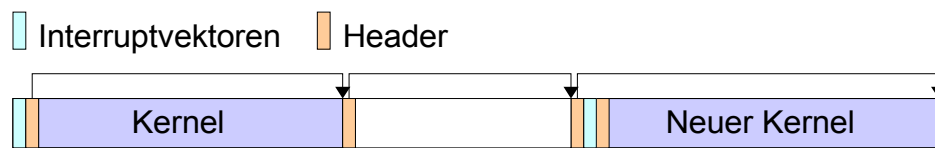


Abbildung 5.5: Puffern eines Kernels

des Kernels beginnen nicht mit einem Header, sondern mit den Interruptvektoren. Der neue Kernel muss jedoch im Flash zwischengespeichert werden und daher auf Grund der Vorgaben der Speicherverwaltung mit einem Header anfangen (vgl. Abb. 5.5). Um Platz für diesen Header zu schaffen, werden die bisherigen Daten im Speicher nach hinten verschoben. Aus diesem Grund muss zur Verarbeitung des Kernels Speicher für die Speicherung zweier Header allokiert werden.

Ist auch die Größe des Kernels aus dem Header extrahiert, wird nach freiem Speicher auf dem Flash gesucht. Hierbei wird die *last fit* Methode verwendet, um den Kernel möglichst weit an das Ende des freien Speicherbereichs zu schreiben. Hierdurch soll vermieden werden, dass, wenn ein Kernel durch einen größeren Kernel ersetzt wird, der neue Kernel beim Kopieren mit sich selber überschrieben werden muss (vgl. Abb. 5.6). Dies ist zwar in sofern unproblematisch, da der Kernel von vorne nach hinten kopiert werden kann, und der Anfang des Kernels bereits kopiert ist, wenn

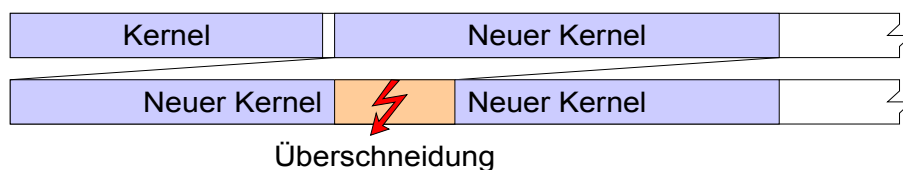


Abbildung 5.6: Überschneidung beim Kopieren eines Kernels

er überschrieben wird, jedoch ist es schwierig den Kopiervorgang fortzusetzen, falls es, zum Beispiel durch einen Spannungsausfall, zu einer Unterbrechung des Kopiervorgangs gekommen ist. Nachdem auch die zukünftige Position des Kernels bekannt ist, wird der Header des Speicherblocks generiert.

Nachdem alle benötigten Daten in die Verwaltungsstruktur übernommen wurden, werden die Binärdaten von Kernel und Modul wieder gleich behandelt. Sollen die Daten schon während der Übertragung in den Flash geschrieben werden, werden nun die Verwaltungsstrukturen vor und hinter dem Block, in welchen das Modul geschrieben werden soll, angelegt. Abbildung 5.7 zeigt dies exemplarisch. Zunächst wird die Länge des Blocks, in welches das Modul geschrieben werden soll, gekürzt und anschließend der Header des Blocks, welcher sich hinter dem Modul befindet, geschrieben. Anschließend wird der Block, welcher das Modul enthält, geschrieben. Überschneiden sich Beginn oder Ende des Moduls mit dem freien Block, braucht der entsprechende Schritt nicht ausgeführt werden. Anschließend wird `allocated` in der Verwaltungsstruktur auf eins gesetzt. Ist der Puffer mit einer Seite gefüllt, wird diese auf den Flashspeicher geschrieben und der Puffer freigegeben.

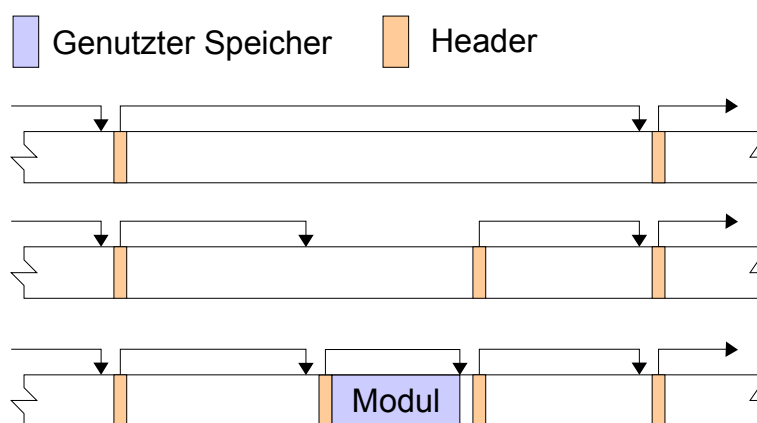


Abbildung 5.7: Allokation von Speicher im Flash

Sollen die Daten im Block auf den Flash geschrieben werden, wird der Puffer so vergrößert, dass alle Daten gespeichert werden.

Die Datenübertragung wird mit `app_loadmod_finish()` abgeschlossen. Wurde der Speicher noch nicht für das Schreiben der Daten vorbereitet, werden nun die im vorherigen Absatz beschriebenen Schritte durchgeführt. Anschließend werden alle im Puffer verbliebenen Daten geschrieben. Verließ der Schreibvorgang fehlerfrei, werden der Puffer und die Verwaltungsstruktur wieder freigegeben.

5.3.8 Fehlerbehandlung der Modulschnittstelle

Da es vorgesehen ist, den Knoten über Funk zu reprogrammieren, ist es meist schwer möglich einen Debugger anzuschließen oder umfangreiche Fehlermeldungen ausgeben zu lassen. Aus diesem Grund wurde Wert auf eine recht detaillierte Fehlerrückmeldung gelegt. Alle Funktionen liefern, wie bereits erwähnt, im Erfolgsfall 1 und im Fehlerfall 0 zurück. Der Fehler wird in der Verwaltungsstruktur abgelegt und kann mit dem Befehl `app_loadmod_geterror()` abgerufen werden. Nach einem Fehler muss der Speicher durch einen expliziten Aufruf der Funktion `app_loadmod_finish()` wieder freigegeben werden.

Während des Speicherns der Module werden folgende Fehler abgefangen:

- `APP_LM_ERR_NOMEM`: Nicht genügend Speicher um den Puffer zu initialisieren.
- `APP_LM_ERR_TOO_MUCH`: Es wurden mehr Daten übergeben als im Header angegeben.
- `APP_LM_ERR_HEADER`: Die Plausibilitätsüberprüfung des Headers ist fehlgeschlagen.
- `APP_LM_ERR_NO_FREE_PAGE`: Nicht ausreichend freier Flash, um den Kernel zu speichern.
- `APP_LM_ERR_NOREALLOC`: Es konnte nicht ausreichend Speicher allokiert werden, um alle Daten zu speichern.

- APP_LM_ERR_FLASH: Der Versuch, die Daten in den Flash zu schreiben, ist drei mal gescheitert.
- APP_LM_ERR_WRONG_DATALEN: Die Funktion `app_loadmod_finish()` wurde aufgerufen, obwohl nicht ausreichend Daten übergeben wurden.
- APP_LM_ERR_WRONG_KERNEL: Das Modul wurde für einen anderen Kernel gelinkt.
- APP_LM_ERR_ALLOC: Die Allokation des Flash ist gescheitert. Möglicherweise wurden bereits Daten in den vom Modul benötigten Bereich geschrieben.
- APP_LM_ERR_NOMEM_STRUCT: Es wurde kein Speicher für die Verwaltungsstruktur allokiert. Entweder wurde `app_loadmod_init()` nicht aufgerufen oder die Allokation ist fehlgeschlagen.

5.4 Starten und Stoppen von in Modulen enthaltenen Applikationen

Ist eine Applikation über die Modulschnittstelle auf einen Knoten geladen worden, kann diese gestartet und auch wieder gestoppt werden. Damit dies effizient geschieht, mussten einige Anpassungen an der Nut/OS Threadverwaltung vorgenommen werden. In diesem Kapitel werden zunächst die Nut/OS Threadverwaltung und anschließend die vorgenommenen Modifikationen erklärt.

5.4.1 Aufbau von Nut/OS Threads

Wie auch bei vielen anderen Funktionen von Nut/OS wird auch bei Threads vom dynamischem Speicher Gebrauch gemacht. Hierdurch wird die Anzahl der gleichzeitig laufenden Threads nur durch den Speicher begrenzt. Auch kann der gleiche Code mehrmals instantiiert werden. Über einen Zeiger ist es möglich Parameter an den Thread zu übergeben. Des Weiteren werden 244 verschiedene Prioritäten und das Warten auf Ereignisse und Timer unterstützt.

Der Scheduler arbeitet mit einer einfachen verketteten Liste. In dieser werden die Threads nach Priorität aufgelistet. Es wird immer der Thread mit der höchsten Priorität ausgeführt. Gibt es mehrere Threads mit der gleichen Priorität, so werden diese reihum ausgeführt. Threads mit niedriger Priorität werden daher nie ausgeführt, wenn ein Thread mit höherer Priorität nie die CPU abgibt, indem er auf ein Ereignis, wie einen Timer, wartet. Warten Threads auf ein Ereignis, werden sie aus der Hauptwarteliste ausgehängt und in die Liste des Ereignisses eingereiht. Tritt das Ereignis ein, werden die Threads wieder in die normale Hauptwarteliste eingehängt.

Threads werden in einer Struktur verwaltet, welche dynamisch allokiert wird:

```
struct _NUTTHREADINFO{
    NUTTHREADINFO *td_next;
    NUTTHREADINFO *td_qnxt;
    volatile u_int td_qpec;
    char td_name[9];
    u_char td_state;
    uptr_t td_sp;
    u_char td_priority;
    u_char *td_memory;
    HANDLE td_timer;
    volatile HANDLE td_queue;
};
```

Die Parameter sind:

- `td_next`: Zeigt auf das nächste Element in der verketteten Liste aller Threads.
- `td_qnext`: Die Threads werden vom Scheduler als verkettete Listen verwaltet. `qnxt` zeigt auf das nächste Element in der Liste.
- `td_qpec`: Zähler für die Anzahl der Ereignisse, welche an einen Thread gemeldet wurden.
- `td_name`: Der Name des Threads
- `td_state`: Speichert den Zustand des Threads. bereit, laufend, schlafend, beendet
- `td_sp`: Der Stackpointer des Threads.
- `td_priority`: Die Priorität des Threads
- `td_memory`: Zeiger auf den Anfang des für den Thread allokierten Speicherbereichs.
- `td_timer`: Zeigt auf den Timer, falls der Thread auf diesen wartet.
- `td_queue`: Zeigt auf die Wurzel der Schlange, in welcher der Thread gerade wartet.

5.4.2 Starten und Stoppen von Applikationen

Zum Starten und Stoppen von Applikationen wurden die Funktionen `app_start()` und `app_stop()` implementiert. Diesen wird der Name der zu startenden oder stoppenden Applikation übergeben. Die Funktion `app_start()` sucht anhand des Namens nach einem gleichnamigen Modul. Aus dem Header des Moduls werden die Einsprungadresse (`entryfn`) und die Stackgröße ausgelesen und zusammen mit dem Namen an die Nut/OS Systemfunktion `NutThreadCreate()` übergeben, welche die Applikation als Thread startet.

Um die Applikation zu beenden, mussten ein paar Anpassungen gemacht werden, da Nut/OS keine Signalisierung eines speziellen Threads unterstützt. Gegebenenfalls hätte für jeden Thread eine eigene Message Queue angelegt werden müssen. Aus diesem Grund wurde die Struktur `_NUTTHREADINFO` um folgende Struktur erweitert:

```
volatile struct{
    unsigned shutdown:1;
}signal;
```

Diese Erweiterung ermöglicht die binäre Signalisierung von Threads. Bis jetzt wurde das Signal `shutdown` implementiert, welches den Thread anweist sich zu beenden. Weitere Signale können mit geringem Aufwand hinzugefügt werden. Threads können mit der Funktion `app_signal()` signalisiert werden. Das Makro `app_stop()` sendet das `shutdown` Signal an den entsprechenden Thread. Es liegt jedoch in der Verantwortung des Entwicklers, in regelmäßigen Abständen mit `app_getShutdown()` zu überprüfen, ob das `shutdown` Signal empfangen wurde.

In Nut/OS kann sich ein Thread nur selbst beenden. Es ist daher nicht möglich das Beenden des Threads, zum Beispiel wenn das `shutdown` Signal ignoriert wird, zu erzwingen. Aus diesem Grund wurde ein `app_kill()` Befehl implementiert. Dieser setzt die Priorität des Threads auf 255, welches ihn terminiert.

5.4.3 Verhinderung von Speicherlecks

Das Terminieren von Threads ist in sofern nicht unproblematisch, da allozierter Speicher den Threads nicht zugeordnet wird. Dies bedeutet, dass allein der Thread für die Freigabe des Speichers verantwortlich ist. Wird ein Thread terminiert, kann dieser seinen Speicher nicht mehr freigeben und mangels Verwaltungsstrukturen ist es auch nicht mehr möglich herauszufinden, welchen Speicher der Thread verwendet hat.

Um Speicher bei der Termination eines Threads freizugeben, muss der Speicher dem Thread eindeutig zugeordnet sein. Dies wurde mit einer verketteten Liste realisiert. Ein Zeiger auf diese Liste wird in der Threadverwaltungsstruktur gespeichert. Die Verwaltungsstruktur von allokiertem Speicher, welche standardmäßig nur die Größe des Elements speichert, wurde der Freispeicherverwaltung angeglichen. In ihr kann nun auch ein Zeiger auf einen weiteren allokierten Speicherblock gespeichert werden.

Wird Speicher allokiert, so wird dieser an den Anfang der verketteten Liste des Threads eingereiht. Dies ist sowohl beim Allokieren die effizienteste Methode, da die Liste nicht durchsucht werden muss, als auch bei Freigaben, da auf Grund der verschachtelten Struktur von Funktionen der zuletzt eingereihte Speicherblock häufig zuerst wieder freigegeben wird.

Wird ein Thread beendet, so wird überprüft, ob der von dem Thread verwendete Speicher vollständig freigegeben wurde. Ist dies nicht der Fall, werden die in der Liste eingereihten Speicherblöcke freigegeben.

Problematisch ist dieses Verhalten, wenn Daten in allokiertem Speicher von einem Thread an einen anderen übergeben werden. Wird der erste Thread beendet, so wird auch der zugehörige Speicher freigegeben, ohne dass der zweite Thread hierüber informiert wird. Der zweite Thread arbeitet nun gegebenenfalls auf Speicher, welcher nicht mehr verwendet werden darf. Sollte ein Szenario wie dieses implementiert werden, so müssen Funktionen zur Übereignung von Speicher von einem Thread an einen Anderen implementiert werden.

Alternativ kann mit der Funktion `NutHeapAllocNT()` Speicher allokiert werden, welcher keinem Thread zugeordnet ist. Bei diesem Speicher besteht jedoch wieder die Gefahr eines Speicherlecks.

5.5 Einschränkungen der Implementierung

Das Laden von Modulen zur Laufzeit unterliegt verschiedenen Einschränkungen. Einige davon sind systembedingt, andere können durch zusätzlichen Implementierungsaufwand möglicherweise behoben werden. Die meisten Einschränkungen wurden bereits beschrieben, werden aber an dieser Stelle nochmals kurz zusammengefasst.

- Die Echtzeitfähigkeit des Systems ist durch das Schreiben auf den Flash eingeschränkt. Die CPU ist während des Schreibvorgangs für ca. 4 ms blockiert. Echtzeitanforderungen können während dieser Zeit nicht erfüllt werden.
- Bei der Erstellung von Modulen wird aktuell die Verwendung der `.data` und `.bss` Sektionen nicht unterstützt. Dies hat zur Folge, dass weder statische lokale, noch globale Variablen verwendet werden können.
- Es ist momentan nicht möglich bei eingeschalteten Interrupts auf den Flashspeicher oberhalb der 64 KiB zuzugreifen. Die genaue Ursache ist unbekannt, hängt aber höchstwahrscheinlich damit zusammen, dass das RAMZ Register überschrieben wird.
- Soll ein Modul gelöscht werden, wird momentan nicht überprüft, ob in dem Modul laufender Code aktuell ausgeführt wird. Es liegt daher in der Verantwortung des Anwenders, dies auszuschließen.

5.6 Nut/OS Quellcodemodifikationen zur Unterstützung der binären Threadsignalisierung

Um einzelne Threads, wie zuvor vorgestellt, zu signalisieren, mussten einige Anpassungen am Nut/OS Betriebssystem vorgenommen werden. Im Folgenden werden diese dokumentiert.

5.6.1 include/sys/heap.h

In die Struktur `_NUTTHREADINFO` wurde eine Union zur Signalbehandlung eingefügt.

```
volatile union{
    uint8_t signals;
    struct{
        unsigned shutdown:1;
    }signal;
};
```

Diese wird genutzt, um das an einen Thread gesendete Signal zu speichern. Eine Union wird verwendet, um auf alle Signale gleichzeitig zuzugreifen. Bis jetzt wurde lediglich das Signal `shutdown` implementiert. Damit Signale auch von einem Interrupt an den Thread geschickt werden können, muss die Union volatile sein.

5.6.2 arch/avr/os/context_gcc.c

Die zu der Struktur `_NUTTHREADINFO` hinzugefügte Union muss mit 0 initialisiert werden. Daher wurde der Funktion `NutThreadCreate()` die Zeile

```
td->signals = 0;
```

hinzugefügt.

Die Thread-Struktur wird in Nut/OS architekturabhängig initialisiert. In der aktuellen Entwicklungsumgebung wird die GCC Toolchain verwendet, um Code für die AVR-Architektur zu compilieren. Soll der Code mit einem anderen Compiler oder Architektur verwendet werden, muss die entsprechende Version von `NutThreadCreate()` angepasst werden. Ein entsprechender Patch, um die Thread-Struktur plattformunabhängig zu initialisieren, wurde eingereicht und wird möglicherweise in zukünftigen Versionen von Nut/OS enthalten sein.

5.7 Nut/OS Quellcodemodifikationen um Speicher einem Thread zuzuordnen.

Auch bei der Implementierung der Zuordnung von Speicher zu Threads mussten einige Anpassungen an Nut/OS Quellcode vorgenommen werden. Bei diesen besteht die Chance, dass sie in das Nut/OS Repository aufgenommen werden. Aus diesem Grund wurde die Möglichkeit geschaffen, die Unterstützung mit dem Makro `NUTMEM_THREAD` ein- und auszuschalten.

5.7.1 include/sys/heap.h

```
typedef struct _UHEAPNODE{
    size_t hn_size;
#ifdef NUTMEM_THREAD
    UHEAPNODE *hn_next;
#endif
} UHEAPNODE;
```

Die Struktur `UHEAPNODE` wurde neu eingeführt, um die statische Verwendung von `size_t` für den Header zu ersetzen. Die Verwendung einer Struktur ermöglicht eine einfache Erweiterung um neue Funktionen. Ist das Makro `NUTMEM_THREAD` definiert, wird der Header um einen Zeiger erweitert, mit welchem der von einem Thread allokierte Speicher verkettet wird.

5.7.2 include/sys/thread.h

```
#ifndef NUTMEM_THREAD
#include <sys/heap.h>
#endif
```

Die Headerdatei `heap.h` musste in die `thread.h` eingebunden werden, damit die `UHEAPNODE` Struktur definiert ist.

In der `_NUTTHREADINFO` Struktur wurde ein Zeiger vom Typ `UHEAPNODE` hinzugefügt:

```
#ifndef NUTMEM_THREAD
    UHEAPNODE * td_heap;
#endif
```

Der Zeiger zeigt auf der erste Element der verketteten Liste des durch den Thread allokierten Speichers.

5.7.3 arch/avr/os/context_gcc.c

In der Funktion `NutThreadCreate()` muss der in der `thread.h` hinzugefügte Zeiger mit `NULL` initialisiert werden:

```
#ifndef NUTMEM_THREAD
    td->td_heap = NULL;
#endif
```

5.7.4 os/heap.c

In der `heap.c` mussten mehrere Anpassungen gemacht werden. Die Speicherverwaltung arbeitet intern mit Zeigern auf den Header. Übergeben werden aber stets Zeiger auf den Speicher hinter dem Header. Beispielsweise muss in `NutHeapFree()` der übergebene Zeiger in einen Zeiger auf den Header umgewandelt werden. Dies geschieht mit folgenden Code:

```
fnode = (HEAPNODE *) (((size_t *) block) - 1);
```

Um flexibel auf Änderungen im Header von belegtem Speicher reagieren zu können, musste dies durch `UHEAPNODE` ersetzt werden:

```
fnode = (HEAPNODE *) (((UHEAPNODE *) block) - 1);
```

Diese Änderung musste des Weiteren bei der Definition von `MEMOVHD` und den Funktionen `NutHeapAdd()` und `NutHeapAlloc()` vorgenommen werden.

Um den Speicher einem Thread zuzuordnen zu können, muss die Thread-Struktur sowie der Zeiger auf den aktuell laufenden Thread, `runningThread`, bekannt sein. Aus diesem Grund wurde die Datei `thread.h` eingebunden.

```
#ifndef NUTMEM_THREAD
#include <sys/thread.h>
#endif
```

Bevor in der Funktion `NutHeapAlloc()` der freie Speicherbereich zurückgegeben wird, muss dieser dem laufenden Thread zugeordnet werden, in dem er vorne in die verkettete Liste des zugeordneten Speichers eingereiht wird:

```
#ifndef NUTMEM_THREAD
    if(runningThread != NULL){
        ((UHEAPNODE *) fit)->hn_next = runningThread->td_heap;
        runningThread->td_heap = ((UHEAPNODE *) fit);
    }
#endif
```

Wird `NutHeapFree()` aufgerufen, so muss der Speicher aus der verketteten Liste wieder entfernt werden.

```
#ifndef NUTMEM_THREAD
//Remove from thread;
    if(runningThread != NULL){
        UHEAPNODE **tnode;
        tnode = &(runningThread->td_heap);
        while(*tnode != NULL){
            if((uptr_t)(*tnode) == (uptr_t)fnode){
                *tnode = (*tnode)->hn_next;
                break;
            }
            tnode = &((*tnode)->hn_next);
        }
    }
#endif
```

6 Named Memory zur Datenwiederherstellung

Nach dem Austausch eines Kernels oder einem Reset, zum Beispiel durch das Auslösen des Watchdog Timers, wird das auf dem Mikrocontroller laufende System neu initialisiert. Hierbei gehen im Normalfall alle im Arbeitsspeicher gespeicherten Daten verloren. Werden diese auch nach dem Neustart benötigt, müssen sie in nicht flüchtigem Speicher wie dem EEPROM oder Flash gespeichert werden. Zu den Statusdaten gehören neben dem Betriebszustand häufig auch Zähler und Tokens. Diese können sich je nach Anwendung alle paar ms ändern.

Speichertyp	Lebensdauer (typisch)	Lebensdauer bei einem Schreibzyklus je Sekunde
Flash	10 000 Schreibzyklen	2,7 Stunden
EEPROM	100 000 Schreibzyklen	1,2 Tage
SRAM	∞	∞

Tabelle 6.1: Vergleich der Speichersysteme

In Tabelle 6.1 ist deutlich zu erkennen ist, dass Flash oder EEPROMs sich nicht zur Speicherung sich häufig ändernder Daten eignen. Wird die gleiche Speicherzelle sekundlich beschrieben, ist ihre Lebensdauer bereits nach einigen Stunden oder Tagen erschöpft. Eine Möglichkeit diesem Problem zu begegnen ist das so genannte *wear leveling*. Hierbei wird bei jedem Schreibvorgang eine andere Speicherzelle verwendet. Der Nachteil diese Systems ist, dass ein zusätzlicher Verwaltungsaufwand benötigt wird, um sicher zu stellen, dass beim Lesen auf die zuletzt geschriebene Speicherzelle zugegriffen wird. Des Weiteren ist die Lebensdauer des Speichers trotz *wear leveling* immer noch begrenzt.



Abbildung 6.1: SRAM mit Anschlüssen zum "Aufsetzen" einer Batterie

Die einzige Möglichkeit Daten, welche sich häufig ändern, zu speichern ist daher SRAM. Hierbei handelt es sich um flüchtigen Speicher, das heißt, dass nach einem Spannungsausfall alle Daten verloren gehen. Auf Grund des geringen Stromverbrauchs von wenigen μA (typ. 10) können die Daten mit einem Superkondensator für mehrere Tage oder mit einer Batterie für mehrere Jahre erhalten werden. In solchen Fällen redet man auch von NVRAM (Non Volatile Random Access Memory). Hier gibt es fertige Lösungen, welche die Umschaltung auf die alternative Energiequelle, wie eine Batterie, in den Speicherbaustein integriert haben oder ein Aufklipsen einer Batterie ermöglichen (Abb. 6.1). Alternativ gibt es Lösungen, welche bei Spannungsverlust den Inhalt des SRAM automatisch in ein Flash speichern oder bereits mit einem wiederaufladbaren Akku versehen sind.

Problematisch ist der Zugriff auf die Daten nach einem Reset des Mikrocontrollers. Durch eine Kernelaktualisierung hat sich möglicherweise das Speicherlayout geändert und die Variablen werden jetzt an anderen Stellen als zuvor gespeichert. Auch ist es im Normalfall nicht vorgesehen, dass im Datenspeicher Daten stehen, welche verwendet werden sollen. Es muss daher ein System entwickelt werden, welches in der Lage ist Daten nach einem Neustart des Mikrocontrollers aus dem SRAM wieder verfügbar zu machen.

6.1 Anforderungen

Es soll eine Lösung gefunden werden, welche es ermöglicht Daten im SRAM zu speichern und auf diese nach einem Neustart des Systems wieder zuzugreifen. Hierbei müssen verschiedene Kriterien erfüllt werden:

- Die Implementierung muss einfach zu verwenden sein. Es müssen dem Entwickler Funktionen zur Verfügung gestellt werden, welche eine einfache Handhabung ermöglichen, ohne dass Wissen über die zugrunde liegende Speicherverwaltung benötigt wird.
- Die Größe des Speicherblocks muss zur Laufzeit gewählt werden können.
- Es muss auch nach der Ersetzung des Kernels möglich sein, die Daten wiederherzustellen. Die Lösung kann also keine Strukturen verwenden, welche sich im `.data` – Bereich des Kernels befinden.

6.2 Lösungskonzept

Zur Erfüllung der Anforderungen wird eine Lösung implementiert, welche es ermöglicht Speicher unter einem bestimmten Namen zu allokalieren und zu einem späteren Zeitpunkt unter Verwendung dieses Namen wieder auf die Daten zuzugreifen. Dieser Speicher wird im folgenden *Named Memory* genannt. *Named Memory* ist mit *Shared Memory* zu vergleichen. Während beim *Shared Memory* normalerweise eine ID-Nr zur Identifizierung der Speicherbereiche verwendet wird, wird beim *Named Memory* eine Zeichenkette verwendet.

Speicherverwaltung wird auch der deutlich größere externe Arbeitsspeicher verwendet. Es kann davon ausgegangen werden, dass sich die Elemente des *Named Memory* im externen Arbeitsspeicher befinden und es daher keine Überschneidungen mit dem, vom Bootloader verwendeten, Arbeitsspeicher gibt.

6.3 Umsetzung

Der *Named Memory* wurde in Nut/OS implementiert, da das Konzept vom BNode Sensorknoten unabhängig ist. Um den Einfluss auf den Nut/OS Kernel so gering wie möglich zu halten, wurden alle Funktionen in einer externen Bibliothek implementiert. Dennoch mussten einige Anpassungen an der Speicherverwaltung vorgenommen werden:

Zum einen wurde die Funktion `NutHeapAdd()`, welche die Speicherverwaltung initialisiert so angepasst, dass diese die Initialisierungsfunktion des *Named Memory* aufruft. Zum anderen wurde `NutHeapAlloc()` so erweitert, dass Speicherblöcke nicht nur nach *best fit*, sondern auch nach *last fit*, allokiert werden können. Hierzu wurde die Funktion `NutHeapAlloc()` in `NutHeapAllocAlg()` umbenannt, an welche der zu verwendende Algorithmus als Parameter übergeben werden kann. Die neue `NutHeapAlloc()` Funktion ruft nun `NutHeapAllocAlg()` mit *best fit* als den zu verwendendem Algorithmus auf.

Die Länge des Namens wurde auf acht Zeichen begrenzt. Die Prüfsumme wird über den gesamten Header berechnet. Für die Berechnung der Prüfsumme wurde der Einfachheit halber ein *CRC8* Algorithmus verwendet.

Wie bereits in Kapitel 5.7.1 erwähnt, wurde der Header für allokierten Speicher von `size_t` in eine Struktur umgewandelt:

```
typedef struct _UHEAPNODE{
    size_t hn_size;
} UHEAPNODE;
```

Um die Handhabung zu vereinfachen, wurde diese in den Header der *Named Memory* Verwaltungsstruktur integriert:

```
struct nmem_node_t{
    UHEAPNODE hn;
    struct nmem_node_t * next;
    char name[8];
    size_t allsize;
    uint8_t crc8;
};
```

Das Strukturelement `next` zeigt auf das nächste Element der verketteten Liste des *Named Memory*. In `name` wird der Name des *Named Memory* gespeichert. Da die Länge auf acht Zeichen begrenzt ist, muss bei der Verwendung der entsprechenden Funktionen kein Byte für die Nulltermination reserviert werden. In `allsize` wird die allokierte Größe gespeichert. Dies ist notwendig,

da auf Grund der von der Speicherverwaltung verwendeten Algorithmen nicht von der in `UHEAPNODE` angegebene Größe `hn_size` auf die ursprünglich angeforderte Datenmenge zurück geschlossen werden kann (Siehe auch nachfolgendes Kapitel).

6.3.1 Funktionsweise der Nut/OS Speicherverwaltung

Nut/OS macht intensiven Gebrauch von der dynamischen Speicherverwaltung. Wie auf anderen Systemen auch ist es möglich Speicher mit `malloc()` zu allokiere und mit `free()` wieder freizugeben. Die freien Bereiche werden mit einer linear verketteten Liste verwaltet. Jeder freie Bereich enthält eine Struktur mit der Größe des freien Bereichs und einen Zeiger auf den nächsten freien Bereich.

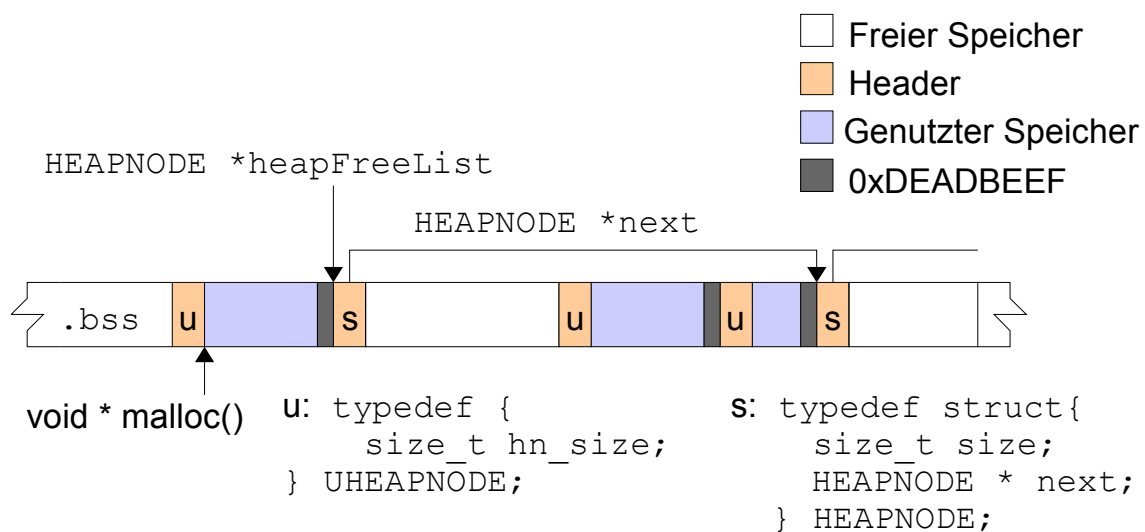


Abbildung 6.3: Nut/OS Speicherverwaltung

Abbildung 6.3 zeigt einen Ausschnitt des durch Nut/OS verwalteten Speichers. Auf der linken Seite endet die `.bss` Sektion. Der Speicher hinter der `.bss` Sektion wird von der Speicherverwaltung verwaltet. Direkt daran angeschlossen befindet sich im Beispiel ein allokiertes Bereich.

Der freie Speicher wird in einer verketteten Liste gespeichert. Die Variable `heapFreeList` zeigt auf den ersten freien Bereich. In diesem ist eine Struktur gespeichert, welche nicht nur die Größe des Bereichs, sondern auch einen Zeiger auf den nächsten freien Bereich speichert.

Freier Speicher wird nach der *best fit* Methode allokiert. Dies bedeutet, dass aus der Liste der kleinste freie Speicherbereich gewählt wird, welcher groß genug ist, um die Daten aufzunehmen. Abbildung 6.4 zeigt einen Ausschnitt des Speichers. (A) zeigt die Ausgangssituation. Der freie Speicherbereich in der Mitte repräsentiert den nach der *best fit* Methode gefundenen Speicherblock. Überschreitet der freie Speicher hinter dem zu allokierten Bereich einen bestimmten Schwellwert, so wird der Speicher wieder in die Freispeicherverwaltung eingegliedert (B). Wird der Schwellwert unterschritten, so wird der nicht benötigte Speicher dem zu allokierten Block zugeordnet (C). Der Schwellwert muss mindestens so groß wie die Freispeicherverwaltungsstruktur

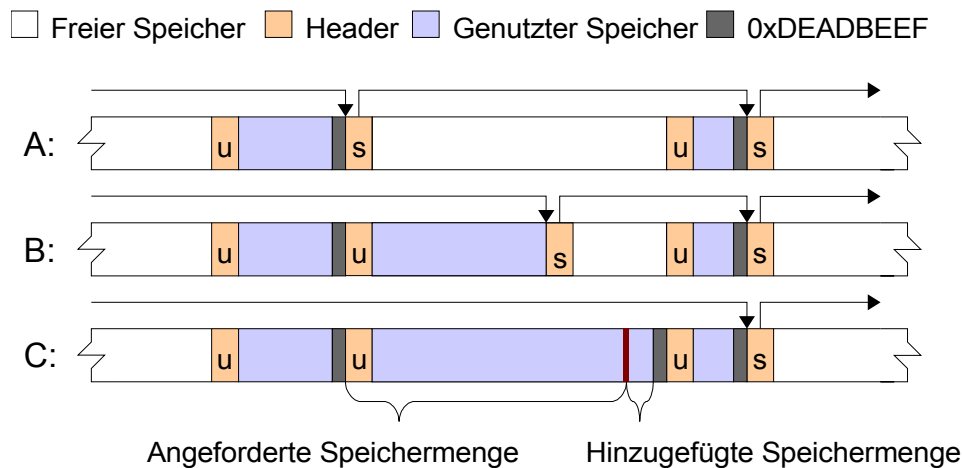


Abbildung 6.4: Allokation von Speicher

sein, da diese sonst nicht im freien Speicherbereich abgelegt werden kann. Des Weiteren macht es Sinn, den Schwellwert einige Byte groß zu konfigurieren, da sonst auf Dauer sehr viele kleine Speicherbereiche entstehen. Diese werden selten benötigt, verlangsamen jedoch das Durchsuchen der Liste nach einem geeigneten Speicherbereich erheblich.

Die Größe des allokierten Bereichs wird im Header des Bereichs gespeichert (Abb. 6.3, u). Nut/OS verwendet hierzu den Datentyp `size_t`. Die Größe des Speicherblocks muss gespeichert werden, da beim Freigeben des Speichers nur ein Zeiger übergeben wird, die Größe zum Einreihen in die Freispeicherverwaltung jedoch bekannt sein muss. Da mehrere allokierte Bereiche aufeinander folgen können, ist es auch nicht möglich die Größe eines allokierten Bereichs über die verketteten Liste der freien Bereiche zu bestimmen.

Der Aufruf von `malloc()` liefert die Adresse hinter dem Header zurück, so dass der Header und somit auch die tatsächlich reservierte Speichermenge für den Programmierer transparent bleibt. Hinter dem reservierten Bereich wird der Wert `0xDEADBEEF` geschrieben. Dieser Wert ist willkürlich und kann beim Freigeben des Speichers überprüft werden. Dies ermöglicht die Erkennung von Pufferüberläufen, welche leicht entstehen, wenn zum Beispiel bei der Allokation einer Zeichenkette vergessen wird Speicher für den `NULL`-Char zu reservieren. Der für den Header und den Wert `0xDEADBEEF` benötigte Speicher muss bei der Suche nach einem geeigneten Speicherblock selbstverständlich beachtet werden.

Bei der Freigabe des Speichers wird zunächst überprüft, ob sich vor oder hinter dem freizugebenden Bereich ein freier Bereich befindet. Ist dies der Fall, wird der freizugebende Bereich mit diesem zu einem großen, freien Bereich verschmolzen.

Pufferüberläufe sind bei dieser Art der Speicherverwaltung besonders problematisch. Wird über die `0xDEADBEEF`-Konstante hinaus geschrieben, werden zunächst Strukturen zur Verwaltung des Speichers überschrieben. Handelt es sich um freien Speicher, wird die verkettete Liste zur Freispeicherverwaltung zerstört. Ein Aufruf einer Speicherverwaltungsfunktion wird in diesem Fall entweder eine fehlerhafte Aktion ausführen, oder in einer Endlosschleife hängen bleiben.

Befindet sich hinter der `DEADBEEF`-Konstante allozierter Speicher, werden dessen Daten überschrieben. Das Verhalten ist nicht mehr vorhersehbar. Spätestens jedoch bei der Freigabe des überschriebenen Speicherbereichs wird die Speicherverwaltung zerstört.

Da der Zugriff auf externen Speicher langsamer ist als auf internen, bietet Nut/OS die Möglichkeit Speicher exklusiv für Stacks zu reservieren. Praktisch wird hierzu ein zweite Freispeicherstruktur instantiiert, welche für die Verwaltung des Stack-Speichers zuständig ist.

6.3.2 Initialisierung

Die Initialisierung des *Named Memory* wird durch die Funktion `NutNmemHeapAdd()` durchgeführt. Ist der *Named Memory* bei der Compilation des Nut/OS Systems aktiviert, so wird diese von der Funktion `NutHeapAdd()` aufgerufen. `NutNmemHeapAdd()` liefert im Erfolgsfall 1 zurück und `NutHeapAdd()` kehrt zurück. Wird 0 zurückgeliefert, wird der Speicher durch `NutHeapAdd()` initialisiert. Dies ist zum Beispiel der Fall, wenn spezielle Bereiche für Stack reserviert werden und dies von `NutNmemHeapAdd()` erkannt wurde.

```
void NutHeapAdd(void *addr, size_t size){
#ifdef NUTMEM_NAMED_MEM
    if (NutNmemHeapAdd(addr, size)) return;
#endif
    // Normale initialisierung
}
```

Als Parameter bekommt `NutHeapAdd()` und damit auch `NutNmemHeapAdd()` die Anfangsadresse und die Größe des für die Speicherverwaltung zur Verfügung gestellten Arbeitsspeichers übergeben. Zunächst wird überprüft, ob es sich um Speicher handelt, welcher für Stack verwendet werden soll, und gegebenenfalls 0 zurückgeliefert. Anschließend wird anhand der Prüfsumme überprüft, ob sich am Ende des Speicherbereichs ein gültiger *Named Memory*-Header befindet. Ist dies nicht der Fall, wird ein Header angelegt und der sich davor befindende Speicher freigegeben.

Befindet sich am Ende des übergebenen Speichers ein gültiger *Named Memory* Header, so wird die verkettete Liste überprüft. Dies ist wichtig, da nicht ausgeschlossen werden kann, dass ein Element des *Named Memory* überschrieben wurde. Kann ein Header nicht validiert werden, so wird die verkettete Liste am vorherigen Element terminiert. Dies bedeutet einen Datenverlust, der jedoch bei diesem Lösungskonzept nicht vermieden werden kann.

Anschließend wird der freie Speicher zwischen den einzelnen *Named Memory* Elementen freigegeben. Da es sich bei der verketteten Liste des *Named Memory* um eine unsortierte Liste handelt, muss stets die gesamte Liste durchsucht werden um das erste Element zu finden. Der Speicher vor

diesem Element wird nun freigegeben. Anschließend wird die Liste nach dem zweiten Element durchsucht, um den Speicher zwischen dem ersten und zweiten Element freizugeben. Dies wird wiederholt, bis die Liste abgearbeitet ist.

6.3.3 Named Memory verwalten

Zur Verwaltung des *Named Memory* stehen drei Funktionen zur Verfügung: `NutNmemCreate()`, `NutNmemGet()` und `NutNmemFree()`. Diese dienen dazu einen neuen Speicher zu erstellen, einen Zeiger auf einen vorhandenen *Named Memory* zu bekommen oder *Named Memory* freizugeben.

Soll ein neuer *Named Memory* erstellt werden (`NutNmemCreate()`), werden Name und Größe benötigt. Zunächst wird überprüft, ob bereits ein *Named Memory* mit diesem Namen existiert und gegebenenfalls `NULL` zurückgeliefert. Ist dies nicht der Fall, wird der benötigte Speicher von der Speicherverwaltung angefordert und der Header des *Named Memory* initialisiert. Anschließend wird ein Zeiger zurückgeliefert, welcher auf die Speicherstelle hinter dem Header zeigt, so dass dieser für den Anwender nicht sichtbar ist.

Zu einem späteren Zeitpunkt kann mit `NutNmemGet()` auf den zuvor erstellten Speicher zugegriffen werden. Als Parameter werden der Name des Speichers, sowie ein Zeiger auf eine Variable vom Typ `size_t` übergeben. Die verkettete Liste wird nun nach einem *Named Memory* mit dem entsprechenden Namen durchsucht. Ist dies erfolgreich, wird die Größe des Blocks in die übergebene Variable vom Typ `size_t` geschrieben und ein Zeiger, welcher hinter den Header zeigt, zurückgegeben. Andernfalls wird `NULL` zurückgeliefert.

Freigegeben wird der Speicher mit der Funktion `NutNmemFree()`. Es ist wichtig, dass *Named Memory* nicht mit `NutHeapFree()` freigegeben wird, da dies gegebenenfalls zu einer Korruption der Speicherverwaltung führen kann. Um den *Named Memory* freizugeben, wird der Block zunächst aus der verketteten Liste entfernt und der verwendete Speicher anschließend an die Speicherverwaltung zurückgegeben.

6.3.4 Einschränkungen

Obwohl die oben beschriebene Lösung auf den BTnodes funktioniert, kann es auf anderen Systeme zu Problemen kommen. Hierbei sind zwei Fälle besonders zu beachten:

Erstens haben die BTnodes einen großen externen Speicher. Da der externe Speicher explizit aktiviert werden muss, ist es nicht möglich, dass das Wurzelement durch den normalerweise sich am Ende des Speicherbereichs befindlichen Stack bei der Initialisierung oder durch den Bootloader überschrieben wird. Bei Systemen ohne externen Speicher muss gegebenenfalls die Endadresse des Stacks angepasst werden. Hierbei reicht es jedoch nicht diese vor das Wurzelement zu setzen, da sonst andere *Named Memory* Elemente, welche sich am Ende des Speicherbereichs befinden, überschrieben werden. Der Stackpointer muss daher an einer möglichst niedrigen Speicheradresse

initialisiert werden. Auf Grund der Größe des externen Speichers des BTreeNode ist es unwahrscheinlich, dass Named Memory Elemente im internen Speicher allokiert werden und somit überschrieben werden können.

Zweitens unterstützt Nut/OS die Verwendung von nicht zusammenhängenden Speicherbereichen. Im Moment wird das Wurzelement des *Named Memory* an das Ende des ersten Speicherbereichs gesetzt, welches der Speicherverwaltung übergeben wird. Die einzelnen Elemente des *Named Memory* können sich aber über die verschiedenen Speicherbereiche verteilen.

6.4 Nut/OS Codemodifikationen

Um den Eingriff in Nut/OS möglichst gering zu halten, wurden die meisten Funktionen in eine eigene Quellcodedatei geschrieben. Da der *Named Memory* teilweise stark in die Speicherverwaltung eingreift, sind Modifikationen am Quelltext unabdingbar. Die Modifikationen betreffen zwei Funktionen: Zum einen die `NutHeapAdd()`, welche die Speicherverwaltung initialisiert, und zum anderen `NutHeapAlloc()` um optional nach der *last fit* Methode Speicher allokiert zu können. Wie auch schon bei der Möglichkeit Speicher Threads zuzuordnen, kann die Unterstützung des *Named Memory* mit dem Makro `NUTMEM_NAMED_MEM` ein- und ausgeschaltet werden.

6.4.1 os/heap.c

Da die Heap und *Named Memory* Funktionen sehr eng verknüpft sind, hat sich die direkte Einbindung der `namedmem.c` in die `heap.c` als die einfachste Lösung herausgestellt. Dies muss unterhalb der Definitionen von `setBeef()` und `checkBeef()` geschehen, da diese beiden Funktionen in der `namedmem.c` benötigt werden:

```
#ifndef NUTMEM_NAMED_MEM
#include "namedmem.c"
#endif
```

Um die Auswirkung auf `NutHeapAdd()` zu minimieren, wurde die Speicherinitialisierung in die Funktion `NutMmemHeapAdd()` ausgelagert. Hat diese den Speicher erfolgreich initialisiert, liefert sie den Wert 1 zurück. Andernfalls, zum Beispiel wenn der Speicher für Stack initialisiert wird, liefert sie 0 zurück und der Speicher wird „normal“ initialisiert.

```
void NutHeapAdd(void *addr, size_t size)
{
#ifdef NUTMEM_NAMED_MEM
    if(NutMmemHeapAdd(addr, size)) return;
#endif    //ifndef NUTMEM_NAMED_MEM
    *((uptr_t *) addr) = size;
    setBeef((HEAPNODE *)addr);
    NutHeapFree((UHEAPNODE *) addr + 1);
}
```

Die Auswirkung auf `NutHeapAlloc()` sind etwas größer. Zunächst wurde `NutHeapAlloc()` in `NutHeapAllocAlg()` umbenannt und um den Parameter `algo` erweitert. Ist dieser 0 wird *best fit*, bei 1 *last fit* verwendet.

```
void *NutHeapAllocAlg(size_t size, uint8_t algo)
```

In der ersten Schleife, in der die verkettete Liste nach einem ausreichend großen freien Block durchsucht wird, wird bei jedem Treffer der zu verwendende Algorithmus abgefragt. Im Gegensatz zum *best fit*, der eine bereits gefundene Speicherstelle nur verwirft, wenn die aktuelle kleiner ist als die alte, wird beim *last fit* nach der letzten passenden Speicherstelle gesucht.

```
    if (node->hn_size >= size) {
        if(algo == 0){ //best fit
            [...]
        } else { //last fit
            fit = node;
            fpp = npp;
        }
    }
}
```

Ist der gefundene Knoten größer als der benötigte Speicher, so wird der Bereich am Anfang des Speichers verwendet. Beim *last fit* muss der Speicher am Ende des Blocks verwendet werden. Aus diesem Grund musste der Code auch hier nochmal angepasst werden.

```
    if (fit->hn_size > size + sizeof(HEAPNODE) + ALLOC_THRESHOLD) {
        if(algo == 0){
            [...]
        } else {
            node = fit;
            fit = (HEAPNODE *)((uptr_t)fit + node->hn_size - size);
            node->hn_size -= size;
            fit->hn_size = size;
        }
    }
}
```

Um `NutHeapAlloc()` wie bisher aufrufen zu können, wurde eine entsprechende Funktion hinzugefügt.

```
void *NutHeapAlloc(size_t size){
    return NutHeapAllocAlg(size, 0);
}
```

6.4.2 include/sys/heap.h

Um die Indirektion durch den zusätzlichen `NutHeapAlloc()` zu minimieren, wurde im Header eine Inline-Funktion hinzugefügt.

```
extern inline void * NutHeapAlloc(size_t size){
    return NutHeapAllocAlg(size, 0);
};
```

Das Schlüsselwort `inline` weist den Compiler an, die Funktion nach Möglichkeit in die aufrufende Funktion einzubetten. Wird der Code ohne Optimierung compiliert, wird das Schlüsselwort `inline` ignoriert. Da es sich um eine Funktion und nicht um einen Prototypen handelt, bedeutet dies, dass die Funktion in jede Objektdatei, deren Quellcode auf die Headerdatei verweist, eingebunden wird. Hierdurch tritt beim Linken ein Fehler auf, da die Funktion mehrfach implementiert ist.

Aus diesem Grund werden Inline-Funktionen meist als `static` deklariert. Wird keine Optimierung verwendet so wird die Funktion auch in jede Objektdatei, in welcher die Funktion aufgerufen wird, eingebunden, jedoch ist die Funktion nach außen nicht sichtbar. Um dies zu vermeiden, wurde die Funktion `extern` deklariert. Dies bedeutet jedoch auch, dass es eine Objektdatei geben muss, welche diese Funktion nach außen sichtbar enthält. Dies ist in der Datei `nut/os/heap.c` der Fall.

7 Anwendung der implementierten Funktionen

Dieses Kapitel erklärt die Anwendung der implementierten Funktionen. Es soll einem Entwickler helfen die Funktionen zu nutzen, ohne sich mit den Hintergründen auseinandersetzen zu müssen. Aus diesem Grund liegt der Schwerpunkt nicht auf der Implementierung der Funktionen, sondern deren praktischen Anwendung. Einige in den vorherigen Kapiteln besprochenen Themen werden an dieser Stelle nochmals erklärt.

7.1 Einrichtung des Entwicklungssystems

Im folgenden Text, sowie in den Make-Dateien, wird davon ausgegangen, dass die Entwicklungsumgebung, BTnut und Nut/OS in einem Verzeichnis gespeichert sind. Hierbei liegt die Entwicklungsumgebung im Verzeichnis `work`, BTnut in dem Verzeichnis `btnut` und Nut/OS in dem Ordner `nut`. Der Name der Verzeichnisses, welches die Entwicklungsumgebung enthält, kann beliebig benannt werden.

Bei dem Entwurf der Verzeichnisstruktur für die Entwicklungsumgebung wurde sich an Nut/OS orientiert. Sie enthält folgende Ordner:

- `app`: Enthält die verschiedenen Applikationen, welche auf den Knoten nachgeladen werden können.
 - `app_print`: Beispielanwendung, welche die Modulunterstützung demonstriert.
 - `memtest`: Beispielanwendung, welche die dynamische Speicherverwaltung testet.
 - `nmem_counter`: Beispielanwendung, welche die Funktionsweise des *Named Memory* demonstriert.
- `host`: Enthält Komponenten, welche auf dem Host, also dem Rechner auf welchem kompiliert wird, benötigt werden.
 - `bt_reprog`: Enthält die Hostkomponenten für die Reprogrammierung von Sensorknoten über Bluetooth.
- `kernel`: Enthält verschiedene Kernel, sowie zu deren Erstellung benötigte Bibliotheken.
 - `bootloader`: Enthält den Bootloader.
 - `btrep`: Ein Beispielkernel zum Reprogrammieren über Bluetooth.
 - `include`: Enthält die für die Erstellung von Bootloader, Kernen und Applikationen benötigten Header-Dateien.
 - `lib`: Enthält eine Bibliothek von Funktionen, welche zu Erstellung von Bootloadern, Kernen und Applikationen benötigt werden.

- `serial`: Enthält einen Testkernel, welcher über die serielle Schnittstelle reprogrammierbar ist.
- `kernels`: In dieses Verzeichnis werden die Linkerskripte der erstellten Kernel kopiert.
- `nut`: Enthält die in Nut/OS modifizierten Dateien. Die Verzeichnisstruktur entspricht der von Nut/OS.

Nachdem Nut/OS, BTnut und die Entwicklungsumgebung ausgecheckt wurden, muss zunächst Nut/OS angepasst werden, indem die im Verzeichnis `work/nut` enthaltenen Dateien nach `nut` kopiert werden.

Soll *Named Memory* oder die Zuordnung von allokiertem Speicher zu einem Thread verwendet werden, müssen Anpassungen an der Datei `btnut/Makedefs` gemacht werden. Um den *Named Memory* zu aktivieren muss die Zeile

```
CPFLAGS.COMMON += -DNUTMEM_NAMED_MEM
```

hinzugefügt werden; für die Aktivierung der Zuordnung von allokiertem Speicher zu einem Thread die Zeile

```
CPFLAGS.COMMON += -DNUTMEM_THREAD
```

Anschließend kann BTnut kompiliert werden. Hierzu muss im Verzeichnis `btnut` `make install` aufgerufen werden. Durch den Aufruf von `make install` im Verzeichnis `work/kernel` werden zunächst die für die Modulunterstützung benötigten Bibliotheken und anschließend der Bootloader erstellt.

Soll der Mikrocontroller mit dem Beispiel-Bluetooth-Kernel verwendet werden, muss auch noch die Host-Komponente im Verzeichnis `work/host/bt_reprog` kompiliert werden. Hierzu reicht ein Aufruf von `make` im Verzeichnis `work/host`.

7.2 Einbindung der Modulunterstützung in einen Kernel

Bevor ein Modul auf den Knoten geladen werden kann, muss zunächst ein Kernel entwickelt werden, welcher Module unterstützt. Um diesen Prozess zu vereinfachen, wurde eine Bibliothek erstellt, welche die benötigten Funktionen zur Verfügung stellt. Die Funktionen sind im Quellcode ausführlich dokumentiert. Daher wird an dieser Stelle nur eine Übersicht über deren Verwendung gegeben.

Das Laden eines Moduls findet in mehreren Schritten statt. Zunächst wird vom Host die Prüfsumme des Kernels erfragt. Mit dieser kann das Modul kompiliert und dessen Größe bestimmt werden. Anschließend wird im Flashspeicher nach einem ausreichend großen Speicherbereich gesucht. Nachdem ein freier Adressbereich zurück an den Host übermittelt wurde, kann das Modul an die entsprechende Adresse gelinkt werden und anschließend zum Knoten übertragen werden. Es ist nun möglich, die in dem Modul enthaltene Applikation zu starten. Der Vorgang der Modulerstellung wurde in Abbildung 7.1 als Sequenzdiagramm dargestellt.

Um den Vorgang zu vereinfachen, wurde eine Bibliothek erstellt, welche die zur Erstellung und Laden von Modulen sowie zum Starten und Stoppen von Applikationen benötigten Funktionen zur Verfügung stellt.

Die Prüfsumme des Kernels kann mit der Funktion `mod_kernelCRC()` erfragt werden. Ist die Größe des Moduls bekannt, kann eine geeignete Speicheradresse mittels `app_loadmod_requestMem()` erfragt werden.

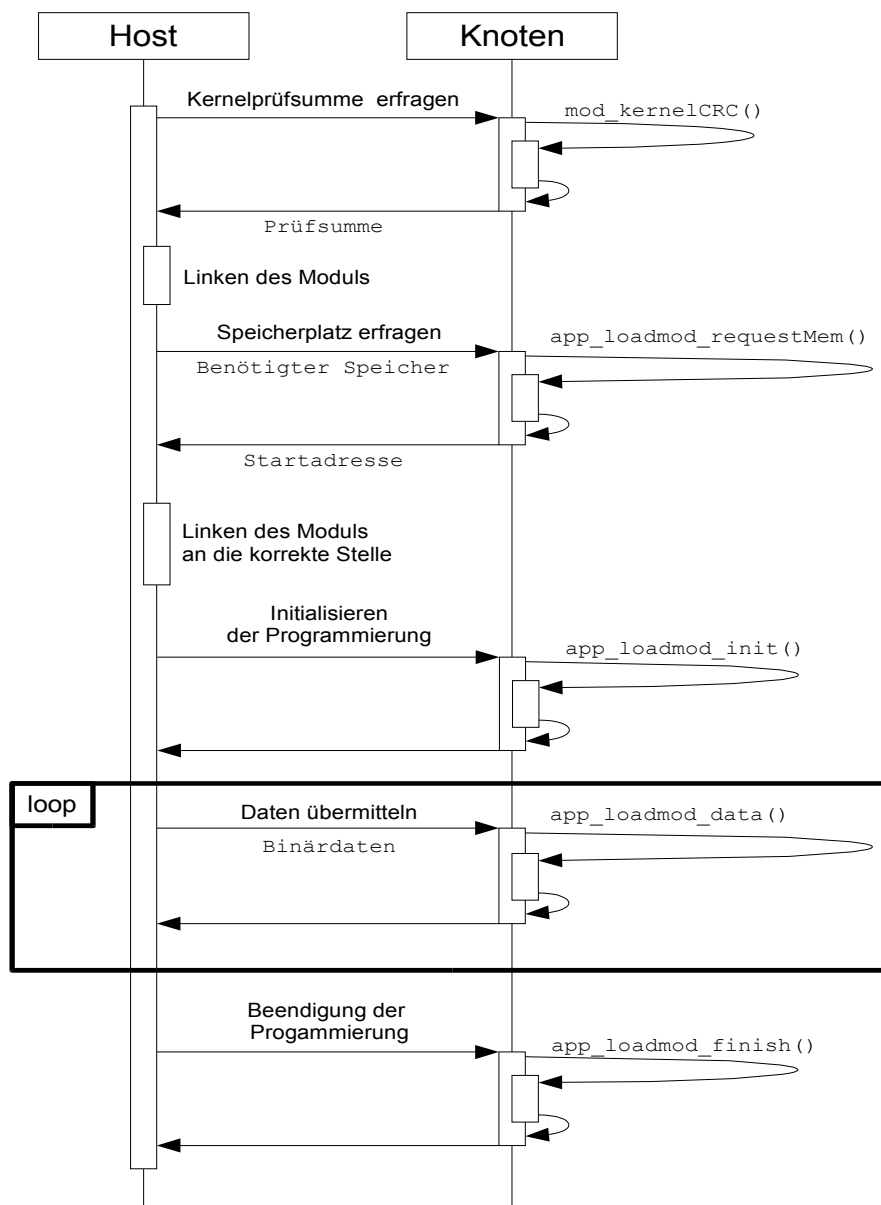


Abbildung 7.1: Ablauf der Programmierung eines Moduls

Nachdem das Modul erstellt wurde, muss dieses auf den Knoten übertragen und im Flash gespeichert werden. Hierzu stehen drei Funktionen zur Verfügung: `app_loadmod_init()`, `app_loadmod_data()` und `app_loadmod_finish()`.

Bevor die Daten in den Flash geschrieben werden können, müssen die benötigten Strukturen und Pufferspeicher mit `app_loadmod_init()` initialisiert werden. Anschließend können die Binärdaten an `app_loadmod_data()` übergeben werden. Alle zum Speichern nötigen Informationen werden automatisch aus den Binärdaten extrahiert. Ist die Datenübertragung abgeschlossen, muss der Vorgang mit `app_loadmod_finish()` beendet werden. Ist der Schreibvorgang erfolgreich, wird der Speicher wieder freigegeben. Das Modul ist nun auf dem Flash gespeichert und die enthaltene Applikation kann gestartet werden.

Tritt während des Speichervorgang ein Fehler auf, liefern die Funktionen 0 zurück. Der genaue Fehler kann anschließend mit `app_loadmod_geterror()` erfragt werden. Anschließend muss der Speicher mit `app_loadmod_cleanup()` wieder freigegeben werden.

Die Module können mit dem Befehl `app_loadmod_del()` wieder gelöscht werden.

Neben den Funktionen, um Module zu laden, muss der Kernel eine Möglichkeit zum Starten der im Modul enthaltenen Applikationen bereitstellen. Hierzu stehen die Funktionen `app_start()` und `app_stop()` zur Verfügung. Diesen muss lediglich der Name der zu startenden oder stoppenden Applikation übergeben werden. Mit der Funktion `app_kill()` ist es möglich einen Thread zu terminieren. Hierbei wird gegebenenfalls vom Thread allozierter Speicher nicht freigegeben (siehe auch Kap. 7.7).

Zur Erstellung des Kernels können die Makefiles der bereits implementierten Kernel verwendet werden. In diesen muss die Variable `PROJ` und eventuell `SRCS`, falls weitere Quellcodedateien verwendet werden, angepasst werden. Das Makefile generiert automatisch einen Vor-Kernel um die Prüfsumme zu berechnen und anschließend den Kernel mit der richtigen Prüfsumme. Außerdem wird die Symboltabelle für den Kernel in das Verzeichnis `kernels` exportiert. Nach dem Aufruf von `make` liegt der Kernel im ELF-, Binär- und im Intel-Hex-Format vor.

Soll ein Kernel erstellt werden, in welchem der Bootloader bereits integriert ist, kann `make boot` aufgerufen werden. Hierdurch wird der Bootloader mit in den Kernel gelinkt. Der Bootloader wird in Form von Binärdaten eingebunden. Es werden daher keine Debugsymbole erstellt (vgl. Kap. 5.3.5). Die Integration des Bootloaders ist nur sinnvoll, wenn der Mikrocontroller über einen ISP oder den Jtag reprogrammiert wird. Andernfalls schlägt der Programmiervorgang fehl, da die Modulschnittstelle mehr Daten erhält als sie erwartet und den Programmiervorgang abbricht.

Um den Kernel auf den Knoten zu übertragen, kann dieser genauso wie ein bereits als Binärdaten vorliegendes Modul zum Knoten übertragen werden. Die `app_loadmod_data()` erkennt automatisch, dass es sich um einen Kernel handelt, und allokiert den benötigten Speicher.

7.3 Beispielimplementierung eines Kernels mit Bluetoothunterstützung

Im Folgenden wird die Implementierung eines Kernels mit Bluetoothunterstützung vorgestellt. Mit diesem ist es möglich über Bluetooth Module auf den Knoten zu laden und die darin enthaltenen Applikation auszuführen. Auf Grund der Größe der Bluetoothbibliothek wird der Kernel größer als 60 KiB. Damit ist es nicht mehr möglich den Kernel zu aktualisieren, da keine zwei Versionen des Kernels im Flash des Knoten gespeichert werden können.

Für die Kommunikation über Bluetooth wurde die verbindungsorientierte *L2CAP* Verbindung gewählt. Diese ist sowohl auf dem Knoten als auch dem Host einfach zu implementieren. Nachdem eine Verbindung zwischen den beiden Endpunkten aufgebaut wurde, können Daten über diesen „Channel“ übertragen werden.

Auf dem BTnode werden die Daten paketorientiert verarbeitet. Aus diesem Grund wurde ein einfaches Protokoll entworfen, welches am Anfang jedes Datenpaketes ein Kommando in Form eines Char-Wertes hat. Der weitere Inhalt des Paketes hängt vom jeweiligen Kommando ab. Variablen werden im Little-Endian-Format übertragen. In Tabelle 7.1 wird eine Übersicht über die implementierten Kommandos gegeben.

Name	Wert	Länge	Daten
BTCMD_ACK	11	1	
			Quittiert einen Aufruf als erfolgreich.
BTCMD_ERROR	6	2	uint8_t error
			Meldet einen Fehler; error enthält den aufgetretenen Fehler.
BTCMD_LDMODERROR	7	4	uint8_t error, uint8_t loadmoderror
			Meldet einen Fehler beim Aufruf einer loadmod-Funktion; error enthält die Fehlerposition; loadmoderror den von app_loadmod_geterror() zurückgelieferten Fehler.
BTCMD_GETCRC	0	1	
			Ruft mod_kernelCRC() auf.
BTCMD_SNDCRC	1	5	uint32_t crc32
			Liefert den Rückgabewert von mod_kernelCRC().
BTCMD_ALLOC	2	13	char name[8]; uint32_t size
			Ruft app_loadmod_requestMem(name, size, 0) auf
BTCMD_ALLOCATED	3	3	uint16_t page
			Liefert den Rückgabewert von app_loadmod_requestMem().
BTCMD_INIT	8	2	uint8_t block
			Ruft app_loadmod_init(block) auf. Auf diesen Befehl wird im Erfolgsfall mit BTCMD_ACK geantwortet.

Name	Wert	Länge	Daten
BTCMD_DATA	4	> 5	uint16_t len; uint16_t seq; uint8_t[len - 5]
			Ruft <code>app_loadmod_data(data, len - 5)</code> auf. <code>len</code> gibt die Länge des Gesamtpaketes an. <code>seq</code> enthält die Sequenznummer des Datenpaketes. Nach dem Aufruf von <code>BTCMD_INIT</code> muss diese mit 0 beginnen und mit jedem Paket hochgezählt werden. Auf diesen Befehl wird im Erfolgsfall mit <code>BTCMD_ACK</code> geantwortet. Wenn ein unerwarteter Wert für <code>seq</code> empfangen wird, antwortet der Knoten mit <code>BTCMD_RESEND</code> .
BTCMD_RESEND	10	3	uint16_t seq
			Fordert den Host auf, die Datenübertragung bei dem Paket mit der Sequenznummer <code>seq</code> erneut zu beginnen.
BTCMD_FIN	9	1	
			Ruft <code>app_loadmod_finish()</code> auf.
BTCMD_RST	5	1	
			Ruft <code>app_loadmod_cleanup()</code> auf.
BTCMD_LSTAPP	12	1	
			Ruft <code>mod_getNext()</code> auf und liefert die Namen der installierten Applikationen mit der <code>BTCMD_APP</code> -Nachricht zurück bis keine Applikationen mehr verfügbar sind.
BTCMD_APP	13	9	char name[8]
			Enthält den Namen einer installierten Applikation. Ist <code>name[0]</code> NULL, sind keine weiteren Applikationen mehr vorhanden.
BTCMD_START	14	9	char name[8]
			Startet die Applikation mit dem Namen <code>name</code>
BTCMD_STOP	15	9	char name[8]
			Stoppt die Applikation mit dem Namen <code>name</code>

Tabelle 7.1: Kommandos des Protokolls zur Kommunikation mit dem Bluetooth Beispielkernel

Auf der Hostseite wurde eine Kommandozeilenapplikation implementiert, mit welcher es möglich ist Module auf den Knoten zu laden, die installierten Applikationen aufzulisten, zu starten und wieder zu beenden. Um das Kompilieren von Modulen zu optimieren, wurde bei den zum Erstellen eines Moduls benötigten Befehlen die Möglichkeit geschaffen, den Rückgabewert in eine Datei zu schreiben.

Der Aufruf hat folgende Syntax:

```
bt_reprog <mac-Adresse> <Kommando> [Parameter]
```

Bei Erfolg liefert `bt_reprog` 0, im Fehlerfall 1 zurück. Es wurden folgende Kommandos implementiert:

- `crc [file]`
Erfragt die CRC32 Prüfsumme des Kernels. Ist `file` angegeben, wird die Prüfsumme als Hexadezimalzahl in die Datei geschrieben.
- `alloc name size [file]`
Erfragt die erste Seite, an welcher das Modul mit dem Namen `name` und der Größe `size` gespeichert werden kann. Ist `file` angegeben, wird die Seite als Hexadezimalzahl in die angegebene Datei geschrieben.
- `upload file`
Lädt die in `file` enthaltenen Binärdaten auf den Knoten.
- `list`
Listet die auf dem Knoten installierten Applikationen auf.
- `start name`
Startet die Applikation mit dem Namen `name`.
- `stop name`
Stoppt die Applikation mit dem Namen `name`.

7.4 Erstellung und Laden einer Applikation

Bei der Entwicklung der Modulunterstützung wurde versucht möglichst viel zu abstrahieren, um die Verwendung so einfach wie möglich zu gestalten. Ein Modul besteht aus einer C-Datei, welche eine Applikationsfunktion enthält.

Das folgende Codebeispiel zeigt eine Applikation, welche nach dem Start alle 125 ms ein U auf die Standardausgabe ausgibt. Bekommt sie den Befehl sich zu beenden, gibt sie vor dem Beenden ein E aus.

```
#include <stdio.h>
#include <sys/timer.h>
#include <sys/thread.h>
#include <application.h>

APPLICATION("PrintU", 192, arg){
    for (;;) {
        if(app_gotShutdown()) {
            putchar('E');
            NutThreadExit();
        }
        putchar('U');
        NutSleep(125);
    }
}
```

Um eine Applikation zu erstellen, muss die `application.h` eingebunden werden. Diese stellt das Makro `APPLICATION` zur Verfügung, mit welchem die Applikation erstellt werden kann. Es kann maximal eine Applikation je Modul erstellt werden. Die Parameter entsprechen den des `THREAD` Makros von Nut/OS. Wichtig ist, dass regelmäßig auf `app_gotShutdown()` geprüft wird. Gibt `app_gotShutdown()` 1 zurück, muss die Applikation beendet werden.

Im Makefile müssen für ein neues Projekt die Parameter `PROJ`, `SRCS`, `KERNEL_CRC`, `TEXTSECTION` angepasst werden. `PROJ` gibt den Projektnamen an und `SRCS` listet die Quellcode-dateien auf. Der Parameter `KERNEL_CRC` enthält die Prüfsumme des Kernels, für welche gelinkt werden soll. In die Variable `TEXTSECTION` wird die Adresse geschrieben, an welche kompiliert werden soll. Beide Informationen müssen vom Kernel erfragt werden. Danach kann die Binärdatei an die entsprechende Stelle des Flash geschrieben werden.

Für den Bluetooth Kernel wurde die Abfrage der Kernelprüfsumme und eines freien Speicherbereichs sowie die Übertragung zum Knoten in das Makefile integriert. Das Modul wird mit dem Befehl `make btinstall <adresse>` kompiliert, gelinkt und auf den Knoten übertragen. Der Platzhalter `adresse` muss die MAC-Adresse des Knotens als Hex-Zahl enthalten.

Bei der Erstellung von Modulen muss darauf geachtet werden, dass das Ablegen von Daten weder in der `.data` noch der `.bss` Sektion unterstützt wird. Dies bedeutet, dass weder globale, noch lokale statische Variablen unterstützt werden. Auch Zeichenketten in Anführungszeichen werden in der `.data` Sektion gespeichert. Sie müssen zur Laufzeit in den Speicher geladen werden. Die AVR-Libc stellt für diese Funktionalität das `PSTR`-Makro zur Verfügung. Von den Zeichenketten-Funktionen gibt es jeweils eine Version mit einem `_P`-Suffix, welche mit dem `PSTR`-Makro verwendet werden können.

Das folgende Beispiel demonstriert, wie dem auf dem Stack liegenden `char`-Array mit Hilfe des `PSTR`-Makros der Zeichenkette `Hello World!` zugewiesen wird.

```
char str[13];
strcpy_P(str, PSTR("HELLO WORLD!"));
```

Wird ein Modul oberhalb von 64 KiB gespeichert, funktioniert dies nicht mehr, da zur Adressierung der Zeichenkette im Flash ein 17 Bit Zeiger benötigt wird. Es ist jedoch geplant entsprechende Funktionen in die AVR-LibC zu integrieren. In die Modulbibliothek wurde daher eine entsprechende Bibliothek integriert, welche das Laden von Zeichenketten oberhalb der 64 KiB Grenze erlaubt:

```
char str[13];
strcpy_PF(str, PFSTR("HELLO WORLD!"));
```

Die durch die Bibliothek zur Verfügung gestellten Funktionen können sich jedoch fehlerhaft verhalten, wenn während des Ladens ein Interrupt auftritt [14] und wurden noch nicht ausgiebig getestet.

7.5 Echtzeitproblematiken bei der Verwendung von Modulen

Während des Schreibens von Daten auf den Flash kann weder Programmcode ausgeführt noch können Interrupts behandelt werden. Da das Schreiben des Flash ca. 4 ms dauert, kann dies zu Echtzeitproblemen kommen. Dies betrifft alle Timer, welche kürzere Zykluszeiten als 4 ms haben. Dies muss auch bei sämtlichen verwendeten Schnittstellen beachtet werden. Wird eine serielle Schnittstelle mit mehr als 300 Baud betrieben, muss sichergestellt werden, dass während des Schreibens auf den Flash keine weiteren Daten übertragen werden. Wird eine Flusskontrolle verwendet, muss darauf geachtet werden, dass bei einer Software-Flusskontrolle sichergestellt wird, dass das entsprechende Kommando zur Unterbrechung des Datenflusses gesendet wurde, bevor der Schreibvorgang gestartet wird.

Wird die serielle Schnittstelle als `stdout` und Hardwareflusskontrolle verwendet, kann das Senden der Gegenseite mit den Befehlen

```
u_long rxstatus;  
rxstatus = UART_RXDISABLED;  
ioctl(_fileno(stdout), UART_SETSTATUS, &rxstatus);
```

unterbrochen werden. Beim nächsten Befehl, welcher von der Schnittstelle liest, wird das Senden der Gegenseite automatisch wieder aktiviert. Für die Softwareflusskontrolle mit Xon/Xoff muss in Nut/OS zunächst eine Möglichkeit geschaffen werden, das Senden des Xoff Kommandos zu erzwingen.

7.6 Verwendung des Named Memory

Named Memory erlaubt es Speicher unter einem bestimmten Namen zu allozieren. Unter Verwendung des Namens kann zu einem späteren Zeitpunkt wieder auf die Daten zugegriffen werden. Dies schließt den Austausch der Firmware oder einen Reset ein.

Named Memory wurde in Nut/OS integriert. Es kann auch ohne Modulunterstützung verwendet werden. Daher ist es auch möglich die im Verzeichnis `nut` enthaltenen Modifikationen in Nut/OS vorzunehmen und anschließend in der BTnut Entwicklungsumgebung zu arbeiten.

Um BTnut mit *Named Memory* zu compilieren, muss in der Datei `btnut/Makedefs` die Zeile

```
CPFLAGS.COMMON += -DNUTMEM_NAMED_MEM
```

hinzugefügt werden. Hierdurch wird der Compiler angewiesen das Makro `NUTMEM_NAMED_MEM` zu definieren. Anschließend muss zumindest der von Nut/OS integrierte Teil von BTnut durch die Ausführung von `make install` im Ordner `btnut/btnode/nut` neu compiliert werden. Der *Named Memory* ist nun in den Bibliotheken enthalten.

Folgende Beispielapplikation zeigt die Verwendung von *Named Memory*:

```
APPLICATION("PrintI", 192, arg)
{
    int * pi;
    char name[3];
    size_t size;
    strcpy_P(name, PSTR("ct"));

    pi = NutNmemGet(name, &size);
    if(pi != NULL && size != sizeof(int) ){
        NutNmemFree(i);
        pi = NULL;
    }

    if(pi == NULL){
        pi = NutNmemCreate(name, sizeof(int) );
        if(pi == NULL){
            NutThreadExit();
        }
        *pi = 0;
    }

    for (;;) {
        if(app_gotShutdown() == 1){
            putchar('E');
            NutThreadExit();
        }
        printf_P(PSTR("%i\n"), *pi);
        (*pi)++;
        NutSleep(125);
    }
}
```

Applikationen unterstützen keine globalen Variablen und somit auch keine mit Anführungszeichen definierten Zeichenketten. Sie mit können dem `PSTR()` Makro im Programmspeicher abgelegt werden (siehe auch Kap. 7.4). In dem Beispiel wird der Name des *Named Memory* „ct“ mit `strcpy_P()` in den Datenspeicher geladen. Anschließend wird mit `NutNmemGet()` versucht auf den *Named Memory* zuzugreifen. Ist dies erfolgreich, wird ein Zeiger auf den Speicherbereich zurückgeliefert und die Größe des Speicherbereichs in die Variable `size` geschrieben. Hat der Speicher nicht die Größe eines Integers, wird der Speicher mit `NutNmemFree()` freigegeben. Auf in dem Speicherbereich enthaltene Daten kann nicht mehr zugegriffen werden.

Konnte kein Speicher mit dem Namen „ct“ gefunden werden, oder wurde dieser zuvor freigegeben, wird mit `NutNmemCreate()` ein neuer Speicher mit dem Namen „ct“ und der Größe eines Integers erstellt und der zurückgelieferte Speicher mit 0 initialisiert.

Anschließend wird der Zähler alle 125 ms hochgezählt und der Wert ausgegeben. Wird die Applikation beendet und neu gestartet, wird der Zähler an der Stelle fortgesetzt, an der er unterbrochen wurde. Dies ist unabhängig davon, ob der Mikrocontroller zwischenzeitlich neu gestartet, ein neuer Kernel, oder eine neue Version der Applikation, die beispielsweise den Zähler als Hex ausgibt, aufgespielt wurde.

7.7 Verwendung der Zuordnung des allokierten Speichers zu Threads

Ein hohes Risiko bei der Verwendung einer Speicherverwaltung sind Speicherlecks. Diese entstehen, wenn Speicher allokiert wird, aber nicht wieder freigegeben wird. Da Nut/OS zwar Threads, jedoch keine Prozesse unterstützt, wird allokiertes Speicher auch beim Beenden eines Threads nicht wieder freigegeben. Dieses Problem wird verstärkt, wenn ein Thread terminiert werden muss, da in einem solchen Fall auch ohne Programmierfehler der Speicher nicht wieder freigegeben werden kann. Um dieses Problem zu vermeiden, wurde eine Lösung implementiert, welche bei jeder Allokation den Speicher dem allozierenden Thread zuordnet. Wird der Thread beendet, wird automatisch aller vom Thread allokiertes, aber noch nicht wieder freigegebener Speicher freigegeben.

Bis jetzt gibt es noch keine Funktionen, welche eine Übereignung des Speichers von einem Thread an einen anderen ermöglichen. Wird daher Speicher von einem Thread allokiert und dann an einen zweiten Thread übergeben, so muss sichergestellt werden, dass der erste Thread nicht beendet wird solange der zweite Thread mit dem Speicher arbeitet.

Um die Zuordnung des allokierten Speichers zu Threads zu aktivieren, muss in die Datei `btnut/Makedefs` die Zeile

```
CPFLAGS.COMMON += -DNUTMEM_THREAD
```

eingefügt werden. Anschließend muss zumindest der von Nut/OS integrierte Teil von BTnut durch die Ausführung von `make install` im Ordner `btnut/btnode/nut` neu compiliert werden.

Wird die Zuordnung des Speichers zu Threads verwendet, muss sichergestellt werden, dass Speicher, welcher nach dem Beenden eines Threads weiter verwendet werden soll, mit dem Befehl `NutHeapAllocNT()` allokiert wird.

8 Zusammenfassung, Diskussion und Ausblick

Im Rahmen dieser Arbeit wurde eine Modulunterstützung für Nut/OS auf BTnode Sensorknoten implementiert. Es wurde gezeigt, dass es trotz einiger Einschränkungen möglich ist Module zur Laufzeit auf Sensorknoten zu übertragen und dort auszuführen. Die Implementierung der benötigten Funktionen in Form einer Bibliothek ermöglicht eine schnelle Implementierung der Modulunterstützung unter Verwendung unterschiedlicher Protokolle und Übertragungsmedien. Dies wurde Anhand eines Beispielkernels gezeigt, welcher Bluetooth sowie ein einfaches, selbst spezifiziertes Protokoll verwendet.

Des Weiteren wurde mit *Named Memory* eine Lösung aufgezeigt, sich häufig ändernde Statusdaten im SRAM zu sichern und nach einem Reset oder einer Aktualisierung des Programmcodes wieder auf diese Daten zuzugreifen. Die Lösung baut auf der dynamischen Speicherverwaltung auf, so dass keine speziellen Speicherbereiche reserviert werden müssen. Das Konzept ist leicht auf andere Mikrocontrollerbetriebssysteme übertragbar.

8.1 Diskussion

Die Aufgabenstellung wurde mit zwei sich ergänzenden, jedoch ansonsten voneinander unabhängigen, Lösungen bewältigt. Eine ganzheitliche Lösung, den Code zur Laufzeit auszutauschen und dabei Statusdaten transparent übernehmen zu können, wie sie in Kapitel 3.4 vorgestellt wird, ist auf jeden Fall attraktiver als die vorgestellte Lösung, jedoch im Bereich der Sensornetzwerke wegen der hohen Anforderungen nicht zufriedenstellend realisierbar.

Aus diesem Grund wurde auf die Verwendung einer Kombination aus Modulen und der Möglichkeit, Daten unter einem Namen im Arbeitsspeicher ablegen zu können, zurückgegriffen. Die Verwendung des *Named Memory* ist ein Kompromiss zwischen der Zielsetzung, die Daten transparent übernehmen zu können, und den beiden Alternativen, die Daten entweder zu verlieren, oder explizit auf einen nichtflüchtigen Speicher schreiben zu müssen.

Ist eine Aktualisierung des Gesamtsystems zur Laufzeit nicht möglich, ist die Verwendung von Modulen die logische Schlussfolgerung. Hierbei handelt es sich um die Fortführung eines bei leistungsfähigeren Systemen bewährten Konzepts in den Bereich der Sensornetzwerke: Die Verwendung eines Grundsystems und der Möglichkeit Applikationen nach Bedarf zu laden und auszuführen. Die einzige ernsthafte Alternative zu der Verwendung von Modulen sind *virtuelle Maschinen*, welche jedoch nur bei sehr häufigen Änderungen des Codes ihr Potential ausspielen können. Des Weiteren bietet sich die Verwendung von Modulen auf Grund der Architektur des Nut/OS Betriebssystems an, da zur Implementierung kaum Eingriffe in das System gemacht werden müssen.

Aber nicht nur die Möglichkeit Code zur Laufzeit nachzuladen spricht für die Verwendung von Modulen, sondern auch, dass bei einer Aktualisierung deutlich weniger Daten übertragen werden müssen, da der Kernel nicht ausgetauscht werden muss. Die *PrintI*-Beispielanwendung ist als

Modul 296 Byte, die *PrintU*-Beispielanwendung 72 Byte groß - der Kernel mit Bluetoothunterstützung ist über 60 KiB groß. Hierbei sind Module auch dem diff-basierten Ansatz überlegen, da diese nicht nur den neu hinzugekommenen Code enthalten müssen, sondern gegebenenfalls auch an anderer Stelle Anpassungen nötig sind, um die neu hinzugekommenen Funktionen aufrufen zu können. Des Weiteren ist es grundsätzlich möglich Module oder den Kernel auch mit dem diff-basierten Ansatz zu aktualisieren.

8.1.1 Implementierung der Modulunterstützung

Die Unterstützung von Modulen auf Mikrocontrollern ist keine neue Idee und wird bereits von LegOS, Contiki, SOS und einigen weniger bekannten Betriebssysteme unterstützt. Bei der vorgestellten Lösung wurde daher versucht die Stärken anderer Systeme zu übernehmen und gegebenenfalls Lösungen für die Schwächen zu finden.

Eine große Schwäche bei vielen anderen Systemen ist die zuverlässige Identifikation des Kernels, welche beim *Pre-Linking* von Modulen essentiell ist. Die Entwickler von Contiki haben deswegen einen Linker entwickelt, welcher auf dem Sensorknoten läuft. Gelöst wurde dieses Problem mit der Identifikation des Kernels durch eine *CRC32* Prüfsumme. Hierdurch ist es äußerst unwahrscheinlich, dass ein Kernel falsch identifiziert wird.

Die Möglichkeit *Jump Tables*, wie bei beispielsweise SOS, zu verwenden, wäre unbefriedigend gewesen, da hierdurch der Umfang der vom Kernel zur Verfügung gestellten Funktionen erheblich eingeschränkt hätte werden müssen. Die eingeschränkte Funktionsauswahl hätte zum einen die Entwicklung von Modulen verkompliziert, da nicht auf alle vom Kernel bereitgestellten Funktionen zugegriffen werden kann. Zum anderen hätte die Verwendung der *Jump Tables* die Module vergrößert, da bereits im Kernel enthaltene Funktionen nochmals in das Modul eingebunden hätten werden müssen.

Im Folgenden sollen nun die Schwachstellen der vorgestellten Implementierung sowie mögliche Lösungen diskutiert werden.

Statische Variablen

Die aktuell wohl größte Einschränkung bei der Verwendung von Modulen ist, dass weder globale noch statische lokale Variablen verwendet werden können.

Eine Möglichkeit diese Problem zu lösen ist den Speicher vor dem Linken des Moduls auf dem Knoten zu allokkieren. Problematisch ist hierbei, dass der Speicher exklusiv für das Modul reserviert werden muss, also auch nach einem Neustart des Systems sichergestellt werden muss, dass der Speicher nicht anderweitig verwendet wird. Des Weiteren müssen Funktionen bereitgestellt werden, welche den Speicher des Moduls initialisieren, also die `.data` Sektion vom Programmspeicher in den Datenspeicher kopieren und die `.bss` Sektion mit 0 initialisieren.

Alternativ könnte versucht werden den Zugriff auf statische Daten über eine Indirektionsstufe zu leiten. So könnte durch den Kernel ein Zeiger bereitgestellt werden, welcher auf den Speicherblock zeigt, in welchem sich die statischen Daten des Moduls befinden. Dies müsste jedoch auf Compilerbene gelöst werden und ist daher sehr aufwendig.

Die dritte Möglichkeit wäre auch weiterhin keine statischen Variablen zuzulassen und dafür das Laden von Daten vom Programm- in den Arbeitsspeicher durch die Bereitstellung weiterer Funktionen zu vereinfachen (siehe auch nachfolgender Punkt). Diese Lösung ist wahrscheinlich die sinnvollste, zumal globale statische Variablen auch über den *Named Memory* realisiert werden können.

Verwenden von Daten im Programmspeicher

Wie in Kapitel 5.5 erwähnt bestehen beim Zugriff auf Programmspeicher oberhalb von 64 KiB gewisse Einschränkungen. Um den Flash mit 17 Bit adressieren zu können muss anstatt des LPM (Load Programm Memory) der ELPM (Enhanced LPM) Befehl verwendet werden. Auf dem ATmega128 ist dies der einzige Befehl, der das spezielle RAMPZ Register auswertet, in welches das 17te Bit gespeichert wird. Aus diesem Grund wird das Z-Register im Normalfall nicht gesichert. Obwohl die Interrupts nicht auf dieses Register zugreifen sollten, wurden bei eingeschalteten Interrupts Lesefehler beobachtet. Die genaue Ursache ist jedoch unbekannt.

Kann die Ursache für dieses Problem nicht gefunden werden, können die Funktionen so umgeschrieben werden, dass die Interrupts entweder abgeschaltet werden, oder eine Änderung des RAMPZ Registers erkannt wird und die Daten nochmals ausgelesen werden.

Eine weitere Einschränkung, welche momentan besteht ist, dass alle libc Funktionen für den Zugriff auf den Programmspeicher den LPM-Befehl verwenden. Wird ein Modul oberhalb der 64 KiB Grenze abgelegt, können diese nicht verwendet werden. Durch die Einbindung einer entsprechenden Bibliothek stehen in der bereitgestellten Entwicklungsumgebung die Funktionen der `string.h` bereits mit ELPM Unterstützung zur Verfügung. Entsprechende Versionen der `stdio.h`, wie beispielsweise `printf()`, müssen noch implementiert werden.

Unterstützung von Bibliotheken in Modulen

Verwenden verschiedene Module die gleichen Funktionen, welche jedoch nicht im Kernel enthalten sind, kann es sinnvoll sein, diese Funktionen in ein eigenes Modul zu speichern um so die Größe der Module zu reduzieren. Wie auch beim Kernel könnten die Bibliotheksmodule über die CRC32 Prüfsumme identifiziert und die Adressen der Funktionen als Symboltabelle an den Linker übergeben werden. Es müsste jedoch sichergestellt werden, dass die dadurch entstehenden Abhängigkeiten zwischen verschiedenen Modulen immer erfüllt sind.

Verwendung von Position Independent Code

Die Verwendung von Position Independent Code (PIC), also Programmcode, der an einer beliebigen Stelle platziert werden kann, wird momentan nicht unterstützt, weil dieser im Moment aus technischen Gründen auf 4 KiB begrenzt ist (vgl. Kap. 3.5.5).

Momentan können Module auf Grund der Flashspeicherverwaltung nicht beliebig platziert werden. Diese müssten daher entsprechend angepasst werden. Des Weiteren muss sichergestellt werden, dass ein mit `--relax` gelinktes Modul auch wirklich PIC enthält.

Theoretisch wäre es auch möglich den Linker so anzupassen, dass die 4 KiB Grenze durch mehrere `rjmp`-Befehle aufgehoben werden kann. Um 6 KiB zu springen, könnte beispielsweise zunächst 4 KiB und dann noch einmal 3 KiB gesprungen werden. Hierzu müsste der Linker an der entsprechenden Stelle einen `rjmp`-Befehl einfügen. Wie auch schon bei den *Jump Tables* ist der Rücksprung unproblematisch, da die Rücksprungadresse auf den Stack geschrieben wird.

Laden großer Kernel

Im Moment ist nicht möglich Kernel zu aktualisieren, welche größer als 60 KiB sind. Dies liegt daran, dass der neue Kernel vollständig im Flash zwischengespeichert werden muss, bevor der alte durch den neuen Kernel vom Bootloader überschrieben werden kann (vgl. Kap. 5.3.7). Der BTnode hat jedoch weitere 160 KiB externen Arbeitsspeicher zur Verfügung, welcher über Paging angesprochen werden kann. Dieser Speicher kann dazu genutzt werden einen Kernel zwischenzuspeichern. Kommt es bei dieser Lösung beim Schreiben auf den Flash zu einem Wegfall der Spannung, ist es nicht mehr möglich den Kernel wiederherzustellen.

Echtzeitanforderungen

Bei kritischen Echtzeitanforderungen kann es problematisch werden, dass während des Schreibens auf den Flash keine Echtzeitanforderungen unterhalb von ca. 4 ms mehr erfüllt werden können. Im Bootloaderbereich stehen im Moment noch über 3 KiB Programmspeicher zur Verfügung. Dieser kann möglicherweise genutzt werden, um wichtige Interrupts auch während des Schreibens des RWW Bereichs im NRW Bereich zu behandeln (vgl. Kap 5.3.1).

Löschen von verwandten Modulen

Es besteht bei der aktuellen Implementierung die Möglichkeit Module zu löschen, welche von einem Thread verwendet werden. Um dies zu vermeiden müsste gegebenenfalls eine Möglichkeit geschaffen werden um Module so zu sperren, so dass diese nicht gelöscht werden können. Dabei ist zu beachten, dass die Hauptapplikation eines Moduls weitere Threads innerhalb eines Moduls starten kann.

8.1.2 Implementierung des Named Memory

Bei *Named Memory* gibt es zwei Punkte, welche verbessert werden können, jedoch muss untersucht werden, ob und in welchem Umfang eine solche Implementierung Sinn macht.

Erstens wird zur Erkennung einer Korruption des Headers eine *CRC8* Prüfsumme verwendet. Die Verwendung einer *CRC16* Prüfsumme würde die Erkennungswahrscheinlichkeit wesentlich erhöhen. Auf Grund der Größe des verfügbaren Arbeitsspeichers ist es jedoch beim BTnode unwahrscheinlich, dass ein *Named Memory* Element durch die `.bss` oder `.text` Sektion überschrieben

wird. Aus diesem Grund wird die Prüfsumme vor allem dazu verwendet um zu erkennen, ob es sich bei den Daten am Ende des Speichers um das Wurzelement des *Named Memory* oder uninitialisierten Speicher handelt.

Zweitens gibt es bis jetzt keine Funktionssammlung, um die Handhabung der im *Named Memory* gespeicherten Daten zu optimieren. So könnten spezielle Funktionen zur Berechnung von Prüfsummen zur Verfügung gestellt werden.

Auch wäre es möglich einen Datensatz immer doppelt im *Named Memory* zu speichern, wobei eine Version immer aktiv ist. Soll in den Datensatz geschrieben werden, wird in den inaktiven Datensatz geschrieben und dieser nach der Beendigung des Schreibvorgangs aktiviert. Auf diese Weise kann sichergestellt werden, dass die Integrität mindestens eines Datensatzes immer gewährleistet ist, für den Fall, dass der Knoten, beispielsweise durch den Watchdog Timer, neu gestartet wird.

8.1.3 Sonstige Punkte

Ein Gesichtspunkt, der bei der Speicherzuordnung zu Threads bis jetzt nicht genauer betrachtet wurde, ist die Übereignung von Speicher von einem Thread zum anderen. Bei Bedarf kann diese Funktion mit geringem Aufwand implementiert werden. Alternativ kann für diesen Zweck auch der *Named Memory* verwendet werden. Dieser kann im Zweifelsfall von Hand wieder freigegeben werden.

Ein Punkt, welcher weder mit der Modulunterstützung, noch dem *Named Memory* zusammenhängt, ist die automatische Bestimmung der Stackgröße. Dieser Punkt ist auf Mikrocontrollern mit mehreren Threads insofern besonders interessant, da auf Mikrocontrollern wenig Speicher vorhanden ist und daher die Stackgröße möglichst klein gewählt werden sollte. Wird die Stackgröße zu klein gewählt, kommt es zu einem Stacküberlauf, welcher häufig schwer zu erkennen ist. Hier gibt es bereits Werkzeuge, die jedoch eher einen Proof-of-Concept Zustand haben und in der Praxis nicht ohne Probleme eingesetzt werden können [25,26]. Die automatische Bestimmung einer geeigneten Stackgröße könnte jedoch die Stabilität des Gesamtsystems erhöhen und so den benötigten Implementierungsaufwand rechtfertigen.

8.2 Ausblick

Die vorgestellten Funktionen ermöglichen es situationsbedingt zur Laufzeit Applikationen auf die BTnode Sensorknoten zu laden. So ist es möglich flexibel auf neue Situationen reagieren zu können ohne andere, auf dem Knoten laufende Anwendungen neu starten zu müssen. Hierdurch wird es beispielsweise möglich, dass ein Wartungstechniker in einem mit Sensorknoten vernetzten Haus ein Wartungsmodul installieren kann, ohne dass die Hausregelung heruntergefahren werden muss. Das Modul wird hierzu automatisch für jeden Sensorknoten, abhängig vom installierten Kernel und freien Flashspeicher, kompiliert, distributiert und gestartet. Nachdem der Wartungstechniker mit

seiner Arbeit fertig ist, kann das Modul wieder beendet und gelöscht werden, ebenfalls ohne die Hausregelung herunterfahren zu müssen. Die Funktionen können aber auch überall dort eingesetzt werden, wo eine effiziente Aktualisierung zur Laufzeit benötigt wird.

9 Referenzen

- [1] Pei Zhang, Christopher M. Sadler, Stephen A. Lyon, und Margaret Martonosi, "Hardware design experiences in ZebraNet," *Proceedings of the 2nd international conference on Embedded networked sensor systems*, Baltimore, MD, USA: ACM, 2004, S. 227-238.
- [2] Sébastien Truchat, "Rekonfiguration von mobilen autonomen Diensten in heterogener Umgebung," *Dissertation*, Friedrich Alexander Universität Erlangen-Nürnberg, Germany, 2006.
- [3] Zengyu Lu, "Profilbasierte Rekonfiguration von Sensorknoten Profile Matching & Content Adaption," *Studienarbeit*, University of Erlangen-Nuremberg, 2006.
- [4] Adam Dunkels, Niclas Finne, Joakim Eriksson, und Thiemo Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," *Proceedings of the 4th international conference on Embedded networked sensor systems*, Boulder, Colorado, USA: ACM, 2006, S. 15-28.
- [5] Pedro Jos Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, und Kurt Rothermel, "FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks," *Wireless Sensor Networks*, vol. 3868/2006, 2006, S. 212-227.
- [6] Joel Koshy und Raju Pandey, "Remote incremental linking for energy-efficient reprogramming of sensor networks," *Proceedings of the Second European Workshop on Wireless Sensor Networks, 2005.*, 2005, S. 354-365.
- [7] Jonathan Hui, "Deluge 2.0-TinyOS Network Programming," Juli. 2005.
- [8] Falko Dressler, Moritz Strübe, Rüdiger Kapitza, und Wolfgang SchröderPreikschat, "Dynamic Software Management on BTnode Sensors," *4th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS): International Workshop on Sensor Network Engineering (IWSNE)*, Santorini Island, Greece: IEEE, 2008, S. 9-14.
- [9] Mustafa Yücel, "Role and Link-State Selection for Bluetooth Scatternets," *Masterarbeit*, Eidgenössische Technische Hochschule Zürich, 2006.
- [10] Wikipedia contributors, "Task (computers) --- Wikipedia, The Free Encyclopedia," *Wikipedia, The Free Encyclopedia*, Wikimedia Foundation, 2008.
- [11] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, und Muneeb Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," *Proceedings of the 4th international conference on Embedded networked sensor systems*, Boulder, Colorado, USA: ACM, 2006, S. 29-42.
- [12] Su Min Kim, Jo Woon Chong, Byoung Hoon Jung, Min Suk Kang, und Dan Keun Sung, "Energy-Aware Communication Module Selection through ZigBee Paging for Ubiquitous Wearable Computers with Multiple Radio Interfaces," *2nd International Symposium on Wireless Pervasive Computing, 2007*, 2007.
- [13] Derek M. Jones, *The New C Standard: An Economic and Cultural Commentary*, 2005.
- [14] Anatoly Sokolov, "patch #6352: Far pointer library," *AVR C Runtime Library*, Jan. 2008, <http://savannah.nongnu.org/patch/?6352> [Zugegriffen am 16. Juli 2008].
- [15] Philip Levis und David Culler, "Maté: A tiny virtual machine for sensor networks," *ACM SIGOPS Operating Systems Review*, vol. 36, 2002, S. 85-95.

- [16] Sam Michiels, Wouter Horré, Wouter Joosen, und Pierre Verbaeten, "DAViM: a dynamically adaptable virtual machine for sensor networks," *Proceedings of the international workshop on Middleware for sensor networks*, Melbourne, Australia: ACM, 2006, S. 7-12.
- [17] René Müller, Gustavo Alonso, und Donald Kossmann, "A virtual machine for sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 41, 2007, S. 145-158.
- [18] Maik Felser, Rüdiger Kapitza, Jürgen Kleinöder, und Wolfgang Schröder-Preikschat, "Dynamic Software Update of Resource-Constrained Distributed Embedded Systems," *Embedded System Design: Topics, Techniques and Trends*, vol. 231/2007, 2007, S. 387-400.
- [19] Benjamin Oechslein, "Realisierung eines dynamischen Code-Installationssystems für AVR Microcontroller," *Studienarbeit*, Uni Erlangen-Nürnberg, 2007.
- [20] Iulian Neamtiu, Michael Hicks, Gareth Stoye, und Manuel Oriol, "Practical dynamic software updating for C," *ACM SIGPLAN Notices*, vol. 41, 2006, S. 72-83.
- [21] Atmel Corporation, "AVR Instruction Set," 2008.
- [22] Adam Dunkels, "Full TCP/IP for 8-bit architectures," *Proceedings of the 1st international conference on Mobile systems, applications and services*, San Francisco, California: ACM, 2003, S. 85-98.
- [23] Adam Dunkels, "Rime A Lightweight Layered Communication Stack for Sensor Networks," *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, Netherlands: 2007.
- [24] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, und Thiemo Voigt, "Cross-Level Sensor Network Simulation with COOJA," *31st IEEE Conference on Local Computer Networks*, 2006, S. 641-648.
- [25] William P. McCartney und Nigamanth Sridhar, "Abstractions for safe concurrent programming in networked embedded systems," *Proceedings of the 4th international conference on Embedded networked sensor systems*, Boulder, Colorado, USA: ACM, 2006, S. 167-180.
- [26] John Regehr, Alastair Reid, und Kirk Webb, "Eliminating stack overflow by abstract interpretation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 4, 2005, S. 751-778.

10 Abbildungsverzeichnis

Abbildung 1.1: Demonstrator für einen intelligenten Container, welcher über ein Sensornetzwerk kommuniziert.....	1
Abbildung 1.2: Zebra mit Zebranet Sensorknoten um den Hals.....	2
Abbildung 2.1: BTnode - ein drahtloser Sensorknoten.....	9
Abbildung 2.2: Aufbau einer Harvardarchitektur.....	10
Abbildung 2.3: Zusammenspiel von Compiler und Linker.....	17
Abbildung 2.4: Aufbau des Speichers des BTnode Sensorknotens.....	18
Abbildung 3.1: Verwendung einer Symboltabelle, um aus einem Modul auf Funktionen des Kerns zuzugreifen.....	24
Abbildung 3.2: Funktionsweise von Jump Tables.....	26
Abbildung 5.1: Übersicht des Speicherlayouts nach dem Laden eines Moduls.....	38
Abbildung 5.2: RW, NRWW sowie Bootloaderbereich im Flash.....	38
Abbildung 5.3: Aufbau der Flashspeicherverwaltung.....	41
Abbildung 5.4: Löschen des Flashspeichers.....	42
Abbildung 5.5: Puffern eines Kerns.....	50
Abbildung 5.6: Überschneidung beim Kopieren eines Kerns.....	50
Abbildung 5.7: Allokation von Speicher im Flash.....	51
Abbildung 6.1: SRAM mit Anschlüssen zum "Aufsetzen" einer Batterie.....	59
Abbildung 6.2: Aufbau des Named Memory.....	61
Abbildung 6.3: Nut/OS Speicherverwaltung.....	63
Abbildung 6.4: Allokation von Speicher.....	64
Abbildung 7.1: Ablauf der Programmierung eines Moduls.....	73

11 Bildnachweise

Abbildung 1.1: Demonstrator für einen intelligenten Container, welcher über ein Sensornetzwerk kommuniziert: © MCB, www.intelligentcontainer.com

Abbildung 1.2: Zebra mit Zebranet Sensorknoten um den Hals: © Princeton, New Jersey, 08544 USA

Abbildung 6.1: SRAM mit Anschlüssen zum "Aufsetzen" einer Batterie: © Premier Farnell plc

12 Tabellenverzeichnis

Tabelle 2.1: Ausgewählte Sensorknoten.....	8
Tabelle 4.1: Erfüllung der Anforderungen durch die untersuchten Betriebssysteme.....	32
Tabelle 6.1: Vergleich der Speichersysteme.....	59
Tabelle 7.1: Kommandos des Protokolls zur Kommunikation mit dem Bluetooth Beispielkernel...	76