AG VERNETZTE SYSTEME
FACHBEREICH INFORMATIK
AN DER TECHNISCHEN UNIVERSITÄT
KAISERSLAUTERN

BACHELORARBEIT

ENTWICKLUNG EINES FLEXIBLEN UND ROBUSTEN BOOTLOADERS FÜR DIE IMOTE2 PLATTFORM

Markus Engel

---

15. Mai 2009

Entwicklung eines flexiblen und robusten Bootloaders für die Imote2 Plattform

Bachelorarbeit

Arbeitsgruppe Vernetzte Systeme
Fachbereich Informatik
Technische Universität Kaiserslautern

Markus Engel

Tag der Ausgabe : 06. März 2009
Tag der Abgabe : 15. Mai 2009

Themensteller : Prof. Dr. Reinhard Gotzhein
weiterer Prüfer : Prof. Dr. Jens Schmitt
Betreuer : Marc Krämer

Ich erkläre hiermit, die vorliegende Bachelorarbeit selbständig verfasst zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Kaiserslautern, 15. Mai 2009

(Markus Engel)

Abstract

Im Bereich mobiler Sensorknoten ergibt sich häufig das Problem Programme in den permanenten Speicher der Hardware zu übertragen. Herkömmliche Methoden sind zeitaufwändig und verzögern gerade bei der Entwicklung neuer Anwendungen den gesamten Entwicklungsprozess. Ziel dieser Arbeit ist die Entwicklung eines Bootloaders für die Imote2-Plattform. Dieser Bootloader ermöglicht die Programmierung des Sensorknotens über eine schnelle Standard-Schnittstelle. Die Kommunikation erfolgt über ein in dieser Arbeit formal spezifiziertes Übertagungsprotokoll. Der Bootloader wurde als robust gegenüber Fehlern entwickelt, sodass, ohne Zuhilfenahme weiterer Werkzeuge, immer ein definierter Zustand erreicht werden kann. Ein solcher Bootloader ist vor allem bei der Entwicklung neuer Anwendungen für die Sensorknoten von Bedeutung, da er durch die schnelle und zuverlässige Kommunikation kürzere Wartezeiten während der Übertragung erreicht. Dies verkürzt den gesamten Entwicklungsprozess neuer Anwendungen für den Sensorknoten und erleichtert eine spätere Wartung.

The domain of mobile sensornodes often yields the problem of transfering programs to the non-volatile memory of the hardware. Conventional methods are time-consuming and defer the overall development time when developing new applications. The intend of this thesis is the development of a bootloader for the Imote2 platform. This bootloader features programming a sensornode using a fast, standardized interface. The communication utilizes a transmission protocol, which is formally specified in this thesis. The bootloader provides robustness against errors, such that a defined state can always be reached without the need of further tools. A bootloader with these facilities becomes more important during the development of new applications, because it allows shorter transmission times due its faster and more reliable communication method, compared to more conventional ones. This reduces the overall development time of new applications for the sensor node and eases the later (after market) support.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Einsatz eingebetteter Systeme hat in den vergangenen Jahren immer mehr an Bedeutung gewonnen. Sie befinden sich in vielen Alltagsgegenständen wie Mobiltelefonen, Autos, Unterhaltungselektronik, medizinischen Geräten und mobilen Sensorknoten und verrichten dort ihren Dienst.

Neben Hardwareplattformen, die dem jeweiligen Dienst zugeschnitten sind, werden auch vermehrt universell einsetzbare Plattformen entwickelt. Diese Plattformen werden über Software der Aufgabe angepasst, wodurch beispielsweise Sensorknoten in vielfältigen Anwendungsgebieten eingesetzt werden können.
Die eingesetzte Software muss jedoch auch einfach auszutauschen sein. Eine übliche Methode zum Übertragen eines solchen Images ist die Verwendung der JTAG-Schnittstelle, über die der Flashspeicher beschrieben werden kann. Die niedrige Übertragungsrate dieser Schnittstelle impliziert jedoch lange Wartezeiten, die gerade während der Entwicklung neuer Images unerwünscht sind. Eine höhere Geschwindigkeit lässt sich durch die Verwendung anderer Schnittstellen erreichen, die auf dem Sensorknoten einen Bootloader erfordern, der das Image in den Flashspeicher schreibt. In dieser Arbeit wird ein für die Imote2 Plattform entwickelter Bootloader vorgestellt, der ein Image unter Verwendung der USB-Schnittstelle auf die Plattform lädt und dieses Image anschließend ausführen kann.

Diese Arbeit gliedert sich in folgende Abschnitte: In Kapitel 2 wird eine Übersicht über ähnliche und verwandte Projekte gegeben. Kapitel 3 stellt die verwendete Hard- und Software vor, die zur Entwicklung des Bootloaders nötig waren. Ferner wird eine Einführung in die verwendete USB-Kommunikation gegeben. In Kapitel 4 wird auf den Entwurf und die Implementierung des Bootloaders eingegangen. Dabei wird auch die Entwicklung des verwendeten Kommunikationsprotokolls und eines USB-Subsystems für die Plattform näher erklärt. Ferner wird auch auf Erweiterungsmöglichkeiten des Bootloaders eingegangen. Kapitel 5 beschreibt die Entwicklung eines Kodeladers, mit dem es möglich ist, dem Bootloader ein neues Image zu übertragen. Abschließend wird in Kapitel 6 eine Zusammenfassung und ein Ausblick gegeben. Im Anhang befindet sich eine kurze Beschreibung zum Verwenden der Software *OpenOCD* und eine Anleitung zur Konfiguration, Übersetzung und Installation des Bootloaders.

Kapitel 2

Related Work

In diesem Kapitel werden kurz Bootloader vorgestellt, die generelle Ansätze besitzen, aber auch speziell auf die verwendete ARM-Architektur angepasst sind. Einige der hier vorgestellten Konzepte sind bei der Entwicklung eingeflossen, konnten aber nicht vollständig übernommen werden.

2.1 Bootloader

Für die ARM-Architektur oder sogar im Speziellen für die Imote2 Plattform wurden bereits ähnliche Bootloader entwickelt, über die nun eine kurze Übersicht gegeben werden soll.

TinyOS-Bootloader ist ein von Junaith Ahemed Shahabdeen entwickelter Bootloader [30], der auf dem Imote2 vorinstalliert ist. Dieser hat wie der in dieser Arbeit entwickelte Bootloader die Fähigkeit, ein Image über die USB-Schnittstelle entgegenzunehmen und dieses auszuführen.
Der Bootloader hat allerdings diverse Limitierungen und offensichtliche Fehler. Einige dieser Fehler wurden bereits erkannt und im Zuge der Diplomarbeit von Thorsten Schmelzer [29] korrigiert. Das Hochladen eines Image über diesen Bootloader ist ein sehr zeitaufwändiger Prozess, wobei die Ursache dafür nicht vollkommen geklärt ist. Ein weiterer Nachteil ist, dass die PC-seitige Anwendung, die ein Image hochlädt, nur unter einer Cygwin [3] Umgebung lauffähig ist. Cygwin ist eine Sammlung von Bibliotheken, die die Linux-API unter Windows-Betriebssystemen zur Verfügung stellt. Da das Programm jedoch nicht nur von diesen, sondern zusätzlich auch von Windows-eigenen Bibliotheken abhängt, ist es folglich nur unter einem Windows-Betriebssystem mit installierter Cygwin Umgebung ausführbar. Für viele Nutzer stellt dies keine direkte Einschränkung dar, da die Entwicklung von TinyOS-Anwendungen oft mit einer installierten Cygwin Umgebung einhergeht. Problematisch wird allerdings der Einsatz eines anderen Betriebssystems, unter dem die Anwendung entwickelt wird. So ist es zwar generell möglich, TinyOS-Anwendungen unter Einsatz von Linux zu verwenden, jedoch muss zum Übertragen des fertigen Imagen dann die Cygwin Umgebung unter einem Windows-System verwendet werden. Dieses Problem ergibt sich generell auch, wenn TinyOS nicht als Betriebssystem

auf dem Knoten zum Einsatz kommen soll, sodass das Übertragen eines Images unter diesen Umständen für den Endnutzer erschwert wird.
Der Kode des Bootloaders ist an vielen Stellen undokumentiert oder gar falsch [30, 29] und somit schwer verständlich. Dies gilt auch für den Kode aus TinyOS, der teilweise in den Bootloader miteingeflossen ist. Dies führt dazu, den neuen Bootloader von Grund auf neu zu schreiben, anstatt den vorhandenen zu erweitern.

Das U-Boot ist die Weiterentwicklung des bekannten ARMboot-Projektes [1, 4]. Beide bieten generell die Möglichkeit Plattformen, die auf der ARM-Architektur basieren, zu booten und neue Programme zu übertragen. U-Boot ist vor allem wegen seiner zahlreichen Portierungen ein sehr häufig eingesetzter Bootloader für eingebettete Systeme.
Allerdings geschieht das Laden neuer Programme mit Hilfe einer Ethernet Netzwerkschnittstelle, die auf dem Imote2 nicht vorhanden ist. Die Kommunikation über Netzwerk ist zu sehr im Kode verankert, sodass eine Erweiterung um eine USB-Kommunikation nur durch weitreichende Erweiterungen am Kode möglich ist. Zudem wird nur die erste Generation von XScale Prozessoren (PXA25x) unterstützt.

blob ist ein Linuxloader für die StrongARM-Architektur [2]. Es handelt sich also einerseits um einen auf das Starten von Linux spezialisierten Bootloader, anderseits wird die PXA27x Prozessorfamilie nicht unterstützt. Weiter kann nach derzeitigem Kodestand nur die serielle Schnittstelle zur Übertragung genutzt werden. Das Projekt wird außerdem seit dem Jahr 2002 nicht mehr weitergepflegt. Eine Adaption des bestehenden Kodes für den Imote2 wurde nicht weiter verfolgt, da neben der Portierung auf die PXA Plattform auch noch eine Erweiterung der Kommunikation nötig gewesen wäre. Somit hätten diese Anpassungen letztlich zu einem größeren Arbeitsaufwand geführt als den Bootloader neu zu schreiben.
Dennoch ist die Kodequalität des Projekts gut verständlich. Die Idee der Programmierung des Flashspeichers konnte in dieser Arbeit verwendet und erweitert werden.

2.2 DFU Protokoll

Das Device Firmware Upgrade (DFU) [32] Protokoll wurde von mehreren Firmen zum Übertragen von Firmware über eine USB-Schnittstelle entwickelt. Die Idee hinter dem Protokoll ist, dass Geräte, die sowieso über eine USB-Schnittstelle verfügen, auch über diese ihre Firmware aktualisieren können. Beispiele für solche Geräte sind etwa der freie RFID Leser OpenPCD [11] oder die Handysoftware Openmoko [12].
Die Standardisierung des Protokolls erbringt den Vorteil, dass bereits Software zum Übertragen von Firmware existiert.
Jedoch gibt es auch einige Nachteile, die das Protokoll als Grundlage für den entwickelten Bootloader ausschließen. Zum einen zeigt die für das Protokoll gewählte USB-Kommunikationsart diverse Limitierungen, die die Übertragungsrate erheblich verlangsamen. Andererseits wird in diesem Protokoll davon ausgegangen, dass die hochgeladenen Daten immer an die selbe Stelle programmiert werden sollen, sodass das Protokoll keine Möglichkeit bietet, eine konkrete Zieladresse anzugeben.

Kapitel 3

Grundlagen

Im folgendem Kapitel werden einige Grundlagen eingeführt. Zuerst wird eine kurze Terminologie eingeführt, die in dieser Arbeit verwendet wird. Dann wird die verwendete Hardware und Software vorgestellt, und schließlich eine Einführung in die USB-Kommunikation gegeben.

3.1 Terminologie

In der Arbeit wird die folgende Terminologie verwendet:

- Als *Imote* wird, wenn nicht anders erwähnt, die Imote2 Plattform bezeichnet.
- Ein *Image* bezeichnet das Programm, das als eigentliche Anwendung auf dem Imote zum Einsatz kommt.
- Der *Kodelader* ist eine PC Applikation, die ein Image über die USB-Schnittstelle an den Bootloader überträgt, sodass dieser das Image auf den Speicher schreiben kann.
- Als *Bootloader* wird das Programm und dessen Quelltext bezeichnet, das beim Systemstart des Imotes ausgeführt wird. Dieses Programm hat die Fähigkeit, die eigentliche Applikation (Image) zu starten oder ein neues Image über die USB-Schnittstelle entgegenzunehmen und auf den permanenten Speicher des Imotes zu programmieren.
- Als *Flashspeicher* wird, wenn nicht anders erwähnt, der permanente Speicher des Imotes referenziert.

Dieses Anwendungsszenario ist zur Übersicht in Abb. 3.1 skizziert.

3.2 Hardware

Zunächst wird die Hardwareplattform Imote2 mit ihren Eigenschaften vorgestellt. Für die Übertragung des Bootloaders und zur Suche (Debuggen) ist weitere Hardware notwendig, auf die in den folgenden Abschnitten eingegangen wird.

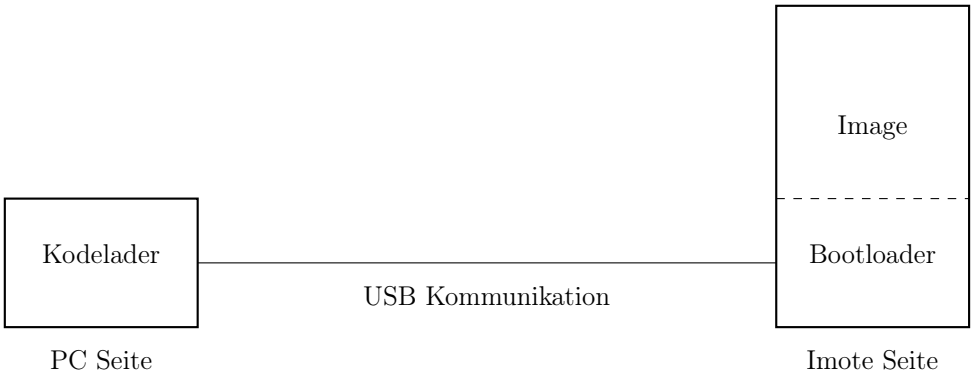

Abbildung 3.1: Übersicht zur Terminologie

Abbildung 3.2: Beide Seiten des Imote2

3.2.1 Imote2

Hauptplattform der in dieser Arbeit verwendeten Hardware ist der von der Firma Crossbow entwickelte Sensorknoten *Imote2* [15]. Hervorzuheben sind folgende Bestandteile des Imotes:

CPU ist der von Intel/Marvel entwickelte PXA271 Prozessor [20]. Dieser gehört der XScale Reihe an und arbeitet mit dem ARM-v5TE Befehlssatz. Der Chip integriert neben der CPU noch weitere Peripherie. Darunter befindet sich ein Speichercontroller, der 256 KiB SRAM, 32 MiB SDRAM und 32 MiB Flashspeicher verwaltet. Der Speicher ist mit dem Prozessor direkt verbunden und in Schichten im selben Chip angeordnet [23, 21]. Zur weiteren Peripherie, die für diese Arbeit benötigt wurden, gehören ein USB 1.1 Client Controller, ein DMA Controller sowie JTAG- und UART-Schnittstellen. Die weiteren Komponenten sind im Datenblatt des PXA271 [20] dokumentiert, werden in dieser Arbeit jedoch nicht benötigt.
Abbildung 3.2 zeigt die beiden Seiten der Imote2 Plattform.

Transceiver Der Texas Instruments CC2420 [20] bildet die drahtlose Schnittstelle der Plattform auf Basis des IEEE 802.15.4 Standards. Der Transceiver arbeitet auf dem 2.4 GHz Band und bietet eine maximale Übertragungsrate von 250 Kbit/s. Er ist über die SPI-Schnittstelle des Prozessors verbunden.

Abbildung 3.3: Imote2 mit Interface Board und JTAGKey-Tiny

3.2.2 JTAG-Schnittstelle

Die JTAG-Schnittstelle des PXA271 Prozessors soll dazu verwendet werden, den Bootloader selbst in den Flashspeicher zu übertragen. JTAG ist der gebräuchlichere Name für den IEEE 1149.1 Standard und bezeichnet eine genormte Schnittstelle, die es ermöglicht, integrierte Schaltungen in ihrer Applikation, also einer fertigen Schaltung, zu debuggen. Darüber hinaus ist es auch möglich, den Flashspeicher über JTAG auszulesen und zu programmieren.
Die physikalische Form der Schnittstelle ist nicht genormt, jedoch sind bestimmte Bauformen üblich. Für die ARM-Architektur wird üblicherweise ein 12/20 poliger Wannenstecker verwendet. Die JTAG Signale werden beim Imote zwar herausgeführt, jedoch wird noch der Anschluss eines von Crossbow entwickelten Interface Board [14] benötigt, um die üblichere Bauform verwenden zu können. Der Anschluss dieser Schnittstelle an den PC wird mit einem weiteren Umsetzer erreicht. Dazu wurde der Amontec JTAGKey-Tiny [13] gewählt, der es nun erlaubt die JTAG-Schnittstelle des Interface Boards mit der USB-Schnittstelle des PCs zu verbinden. Diese Hardwarekonfiguration ermöglicht es nun, den Imote über die Standardsoftware OpenOCD (siehe Abschnitt 3.3.3) anzusprechen, zu steuern und den Flashspeicher zu programmieren. Abbildung 3.3 zeigt den Imote mit aufgestecktem Interface Board und JTAGKey-Tiny.

Genau wie den Bootloader selbst kann man über die JTAG-Schnittstelle natürlich auch das Image übertragen, was den Bedarf eines solchen Bootloaders in Frage stellt. Allerdings muss beachtet werden, dass JTAG um ein Vielfaches langsamer ist als die verwendete USB-Kommunikation, und wie eben beschrieben ist bei der Übertragung über JTAG weitere Hardware erforderlich.

3.3 Software

Der Bootloader ist in der Programmiersprache C entwickelt. Es werden Compiler, Assembler und Linker benötigt, die den Quelltext des Bootloaders in ARM-Maschinenkode übersetzen, um ihn auf dem Imote auszuführen. Die Wasabi

Toolchain bietet neben den erwähnten Tools noch weitere Entwicklungswerkzeuge und die C-Bibliothek µClibc. Eine weitere Software wird benötigt, um den übersetzten Kode über die verwendete JTAG-Hardware auf die Plattform zum übertragen. Hierfür wird die JTAG-Software OpenOCD verwendet.

3.3.1 Wasabi Toolchain

Der Bootloader wird auf einer x86 Plattform entwickelt, aber auf einer ARM Plattform ausgeführt. Der Compiler übersetzt den C-Quelltext in ARM Assemblerkode übersetzt und ist selbst unter der x86 Architektur ausführbar. Einen solchen Compiler nennt man Crosscompiler. Weiter werden noch ein Assembler und ein Linker benötigt, die ebenfalls unter der x86 Architektur ausgeführt werden und mit dem ARM Assemblerkode arbeiten.
Eine Sammlung von Entwicklungswerkzeugen bietet die Wasabi Toolchain [35]. Diese enthält einen C-Crosscompiler, der auf der GNU Compiler Collection (GCC) [5] aufbaut. Die GNU Binutils [5] stellen eine ganze Reihe an essentiellen Werkzeugen dar. Zu den wichtigsten gehören ein Assembler und ein Linker. Darüber hinaus bietet die Wasabi Toolchain noch Werkzeuge zum Debuggen eines Programms, die auf dem GNU Debugger (GDB) [6] aufbauen. Alle Werkzeuge werden durch die Wasabi Toolchain für die x86 Plattform geliefert und arbeiten mit ARM Assembler- bzw. Maschinenkode. Zur Toolchain gehört weiter die µClibc [10], eine Standard C-Bibliothek, die für den Einsatz auf eingebetteten Systeme optimiert ist.

3.3.2 SDL Environment Framework

Das von der AG Vernetzte Systeme entwickelte SDL Environment Framework (SEnF) [17] bietet generell die Möglichkeit, SDL Systeme an eine Umgebung anzubinden. Hierfür werden SDL-Signale von der Umgebung behandelt, die es ermöglichen, verschiedene Hardware über Treiber anzusprechen. Das SEnF wurde in der Projektarbeit von Thorsten Schmelzer [28] erweitert, um SDL Systeme auf dem Imote laufen zu lassen und die Schnittstellen der Plattform durch Treiber anzusprechen.
Zu diesen Treibern gehören auch Routinen zum Initialisieren der Hardware, Umschalten der Taktfrequenz, Ein- und Ausschalten von Interrupts sowie die Definition der Prozessorregister. Diese Teile konnten in dieser Arbeit wiederverwendet eingebunden werden.
Für die USB-Kommunikation, die in dieser Arbeit vorgesehen ist, existiert noch kein Treiber im Framework. Der daher neu entwickelte Kode soll wieder in das SEnF Projekt zurückfließen, sodass eine USB-Kommunikation mit SDL Systemen auf dem Imote ermöglicht wird.

3.3.3 OpenOCD

Der Open On-Chip Debugger [27] ist eine Standardsoftware zur Kommunikation mit JTAG-fähiger Hardware. Die Software ermöglicht es über die JTAG-Schnittstelle den

Prozessor zu steuern und den Flashspeicher zu programmieren. OpenOCD wurde in erster Linie dazu verwendet, den übersetzten Bootloader auf den Imote zu übertragen.
In Anhang A befinden sich Informationen zur Konfiguration und Verwendung von OpenOCD mit der hier verwendeten Hardwarekonfiguration.

3.3.4 libusb

Der Bootloader soll dazu genutzt werden, ein Image auf den Imote zu übertragen. Ein solches Image wird während der Entwicklungsphase häufig übertragen, es ist also wünschenswert, dass dieser Vorgang möglichst schnell abgeschlossen ist um möglichst wenig Zeit während dem Entwicklungsprozess zu verlieren. Da der Imote direkt auf der Plattform eine USB-Schnittstelle bietet, deren Netto-Datenrate mit 12 Mbit/s als ausreichend schnell angesehen wird, soll diese Schnittstelle für den Übertragungsvorgang verwendet werden.
Dazu ist auf Seite des PCs ein Kodelader erforderlich, der den USB-Anschluss des PCs ansprechen kann, um mit dem Bootloader zu kommunizieren. Das Projekt *libusb* [8] bietet eine Bibliothek, die es ermöglicht, USB-Endgeräte direkt aus Anwendungsprogrammen anzusprechen, ohne dass dazu eine Treiberentwicklung auf Betriebssystemebene erforderlich ist. Weiter abstrahiert die Bibliothek vom Betriebssystem, sodass derzeit Linux, BSD Varianten und Mac OS X unterstützt werden. Unter dem Namen *LibUsb-Win32* [9] steht auch eine kompatible Portierung für Windows-Systeme bereit.

Eine weitere Unabhängigkeit vom Betriebssystem wird durch die Entwicklung des Kodeladers in der Programmiersprache Java erzieht. Das Ausführen des Kodeladers kann ohne Neuübersetzung oder Anpassung auf fast allen Java-fähigen Plattformen erfolgen. Für den Zugriff aus Java auf den USB-Anschluss wird zusätzlich noch die *libusbJava* [26] verwendet. Sie stellt die Verbindung zwischen der *libusb* und der Programmiersprache Java her.

3.4 Universal Serial Bus

Der *Universal Serial Bus* (USB) ist ein ursprünglich 1996 von Intel entwickelter Standard [34] zur Verbindung von PCs mit Endgeräten. Die Einsatzgebiete von USB sind sehr weitläufig, so werden beispielsweise Eingabegeräte wie Maus, Tastatur, oder Ausgabegeräte wie Drucker, Soundkarten, aber auch Massenspeicher unterstützt. Um diese Gebiete abzudecken, werden viele unterschiedliche Anforderungen an das Protokoll gestellt. Dies führt dazu, dass das USB-Protokoll sehr schwergewichtig und somit schwierig zu implementieren ist.
Der Einzug von USB hat ältere Schnittstellen wie beispielsweise die serielle und parallele Schnittstelle der PCs fast verdrängt. Diese haben den Nachteil, dass sie oft nur ein Gerät pro Schnittstelle unterstützten und eine im Vergleich zu USB geringe Übertragungsgeschwindigkeit aufweisen. Der Vorteil dieser Schnittstellen liegt jedoch im einfacheren Protokoll, sodass diese immer noch häufig im eingebetteten

Bereich anzutreffen sind. Aus diesem Grund existieren auch USB-Konverter für diese Anschlüsse.

In der USB-Spezifikation sind viele Aspekte der Kommunikation standardisiert. Angefangen bei physikalischen Eigenschaften, wie der Steckerbauform und Leitungslängen, über elektrische Signalformen, sind auch die verschiedenen Transferarten und die logische Sicht der Kommunikation definiert. Über die Transferarten und den logischen Aufbau soll nun eine Einführung gegeben werden.

3.4.1 Standards

Von der USB-Spezifikation gibt es mittlerweile mehrere veröffentlichte Standards, die sich vor allem in ihrer Geschwindigkeit und ihren Fähigkeiten unterscheiden. Hervorzuheben ist dabei, dass alle Standards ab- und aufwärtskompatibel zueinander sind. So kann etwa ein Gerät, das für einen älteren Standard hergestellt wurde, an einen Bus angeschlossen werden, der einem neueren Standard folgt und umgekehrt.

Die vier relevanten Standards werden im folgenden genannt

- USB 1.0 ist die erste Version des USB-Standards, mit einer Netto-Datenrate von 12 Mbit/s
- USB 1.1 korrigierte Fehler des USB 1.0 Standards und fügte kleinere Neuerungen hinzu
- USB 2.0 enthielt weitere technische Neuerungen und erhöhte die Netto-Datenrate auf 480 Mbit/s
- USB 3.0 ist ein bisher nur vorgestellter Standard, der Datenraten von bis zu 5 Gbit/s erreichen soll.

Der USB 1.0 Standard ist, wie bereits erwähnt, fehlerhaft und heute praktisch nicht mehr anzufinden. USB 3.0 ist noch nicht auf dem Markt erhältlich und wird daher hier nicht weiter betrachtet.
USB 1.1 und 2.0 haben bis auf ihren Unterschied in der Geschwindigkeit viele Gemeinsamkeiten, und so sind auch beide Standards in der USB 2.0 Spezifikation enthalten [34]. Soweit nicht anders erwähnt, beziehen sich alle folgenden Angaben auf den USB 1.1 Standard.

3.4.2 Transferarten

Um die in der Einleitung genannten Anwendungsmöglichkeiten nutzen zu können, ist ein hohes Maß an Flexibilität nötig. Sind etwa ein Massenspeicher, eine Webcam und eine Tastatur am selben Bus angeschlossen, so hat die Webcam zwar eine konstante Übertragungsrate, es dürfen jedoch auch einzelne Pakete verloren gehen. Durch die daraus resultierende erhöhte Latenz ist eine Neuübertragung unnötig. Die Kommunikation mit dem Massenspeicher fordert jedoch eine fehlerfreie Übertragung aller Pakete und dabei einen bestmöglichen Durchsatz. Daneben müssen aber auch

noch kurze Pakete der Tastatur möglichst ohne Verzögerung übertragen werden.
Weiter sind auch die Anforderungen an die Übertragung von der Richtung abhängig. Die Webcam überträgt konstant Daten an den PC, erhält aber im Vergleich nur selten Pakete vom PC, beispielsweise um Belichtungsparameter zu ändern. Daher sind einzelne logische USB-Verbindungen unidirektional, sodass im Normalfall mindestens zwei Verbindungen aufgebaut werden, um die Hin- und Rückrichtung der Datenübertragung zu realisieren. Die einzige Ausnahme bildet der `Control Transfer`, der Kontrollnachrichten in beiden Richtungen austauscht und daher bidirektional ist.
Aufgrund der vielen verschiedenen Anwendungsgebiete und der daraus resultierenden Anforderungen bietet USB vier verschiedene Transferarten, die als verschiedene Dienstgüteklassen angesehen werden können. Diese Transferarten werden nun im einzelnen vorgestellt.

Control Transfer ist die einzige Transferart, die ein fest definiertes Format von den zu übertragenden Daten verlangt. Die Übertragung wird bis zu drei mal wiederholt, um Verluste und Fehler zu minimieren.
Die Kommunikation über Controltransfer wird immer vom Host initiiert. Sollen also Daten vom Client gesendet werden, muss der Host diese vom Client anfordern. Da nicht immer bekannt ist, wann neue Daten in diese Richtung zu übertragen sind, ist diese Transferart nur für Protokolle geeignet, bei denen Daten grundsätzlich vom Host angefordert werden.

Das Versenden geschieht in 3 Phasen:

- *Setup*: In dieser Phase sendet der Host ein Paket, in dem die Richtung und Länge der Daten der zweiten Phase angegeben ist. Weiter ist der Empfänger der Datenan- oder aufforderung darin enthalten. Dies kann das Gerät selbst, eine Konfiguration oder Schnittstelle sein (siehe Abschnitt 3.4.3).

- *Data*: In der Datenphase werden kein, ein oder mehrere Datenpakete versendet, je nach dem, was in der Setupphase bekannt gemacht wurde.

- *Status*: In dieser Phase sendet der Empfänger der Daten aus der zweiten Phase eine Quittierung.

Der Control Transfer erscheint wegen des festen Datenformats, sowie der Host-Initiierung des Datentransfers, recht unflexibel. Diese Transferart ist dennoch eine sehr wichtige und wird von jedem USB-Gerät genutzt. Über diese Transferart werden beim Anschluss des Geräts alle Informationen über das Gerät selbst übertragen und so dem Host bekannt gemacht. Diese Informationen teilen dem PC mit, welche Verbindungen zur Verfügung stehen, und mit welchen Eigenschaften sie konfiguriert wurden. Dies kann als Vorstufe eines Verbindungsaufbaus betrachtet werden. Aufgrund dieser Informationen kann der PC die entsprechenden Treiber für das Gerät laden, und mit der eigentlichen Kommunikation fortfahren.
Die Übertragung unterliegt im Besonderen auf dem Imote noch weiteren Einschränkungen, wodurch sie sich nur bedingt für beliebigen Datentransfer eignet.

Interrupt Transfer ist darauf ausgelegt, Datenströme mit geringen Datenaufkommen zu übertragen. Die maximale Paketgröße beträgt dabei 64 Byte. Dabei werden maximale Übertragungsverzögerungen garantiert und eine fehlerfreie Übertragung durch maximal dreimaliges Wiederholen des Pakets sichergestellt.
Aufgrund der Priorisierung des Transfers durch die maximale Übertragungsverzögerung eignet sich diese Transferart für Eingabegeräte wie Tastatur und Maus.

Isochronous Transfer ist für die Übertragung eines Datenstroms mit konstanter Datenrate und einer vereinbarten maximalen Verzögerung ausgelegt. Diese Vereinbarung findet beim Anmelden des Geräts am Host statt, sodass dieser Entscheiden kann, ob die dafür nötigen Ressourcen zur Verfügung stehen. Die maximale Paketgröße ist mit bis zu 1023 Byte um ein Bedeutsames größer als in den allen anderen Übertragungsarten. Werden Pakete fehlerbehaftet übertragen, werden sie ohne Neuübertragungsversuch vom Empfänger verworfen.
Aufgrund der Garantien und der erweiterten Paketgröße wird diese Transferart bevorzugt bei Echtzeitanwendungen wie einer Videoübertragung einer Webcam eingesetzt.

Bulk Transfer ist geeignet für alle Anwendungen, die keine Garantien bezüglich Verzögerung oder minimaler Bandbreite benötigen. Bulk Übertragungen erhalten die Bandbreite, die nach Abhandlung der anderen Transferarten noch zur Verfügung steht. Um die Aushungerung von Bulk Übertragungen zu verhindern, werden 10% der Buskapazität für diese Übertragungen reserviert. Wie bei Interrupt Transfer wird ein fehlerhaftes Paket bis zu drei mal wiederholt, bevor es negativ quittiert wird. Die maximale Paketgröße beträgt auch hier 64 Byte.
Aufgrund der garantiert korrekten Übertragung eignet sich diese Transferart vor allem zum Übertragen großer Datenmengen, die diese Eigenschaft erfordern und keine Garantien bezüglich der Übertragungsverzögerung beanspruchen. Beispiele hierfür sind Massenspeicher oder Drucker.

3.4.3 Konfiguration

Ein USB-fähiges Gerät kann verschiedene Funktionen über eine einzelne physikalische Verbindung anbieten. Um auch hier eine große Flexibilität zu erreichen, ist die Konfiguration eines USB-Geräts hierarchisch angelegt. Die einzelnen Ebenen der Konfigurationshierarchie werden im folgenden vorgestellt.

Gerätedefinition steht auf der obersten Ebene und beinhaltet eine vom USB-Implementers-Forum (USB-IF) vergebene Vendor-ID und eine vom Hersteller vergebene Product-ID. Diese kennzeichnen eindeutig ein Produkt. Weiter kann vom Hersteller eine Seriennummer vergeben werden, wodurch das Gerät in Kombination mit Vendor- und Product-ID eindeutig identifiziert werden kann.
Je nach Standard und Protokoll, dem ein Gerät folgt, kann noch eine Geräteklasse [33], Subklasse sowie ein Protokoll angegeben sein. Hat das Gerät mehrere Funktionen (etwa eine Tastatur mit integrierter Maus), so sind diese Klassen erst auf

tieferer Ebene definiert. Ein Gerät kann natürlich auch einem herstellerspezifischen Standard folgen.

Konfigurationen sind in der Gerätedefinition enthalten und repräsentieren disjunkte Funktionen des Geräts. Der Host kann eine der angegebenen Konfigurationen wählen. Eine Konfiguration enthält außerdem den maximalen Stromverbrauch, den das Gerät vom Bus beziehen will, wenn diese Konfiguration vom Host gewählt wird. Der Standard legt das Limit für die Stromaufnahme auf 500 mA fest. Wurde noch keine Konfiguration gewählt, darf das Gerät nicht mehr als 100 mA Strom über den Bus beziehen, wobei die meisten Hostcontroller und Hubs in dieser Hinsicht sehr tolerant sind.

Schnittstellen werden in den jeweiligen Konfigurationen definiert und repräsentieren einzelne Funktionen des Geräts, die nebeneinander existieren. So kann etwa ein Gerät eine Tastatur und eine Maus integrieren und gibt diese Funktionen mit zwei Schnittstellen, die einer Konfiguration angehören, bekannt.

Alternative Einstellungen stellen ein Konzept dar, das es ermöglicht, einzelne Schnittstellen einer gewählten Konfiguration durch Alternativen auszutauschen. So kann dieses Konzept als eine weitere Staffelung in der Konfigurationshierarchie von USB angesehen werden. Alternative Einstellungen werden jedoch relativ selten verwendet und im Bezug auf diese Arbeit sei angemerkt, dass alternative Einstellungen aufgrund eines Hardwarefehlers vom auf dem Imote enthaltenen Prozessor PXA271 nicht unterstützt werden [24].

Endpunkte bezeichnen einzelne logische Ende-zu-Ende-Verbindungen, über die Daten zwischen Host und Client ausgetauscht werden können. Jede Schnittstelle definiert eine Menge von Endpunkten, durch die die Kommunikation aufgenommen werden kann. Jeder Endpunkt ist mit einer eindeutigen Nummer, einer bestimmten Transferart (siehe Abschnitt 3.4.2), der maximalen Paketgröße und ist, insofern kein Control Transfer verwendet wurde, mit einer Richtung belegt. Für die Transferrichtung des Endpunkts werden die Bezeichnungen `IN` und `OUT` Endpunkt verwendet, wobei sich diese Richtungsangaben immer auf Sicht des Hosts beziehen.

Endpunkt 0 (Endpoint Zero) nimmt eine besondere Stellung ein: Er existiert immer und für jedes USB-Gerät unabhängig von der Schnittstellen- und Konfigurationshierarchie. Für Endpunkt 0 wird immer Control Transfer verwendet, und darüber die Bekanntmachung des Geräts und seinen Konfigurationen, Schnittstellen und Endpunkten ausgehandelt.
In der Praxis ist Endpunkt 0 der einzige Endpunkt, der Control Transfer verwendet, bei der verwendeten Imote-Plattform sogar der einzig mögliche.

Kapitel 4

Bootloader

Allgemein wird als Bootloader ein spezielles Programm bezeichnet, das nach einem Systemstart als erstes auf einer Plattform läuft. Seine Aufgabe besteht darin, die Hardware soweit zu initialisieren, dass ein weiteres Programm, etwa ein Betriebssystemkern, zur Ausführung gebracht werden kann, und dann dieses auch zu starten. Nach der Terminologie in der Einführung wird das vom Bootloader gestartete Programm im folgenden als Image bezeichnet.
Die Images, die auf eingebetteten Systemen ausgeführt werden, sind oft auf eine bestimmte Funktionalität zugeschnitten und bieten daher keine Möglichkeit, das Image selbst auszutauschen. Aus diesem Grund übernimmt auf diesen Plattformen der Bootloader diese Aufgabe: Zwischen der Initialisierung der Hardware und der Ausführung des Programms wird ein Zeitfenster eingeschoben, in dem die Kommunikation über eine äußere Schnittstelle mit dem Bootloader aufgenommen werden kann. Im Zuge dieser Kommunikation ist es möglich, dem Bootloader ein Image zu übergeben, das von diesem auf den permanenten Speicher der Plattform geschrieben wird.
Die Konzepte, der Entwurf und die Umsetzung eines solchen Bootloaders für die Imote-Plattform werden nun in den folgenden Abschnitten beschrieben.

4.1 Konzepte

Um die genannten Funktionalitäten des Bootloaders zu realisieren, müssen Konzepte erarbeitet werden, die die konkrete Arbeitsweise, die Form und Eigenschaften der Kommunikation und weitere Anforderungen an den Bootloader definieren. Diese Konzepte werden im folgenden vorgestellt.

4.1.1 Generelle Arbeitsweise

Die Aufgaben eines Bootloaders für eingebettete Plattformen wurden bereits in der Einführung angesprochen. Die Arbeitsweise des entwickelten Bootloaders soll nun im folgenden konkretisiert werden. Abbildung 4.1 zeigt dazu einen vereinfachten Startvorgang.

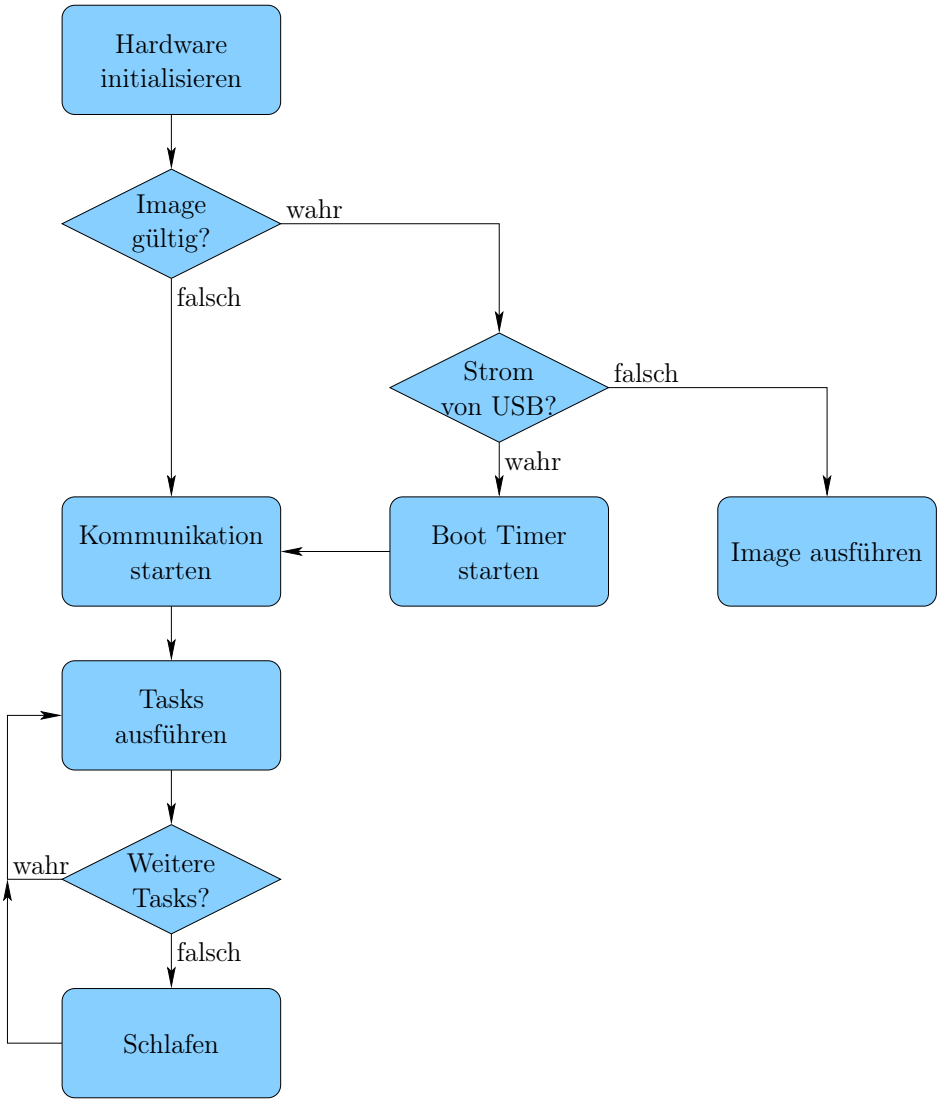

Abbildung 4.1: Arbeitsweise des Bootloaders

Im ersten Schritt wird die Hardware initialisiert, sodass der Bootloader ggf. eine Kommunikation aufnehmen oder das Image ausführen kann. Zu den initialisierten Kernkomponenten gehören die Prozessorcaches, die Memory Management Unit und die Taktfrequenz des Prozessors. Weiter werden einzelne Interrupts und die Subsysteme des Bootloaders aktiviert. Nach der Initialisierung ist der Bootloader in der Lage ein Image auszuführen. Wie schon in der Einleitung beschrieben, soll vor der tatsächlichen Ausführung die Möglichkeit bestehen, ein neues Image auf den Flashspeicher des Imotes zu übertragen. Dies geschieht in mehreren Entscheidungsstufen: Zuerst wird ein eventuell vorhandenes Image im Flashspeicher auf seine Vollständigkeit geprüft. Durch eine Anpassung im Linkerskript wurde erreicht, dass die Länge des Images im Image selbst hinterlegt ist und das Ende des Images durch eine magische Zahl angezeigt wird. So kann geprüft werden, ob am erwarteten Ende die magische Zahl eingetragen ist, also das tatsächliche Ende geschrieben wurde. In diesem Fall wurde das Image vollständig übertragen und kann ausgeführt werden.
Als nächstes wird überprüft, ob eine physikalische USB-Verbindung zu einem PC besteht. Besteht keine Verbindung, kann auch kein neues Image übertragen werden und der Bootloader startet schließlich das Image. Besteht eine Verbindung, so wird eine bestimmte Zeit abgewartet, in der eine Kommunikation vom PC aufgebaut werden kann um ein neues Image zu übertragen. Ist diese Zeit abgelaufen, ohne dass eine Kommunikation aufgebaut wurde, wird ebenfalls das Image gestartet.
Eine Ausnahme von diesem Verhalten tritt ein, wenn ein Image nicht vollständig übertragen wurde. In diesem Fall wird unabhängig davon, ob eine physikalische USB-Verbindung erkannt wurde, auf ein neues Image gewartet.

Wird im gegebenen Zeitfenster eine Kommunikation aufgenommen, so wird die Hauptschleife des Bootloaders betreten. Der Bootloader reagiert größtenteils nur auf Ereignisse, die durch Interrupts angezeigt werden. Beispiele für solche Ereignisse sind das Empfangen eines Pakets über die USB-Schnittstelle oder das Ablaufen einer Zeit. Das Behandeln der Interrupts sollte innerhalb kurzer Zeit geschehen, da während dessen weitere eintreffende Interrupts verzögert werden. Daher wurden zeitaufwändige Aufgaben wie das Programmieren des Flashspeichers in zwei Phasen aufgeteilt. Diese Aufgaben werden im folgenden *Tasks* genannt. Die Behandlung des zugehörigen Ereignisses reiht den eigentlichen Task nur in eine Warteschlange ein. Die Tasks in der Warteschlange werden abgearbeitet, wenn keine weiteren Ereignisse anstehen. Die Aufspaltung des Interrupthandlers wird auch in vielen Betriebssystemen eingesetzt und die beiden Teile werden dort als *First-* und *Second-Level Interrupt Handler* bezeichnet.

4.1.2 Anforderungen

Die Hauptfunktionalität des Bootloaders liegt im Ausführen eines Images und dem entgegennehmen eines neuen Images, das auf den Flashspeicher programmiert wird. Neben diesen Funktionen werden nun noch weitere Anforderungen formuliert, die den Leistungsumfang des Bootloaders definieren.

- **Keine Abhängigkeiten** zu anderen Daten des Flashspeichers
  Ziel ist es, unter alleiniger Verwendung des Bootloaders immer einen definier-

ten Zustand zu erreichen, auch wenn Teile des Flashspeichers korrupte Daten enthalten.

- **Wiederverwendung** des SDL Environment Frameworks
  Um Kodeduplizierung zu vermeiden und die Wartung zu erleichtern, sollen Teile des SDL Environment Frameworks (SEnF) verwendet werden.

- **Integration** neuer Teile in SEnF
  Da für einige Schnittstellen der Plattform noch keine Treiber in SEnF existieren, sollen diese neu entwickelten Subsysteme so generisch ausgelegt werden, dass diese im SEnF oder anderen Projekten verwendet werden können. Dies fordert auch eine ausreichende Dokumentation der Subsysteme.

- Prüfung auf **unvollständige Übertragungen** vor dem Ausführen
  Das auszuführende Image wird vor seiner tatsächlichen Ausführung auf seine Vollständigkeit überprüft. Wurde ein vorangegangener Übertragungsversuch abgebrochen, kann dies somit erkannt werden und das Image wird nicht ausgeführt. Stattdessen wird auf die Übertragung eines neuen Images gewartet.

- **Fortschrittsanzeige**
  Der Bootloader den Fortschritt von Operationen anzeigen und vor allem in Fehlerfällen ausreichende Informationen liefern. Eine einfache Statusanzeige wird durch die Verwendung der LEDs erreicht, genauere Informationen werden über die serielle Schnittstelle ausgegeben.

- **Programmieren beliebiger Adressen**
  Es soll möglich sein, eine beliebige Zieladresse des Flashspeichers zu programmieren. So können Teile des Images, wie etwa Konfigurationen oder Audiosamples ausgetauscht werden, ohne dass das gesamte Image neu übertragen werden muss. Eine Konfiguration enthält beispielsweise eine ID, die den Knoten eindeutig identifizieren kann, was beim Implementieren von Kommunikationsprotokollen eine häufige Anforderung ist. Der Austausch der Konfigurationen ermöglicht so verschiedene Ausprägungen des Images auf verschiedenen Knoten, ohne dass das Image dazu neu übersetzt werden muss.

- Hinterlassen der Hardware in **definiertem Zustand**
  Die Images, die vom Bootloader ausgeführt werden, erwarten einen bestimmten Zustand von der Hardware. Die Images sind so konzipiert, dass sie auch ohne Bootloader arbeiten können, deshalb erwarten sie den Einschaltzustand der Hardware. Somit muss der Bootloader die Effekte seiner eigenen Initialisierungen wieder rückgängig machen, bevor ein Programm gestartet wird. Entfällt diese Maßnahme, können unerwartete Interrupts, erhöhter Stromverbrauch oder sonstige unvorhersehbaren Effekte während der Ausführung des Images auftreten, die es zu verhindern gilt.

- **Modularisierung** des Quelltexts
  Der Aufbau des Bootloaders soll einer logischen Strukturierung folgen, die sich auch in der Einteilung in einzelne Dateien wiederfindet. Beispiele für diese logische Einheiten sind unter Anderem die allgemeinen Startroutinen,

Routinen zum Programmieren des Flashspeichers, Schnittstellen zur Kommunikation und das Kommunikationssystem selbst. Durch die Aufteilung kann erreicht werden, dass einzelne Teile ausgetauscht oder erweitert werden können, ohne dass dazu weitreichende Anpassungen an den anderen Komponenten nötig sind. So kann beispielsweise die vorhandene USB-Kommunikation um eine drahtlose Kommunikation erweitert werden.

4.1.3 Übertragungsprotokoll

Der Bootloader soll in der Lage sein, beim Systemstart der Plattform ein neues Image entgegenzunehmen und dieses auf den Flashspeicher des Imotes zu programmieren. Dafür wird die USB-Schnittstelle des Imotes verwendet. Die Kommunikation über diese Schnittstelle erfordert ein weiteres Protokoll, mit dem die erforderlichen Parameter und schließlich das Image zwischen einem PC und dem Imote ausgetauscht werden können.
Das dazu entwickelte Protokoll ist in Abb. 4.2 in Form eines Zustandsautomats skizziert. Die Kommunikation befindet sich stets in einem bestimmten Zustand. Die Zustände definieren jeweils die Signale, die Empfangen werden können. Diese Signale werden als Eingangssignale bezeichnet. Der Übersicht halber sind Parameter der Signale weggelassen. Der Empfang eines Eingangssignals löst ggf. eine damit verbundene Aktion aus und sendet ein Ausgangssignal, das aus einem Statuskode besteht. Der Statuskode zeigt den Erfolg der ausgeführten Aktion an. Nachdem das Ausgangssignal versendet wurde, wird ein Folgezustand betreten. Dieser hängt vom ursprünglichen Zustand, dem Eingangssignal und dem Ausgang der Aktion ab. Wird ein nicht erwartetes Signal in einem Zustand gesendet, führt dies direkt in den `ERROR`-Zustand.
Die einzelnen Zustände, Ein- und Ausgangssignale des Protokolls werden nun in den folgenden Unterabschnitten genauer erklärt.

4.1.3.1 Eigenschaften

Das entwickelte Protokoll weist verschiedene Eigenschaften auf, die vor allem zur Fehlerfindung und -behandlung beitragen. Die einzelnen Eigenschaften werden im Folgenden erklärt:

- Die **Reseteigenschaft** des Protokolls ermöglicht, den Automat immer in den `IDLE`-Zustand zurückzusetzen. Durch das Senden eines `Reset`-Signals ist somit sichergestellt, dass sich der Automat nach einer abgebrochen Übertragung immer in einem bekannten Zustand befindet, von dem eine erneute Übertragung erfolgen kann.
- Eine **nicht definierte Transition** liegt vor, wenn ein Signal eingeht, das im aktuellen Zustand nicht verarbeitet werden kann. In diesem Fall wird der Automat in den Fehlerzustand überführt und es wird ein entsprechender Statuskode zurückgesendet.
- Der Automat besitzt einen **Fehlerzustand**. Dieser wird in allen Fehlerfällen betreten und kann nur durch ein Neustart der Plattform verlassen werden.

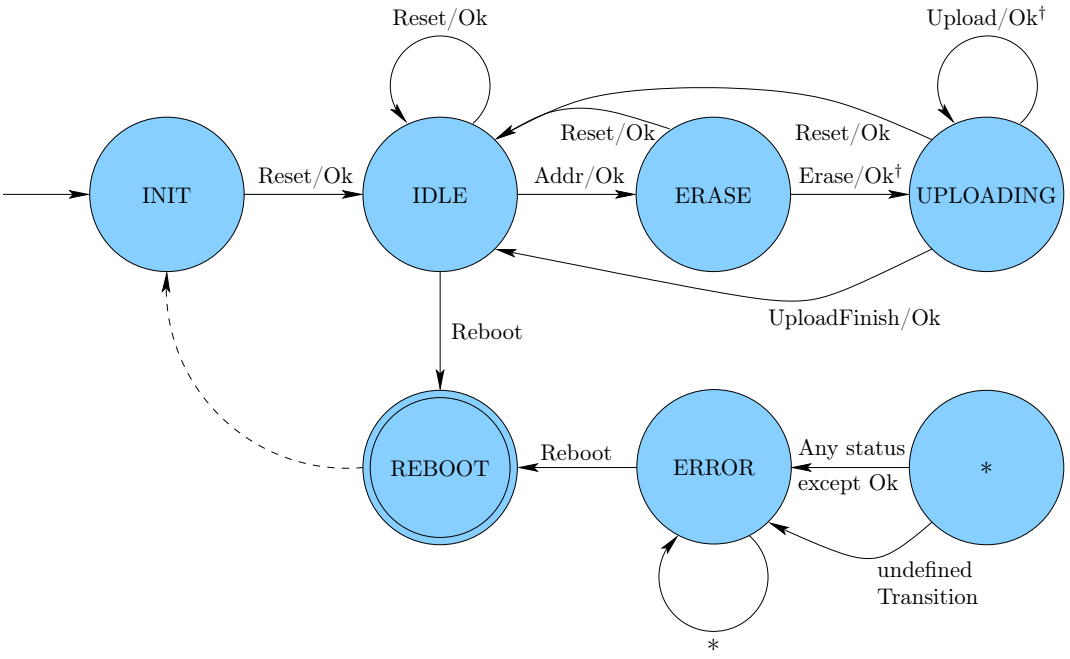

Abbildung 4.2: Übertragungsprotokoll auf Bootloaderseite als Zustandsautomat

4.1.3.2 Eingangssignale

Zunächst werden die Eingangssignale, also die Signale die der Automat erhält, erklärt: Der Empfang eines `Reset`-Signals versetzt den Automaten grundsätzlich in den `IDLE`-Zustand. Dies wird dazu verwendet, dass der Kodelader, nachdem er eine Verbindung zum Imote aufgebaut hat, auch dessen Zustand kennt. Wenn etwa eine Übertragung eines Images abgebrochen wird, beispielsweise auf Benutzerwunsch oder aufgrund eines Fehlers, kann der Kodelader durch das `Reset`-Signal nun also den Automat wieder in einen definierten Zustand versetzen.

Der Bootloader hat die Anforderung, eine beliebige Adresse programmieren zu können. Dazu muss die Zieladresse des zu übertragenden Images übermittelt werden. Der Kodelader sendet daher das `Addr`-Signal, das neben der Zieladresse noch die Länge des zu übertragenden Images enthält. Aufgrund dieser Angaben kann bereits geprüft werden, ob die Ziel- und Größenangaben verschiedene Bedingungen erfüllen. Ist eine dieser Bedingungen verletzt, wird das Signal mit einer negativen Quittierung beantwortet. Die einzelnen Randbedingungen, die für Zieladresse und Länge gelten, werden nun im folgenden erklärt.

- Es dürfen nur Adressen über der **4 MiB Grenze** beschrieben werden.
  Dies ist einerseits zum Schutz des Bootloaders, dass dieser sich nicht selbst überschreiben kann und andererseits eine technische Einschränkungen, da einige Flashblöcke unter dieser Grenze nicht aus laufendem Programmkode heraus entsperrt werden können
- Die **Summe** aus Adresse und Länge darf nicht die Größe des Flashspeichers übersteigen.

- Die Adresse muss mit einem **Flashblock beginnen**, also durch 128 KiB teilbar sein.
  Da immer nur ganze Flashblöcke gelöscht werden können, vereinfacht sich der gesamte Uploadvorgang, wenn vor dem Schreibvorgang die entsprechenden Blöcke gelöscht werden und sich somit in einem definierten Zustand befinden. Da somit ein vorheriges Image gelöscht ist, wird so auch die Funktionalität der Vollständigkeitsprüfung, wie in Abschnitt 4.1.1 beschrieben, sichergestellt.
  Zudem ist der Schreibalgorithmus darauf spezialisiert, nur 64 Byte ausgerichtete Adressen zu schreiben, was mit dieser Forderung sichergestellt ist.

Aufgrund des Aufbaus des Flashspeichers, der sich auf dem Imote befindet, müssen die einzelnen Blöcke des Speichers erst entsperrt und gelöscht werden, bevor sie wiederbeschrieben werden können. Nach Übermittlung der Zieladresse und der Länge durch das `Addr`-Signal wird anschließend durch den Empfang eines `Erase`-Signals der entsprechende Bereich des Speichers gelöscht. Dieser Vorgang kann in Abhängigkeit der Länge der Daten einige Zeit in Anspruch nehmen. Je nach Ausgang des Löschvorgangs wird anschließend ein positiver oder negativer Statuskode übermittelt.

Nachdem die entsprechenden Flashblöcke durch den Empfang des `Erase`-Signal gelöscht wurden, kann nun das Image übertragen und auf den Flashspeicher geschrieben werden. Dies geschieht durch Senden des `Upload`-Signals. Da die Länge eines Images in der Regel 256 KiB, also die Größe des SRAM, um ein Vielfaches übersteigt, kann es nicht in einem einzigen Signal übertragen werden. Daher werden die Daten des Images in mehrere Blöcke unterteilt, die sequentiell in mehreren `Upload`-Signalen übertragen werden, bis die gesamten Daten übermittelt sind. Die Größe der einzelnen Blöcke unterliegt weiteren Einschränkungen, die beim Übertragen beachtet werden müssen: Zum einen darf die maximale Paketgröße des Protokolls, wie in Abschnitt 4.1.3.4 nicht überschritten werden. Zum anderen muss die Größe jedes einzelnen Blocks durch 64 KiB teilbar sein, da der Flashspeicher darauf optimiert ist, 64 Byte parallel zu programmieren. Diese Einschränkung gilt nicht für den letzten Block.

Nach der kompletten Übertragung des Images wird der Programmiervorgang mit dem `UploadFinish`-Signal abgeschlossen. Durch den Empfang dieses Signals wird festgestellt, dass der Kodelader seinerseits alle Daten übertragen hat. Dies ist verifiziert, wenn die Länge der empfangenen Daten mit der im `Addr`-Signal angegebenen Länge übereinstimmt. So kann sichergestellt werden, dass die Automaten keine Fehler aufzeigen und sich an dieser Stelle beide Automaten einig sind, dass das Image vollständig übertragen wurde.

Um den Imote nach einer oder mehreren abgeschlossenen Übertragungen oder einem Fehlerfall neu zu starten, wird vom Kodelader das `Reboot`-Signal versendet, und dadurch die Plattform neu gestartet. Wird innerhalb des Zeitfensters, in dem beim Systemstart eine Kommunikation aufgenommen werden kann, keine Kommunikation aufgebaut, so wird das neu geschriebene Image vom Bootloader gestartet.

4.1.3.3 Behandlung von Fehlern

Während der gesamten Kommunikation und den Operationen auf dem Flashspeicher können verschiedene Fehler auftreten, die geeignet behandelt werden müssen. Eine Mindestanforderung ist dabei, dass der Benutzer eine aussagekräftige Fehlermeldung erhält, anhand derer bestimmbar ist, wo der Fehler aufgetreten ist und wie man ihn beheben kann. Jedes Eingangssignal und die damit verbundenen Aktion wird im Protokoll grundsätzlich durch einen Statuskode quittiert. Bei positiver Quittung geht der Kodelader von einem erfolgreichen Abschluss der Aktion aus. Anhand des Protokolls kann er den neuen Zustand des Bootloaders feststellen. Negative Statuskodes zeigen dagegen einen aufgetretenen Fehler an.

Diese Fehler werden in zwei Fehlerklassen eingeteilt: Die erste Klasse von Fehlern zeigt Fehler in der Kommunikation an. Sendet der Kodelader ein Signal, das im aktuellen Zustand nicht verarbeitet werden kann, wird das Signal mit einem `errUnknown`-Statuskode quittiert. Durch Empfang eines verfrühten `UploadFinish`-Signals oder eines nicht weiter erwarteten `Upload`-Signals wird festgestellt, dass weniger bzw. mehr Daten übertragen werden sollen, als durch das vorausgegangene `Addr`-Signal angegeben wurde. In diesen Fällen werden diese Signale mit einem `errTooShort` bzw. `errTooBig`-Statuskode beantwortet. Diese Statuskodes weisen grundsätzlich auf eine fehlerhafte Implementierung des Protokolls auf der Bootloader- oder Kodeladerseite hin. Diese Fehler sollten folglich nie auftreten.

Die zweite Klasse von Fehlern beschreibt die Fehler, die während dem Löschen und Programmieren des Flashspeichers auftreten können. Durch den Empfang eines `Erase`-Signals wird das Entsperren und Löschen der Flashblöcke angestoßen. Dabei kann der Speichercontroller melden, dass eine dieser Operationen fehlgeschlagen ist, was das Senden eines `errUnlock` bzw. `errErase`-Statuskodes veranlasst.
Beim Programmieren durch das `Upload`-Signal kann der Speichercontroller ebenfalls eine gescheiterte Schreiboperation melden, die dann in Form eines `errProgram`-Statuskodes angezeigt wird. Unter der Annahme, dass die Funktionen für die Flashoperationen fehlerfrei implementiert wurden, zeigen diese Signale grundsätzlich einen schwerwiegenden Fehler auf, der mit großer Wahrscheinlichkeit auf irreparable Schäden an den Flashblöcke zurückzuführen ist.

Das Auftreten eines Fehlers überführt den Automat immer in den Fehlerzustand, der nur das `Reboot`-Signal empfangen kann. Somit muss der Imote für einen neuen Versuch neu gestartet werden, was auch sicherstellt, dass sich das gesamte System wieder in einem definierten Zustand befindet.

4.1.3.4 Format

Die einzelnen Signale, die im Laufe der Kommunikation ausgetauscht werden, werden in Pakete kodiert. Das Format eines Pakets ist in Abb. 4.3 gezeigt. Die Signale können je nach Typ eine variable Anzahl von Parametern enthalten. Daher ist am Anfang jedes Pakets die Gesamtlänge in einem 16 Bit breiten Feld kodiert. Nach diesem Feld folgt das eigentliche Signal. Dieses wird durch eine eindeutige Nummer repräsentiert, die in einem 16 Bit breiten Feld enthalten ist. Nach dem Signal folgen ggf. die Signalparameter.

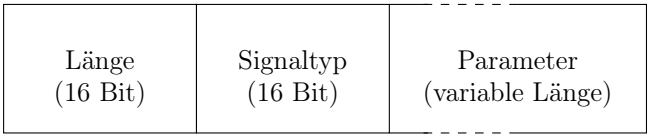

Abbildung 4.3: Datenformat der Übertragung

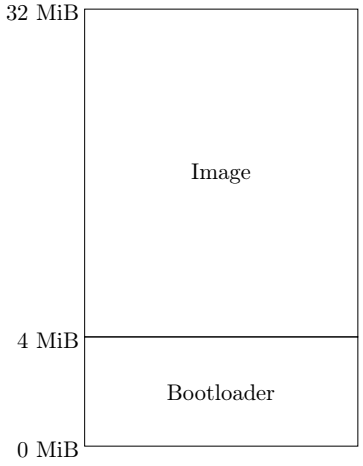

Abbildung 4.4: Aufteilung des Flashspeichers

Die Pakete werden in einem Puffer zwischengespeichert, der in seiner Größe beschränkt ist. Daher musste eine obere Schranke für die Paketgröße gesetzt werden. Diese wurde in der Implementierung auf 2052 Byte gewählt, sodass insgesamt 2048 Byte in den Signalparametern übertragen werden können.
Die Statuskodes in der Rückrichtung, also vom Imote zum Kodelader werden im selben Format versendet. Dabei wird der Statuskode als eindeutige Nummer anstelle des Eingangssignals übertragen. Da die Statuskodes keine Parameter enthalten, ist das Parameterfeld dieser Pakete leer.

4.1.4 Aufteilung des Flashspeichers

Die Aufteilung des Flashspeichers wurde nach dem in der Diplomarbeit von Thorsten Schmelzer [29] angepassten TinyOS-Bootloader übernommen und ist in Abb. 4.4 aufgezeigt.
Der Bootloader selbst beginnt bei Adresse `0x0` und kann bis zu 4 MiB belegen. Diese Zahl erscheint zwar recht groß, wurde aber aus Gründen der Kompatibilität belassen. Alle Images, die durch das entsprechende Linkerskript entstanden, sind darauf ausgelegt an der 4 MiB Grenze zu beginnen. Wird diese Grenze also verschoben, kann der Bootloader keine älteren Images mehr ausführen.
Ab der 4 MiB Grenze beginnt dann das Image, das vom Bootloader ausgeführt wird. Weitere Aufteilungen in diesem Bereich sind dem Benutzer frei überlassen.

4.2 USB-Kommunikation

Die Wahl der Kommunikationsschnittstelle fiel auf die USB-Schnittstelle des Imotes, da diese mehrere geeignete Eigenschaften aufweist. Zum einen besitzt die Plattform selbst eine USB-Mini-A Schnittstelle, sodass der Imote mit einem geeigneten Kabel direkt mit einem PC verbunden werden kann, ohne dass dazu weitere Hardware benötigt wird. Zum anderen deckt die Kommunikation über USB mehrere wünschenswerte Eigenschaften ab, darunter ein hinreichend schneller Datenaustausch mit bis zu 12 Mbit/s, Fehlererkennung und Empfangsgarantien durch ggf. mehrmaliges Übertragen von Paketen sowie Auf- und Abwärtskompatibilität zu anderen Versionen des Standards.
Der Prozessor des Imotes integriert einen USB 1.1 Client Controller, mit dem bis zu 23 Endpunkte frei programmierbar sind. Diese können in drei USB-Konfigurationen und acht Schnittstellen pro Konfiguration aufgeteilt werden. Zusätzlich bietet der Controller noch einen 4 KiB großen Paketpuffer, der auf die einzelnen Endpunkte aufgeteilt wird.
Um die Konfigurationsmöglichkeiten, die der Controller bietet, möglichst komfortabel nutzen zu können, und einen einfachen Zugriff auf die Daten der Endpunkte zu gewährleisten, wurde ein USB-Subsystem für die Imote-Plattform entwickelt, das nun vorgestellt wird.

4.2.1 Übersicht

Eine kurze Übersicht über den Aufbau des USB-Subsystems ist in Abb. 4.5 gezeigt. In der Abbildung sind die einzelnen Module des Subsystems und ihre Abhängigkeiten modelliert. Weiter sind auch Abhängigkeiten zu äußeren Subsystemen eingezeichnet.

Das Modul *Konfiguration* enthält die dynamische Konfiguration des Subsystems. Darunter fallen alle Parameter, die zur Laufzeit bekannt gemacht werden. Weitere Parameter des Subsystems fallen in das Modul *StatConfig*, das die statische Konfiguration enthält. Darunter fallen die Daten, die zur Übersetzungszeit bekannt sein müssen. Sie optimieren das Laufzeitverhalten und den Speicherverbrauch des Subsystems.

Das Modul *Kommunikation* enthält alle Routinen und Schnittstellen, die zum Empfangen und Versenden von Daten über die Endpunkte erforderlich sind. Dazu verarbeitet es alle auftretenden Interrupts und DMA-Aufträge, die zur Endpunkt Kommunikation beitragen, und übernimmt die Pufferung von Daten.

Die *Verwaltung* bietet Routinen zum Starten und Stoppen des Controllers. Zum Stoppen des Controllers gehört auch ggf. laufende Kommunikation abzubrechen und Pufferspeicher freizugeben.

Das Modul *Control Pipe* behandelt den Datentransfer auf Endpunkt 0. Über diesen Endpunkt werden während der Anmeldung des Geräts am Host die Gerätedefinition mit allen Konfigurationen, Schnittstellen und Endpunkten bekannt gemacht. Die Daten für die Einstellungen werden aus dem Konfigurationsmodul bezogen und in das vom USB-Protokoll geforderte Format umgesetzt.

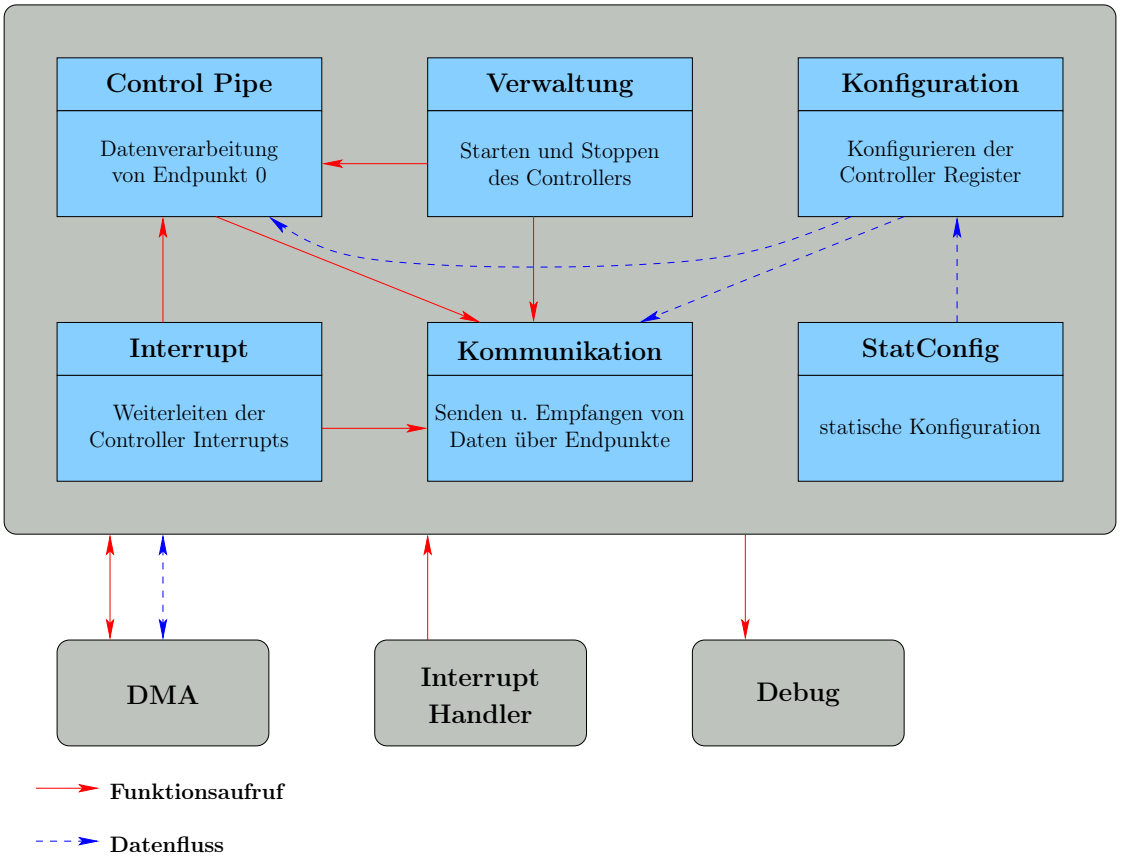

Abbildung 4.5: Aufbau des USB-Subsystems

Der USB-Controller vereint Funktionen, die zur USB-Kommunikation beitragen. Daraus ergeben sich viele Gründe für auftretende Interrupts. Beispiele dafür sind das erfolgreiche Senden oder Empfangen von Daten, Zurücksetzung und Rekonfiguration des Busses und Fehlerfälle. Das Modul *Interrupt* nimmt einen USB-Interrupt entgegen und unterscheidet anhand der Register des Controllers, an welches Modul der Interrupt weitergeleitet werden soll. Dort wird der entsprechende Interrupt dann weiterverarbeitet.

4.2.2 Konfiguration

Das Modul *Konfiguration* enthält, wie bereits erwähnt, Schnittstellen zur Laufzeitkonfiguration des USB-Subsystems. Über diese Schnittstellen werden die Gerätedefinition, die verwendeten USB-Konfigurationen, ihre Schnittstellen und deren Endpunkte bekanntgegeben. Diese Angaben werden einerseits dazu verwendet, die Register des USB-Controllers mit den entsprechenden Werten zu laden und andererseits werden daraus die Daten abgeleitet, die beim Anmelden an den Host über Endpunkt 0 die Gerätekonfiguration bekannt geben.

Die statische Konfiguration im Modul *StatConfig* enthält Parameter, die Optimierungen am USB-Subsystem vornehmen. So kann dort die Anzahl verwendbarer Konfigurationen, Schnittstellen und Endpunkte limitiert werden. Eine derartige Limitierung hat den Vorteil, dass eine Gerätekonfiguration weniger Speicher verbraucht und schneller verarbeitet werden kann.
Weiter kann mit entsprechenden Makrodirektiven angegeben werden, ob für die Endpunkte eine Doppelpufferung verwendet werden soll. Die Doppelpufferung ermöglicht es, ein weiteres Paket zu empfangen, während ein älteres noch ausgelesen wird. Dies führt zu einem Geschwindigkeitsvorteil in der Kommunikation.
Mit einer weiteren Makrodirektive kann entschieden werden, ob Interrupt Transfer oder direkter Speicherzugriff (DMA) verwendet werden soll. Beim Interrupt Transfer tritt bei jedem erfolgreichen Versenden oder Empfangen eines USB-Pakets ein Interrupt auf, der dann entsprechend behandelt werden muss. Diese Behandlung besteht dann daraus, ggf. ein weiteres Paket zu versenden bzw. das empfangene Paket wortweise auszulesen. Bei dieser Methode ist also der Prozessor in den Vorgängen beteiligt. Bei Verwendung von DMA dagegen wird das Auslesen oder Füllen des Sende- bzw. Empfangspuffers an einen DMA Controller abgegeben, sodass dieser erst nach Ausführung der Operation den Ausgang angibt. Am Datentransfer selbst ist der Prozessor somit unbeteiligt und kann währenddessen andere Berechnungen durchführen oder in einen stromsparenden Modus versetzt werden.

4.2.3 Kommunikation

Die Kommunikation über die konfigurierten USB-Endpunkte läuft über das entsprechende Kommunikationsmodul ab. Dabei wird unterschieden, ob Daten gesendet oder empfangen werden sollen.

Empfang von Daten Sobald Daten von einem `OUT` Endpunkt empfangen werden können, werden diese in einen Zwischenpuffer geladen und über eine Callback-Funktion ausgeliefert. Diese Callback-Funktion kann individuell für jeden empfangenden Endpunkt über seine entsprechende Konfiguration im Konfigurationsmodul angegeben werden. Anzumerken ist dabei, dass dieser Callback im Interruptkontext geschieht, also die Funktion so schnell wie möglich wieder zurückkehren sollte.

Senden von Daten Sollen Daten über einen `IN` Endpunkt gesendet werden, so wird einer entsprechenden Routine im Kommunikationsmodul der sendende Endpunkt und die Daten übergeben. Das Modul puffert diese Daten in einer Ausgangswarteschlange und sendet sie reihenfolgeerhaltend in USB-Paketen über den USB-Controller. Wird DMA verwendet, werden an dieser Stelle auch die DMA Anfragen verwaltet.

4.2.4 Abhängigkeiten zu anderen Subsystemen

Das USB-Subsystem hat einige Abhängigkeiten zu anderen Subsystemen, um seine Aufgaben zu erledigen:

DMA Der USB-Controller der Imote-Plattform bietet die Möglichkeit, Daten unter Verwendung von *direktem Speicherzugriff* (DMA) zu empfangen bzw. zu senden. Diese Option auch vom USB-Subsystem angeboten.
Die Verwendung des DMA-Controllers hat den Vorteil, dass der Prozessorkern nicht beim Kopieren von Speicherinhalten beteiligt ist und in dieser Zeit entweder andere Aufgaben übernehmen oder in einen stromsparenden Modus versetzt werden kann.

Interrupt Die Konfiguration und teilweise die Kommunikation wird von dem Empfangen vom Auftreten von Interrupts eingeleitet. Daher muss der globale Interrupthandler bei jedem Interrupt des USB Client Controllers diesen Interrupt an das USB-Subsystem weitergeben. Dieses wertet den Interrupt aufgrund der Werte in den Registern des Controllers aus und leitet weitere Maßnahmen zur Behandlung des Interrupts ein.

Debug Das Auffinden eventueller Fehler des Subsystems wird durch die Generatierung von Debugausgaben, die über das Debugsystem ausgegeben werden, realisiert. Diese Meldungen werden über eine serielle Schnittstelle versendet, und geben Informationen, Warnungen und Fehler aus. Informationen sind Meldungen, die beispielsweise die ausgelösten und behandelten Interrupts dokumentieren. Warnungen treten auf, wenn Eingabedaten nicht den entsprechenden Randbedingungen entsprechen, was auf eine Fehlkonfiguration hinweisen kann. Fehler treten beispielsweise auf, wenn angeforderter Speicher nicht verfügbar ist.

4.3 Flashprogrammierung

Die Imote-Plattform verfügt über einen 32 MiB großen Flashspeicher, der vom Bootloader programmierten werden soll. Der Flashspeicher wird mit Hilfe von Common Flash Interface (CFI) Befehlen angesprochen. Dieses Protokoll erlaubt neben Elementaroperationen wie Löschen und Beschreiben des Speichers auch eine Möglichkeit, diverse Parameter des Flashspeichers auszulesen. Zu diesen gehören Spannungsangaben, Gesamtgröße, Blockgrößen und -aufteilungen, Programmier- und Löschzeiten, Unterstützung erweiterter Befehlssätze usw. Dies bringt den Vorteil, dass man für alle Flashspeicher, die diesen Befehlssatz unterstützen, nur einen allgemeinen Treiber schreiben kann, der die entsprechenden Parameter aus den Flashinformationen ausließt.
Der hier entwickelte Treiber ist allerdings auf Größe, Aufteilung und Fähigkeiten des integrierten Flashspeichers des Imotes abgestimmt. Teilweise können diese Parameter über Definitionen in einer Konfigurationsdatei abgeändert werden, was jedoch nicht nötig sein sollte, sofern der Bootloader auf die Imote-Plattform beschränkt bleibt.

4.3.1 Flash-Modi

Flashspeicher, die über das CFI-Protokoll angesprochen werden, verfügen über mehrere Modi, die die momentane Operation des Flashcontrollers bestimmen. Im *Read*-Modus kann auf die Daten des Speichers zugegriffen werden. Da der Flashspeicher des Imotes an den Speichercontroller des Prozessors angeschlossen ist, unterscheidet sich der Zugriff der Daten des Flashspeichers nicht vom Zugriff auf den Arbeitsspeicher. Dies erlaubt theoretisch auch, dass Daten und Kode gemischt im Flashspeicher oder Arbeitsspeicher liegen können ohne dass dieser Umstand beim Entwickeln von Programmkode gesondert beachtet werden muss.
Im *Erase*-Modus können einzelne Flashblöcke gelöscht werden. Vor dem Programmieren müssen alle betroffenen Blöcke zuerst gelöscht werden. Das Programmieren geschieht dann im *Program*-Modus. Zum Programmiervorgang stehen grundsätzlich zwei Möglichkeiten zur Verfügung: Im *Word Program*-Modus kann der Speicher halbwortweise beschrieben werden. Die Länge eines Halbworts beträgt 16 Bit. Im *Buffered Program*-Modus wird ein 64 Byte großer Puffer des Flashcontrollers mit den zu schreibenden Daten gefüllt und anschließend der eigentliche Programmiervorgang vorgenommen. Dieser Modus erlaubt es dem Flashcontroller verschiedene Optimierungen beim Beschreiben durchzuführen. So wird der gesamte Puffer parallel auf den permanenten Speicher ausgeschrieben, was einen erheblichen Geschwindigkeitsvorteil gegenüber dem *Word Program*-Modus bietet.

Daten aus Flashblöcken können also nur gelesen werden, wenn sich der Flashspeicher im *Read*-Modus befindet, genauer im *Read Array*-Modus. Befindet sich der Flashspeicher nicht im *Read Array*-Modus, um beispielsweise einen Flashblock zu programmieren, liefern Lesezugriffe auf den Speicher nicht die gewünschten Daten, sondern Statusinformationen zum Programmiervorgang. Da sich auch der Programmkode, insbesondere auch der Kode, der zum Programmieren verwendet wird, selbst auf dem Flashspeicher befindet, werden also auch falsche Instruktionen gelesen und

ausgeführt, was im Allgemeinen zu nicht definiertem Verhalten führt und daher verhindert werden muss. Dieser Kode wird im folgenden *kritischer Kode* genannt. Zur Behandlung dieses Problems wurden verschiedene Lösungsansätze erarbeitet, die im folgenden diskutiert werden.

Passive Verwendung des Instruktionsspeichers Der PXA271 verfügt über einen Instruktionsspeicher (Instruction Cache), der in 32 Byte Blöcken Instruktionen aus dem Speicher vorlädt, um so den Zugriff zu beschleunigen.
Übersteigt nun der kritische Kode nicht 32 Byte (gezählt in Instruktionen in Maschinenkode) und richtet man ihn an einer 32 Bytegrenze aus, so wird er vollkommen in den Instruktionsspeicher vorgeladen und von dort ausgeführt.
Der Nachteil dabei ist, dass der Kode unabhängig vom Compiler nicht die 32 Bytegrenze übersteigen darf, also am besten compilerunabhängig und somit direkt in Assembler zu implementieren ist. Weiter ist nicht gewährleistet, dass der Prozessor währenddessen nicht weiteren Kode aus dem Speicher vorlädt und somit wieder ungültiger Kode geladen wird.
Insgesamt ist diese Methode also **nicht empfehlenswert**. Sie wurde stellenweise im originalen Bootloader verwendet.

Aktive Verwendung des Instruktionsspeichers Man kann dem Prozessorcontroller mitteilen, dass bestimmte Speicherbereiche in den Instruktionsspeicher vorgeladen werden sollen und dort nicht verdrängt werden, bis dies explizit aufgehoben wird.
Dies erlaubt es auch längeren Kode vorzuladen. Da die Länge der vom Compiler generierten Maschinenbefehle bestimmbar ist, kann somit der gesamte Kode in C geschrieben werden.
Der Nachteil ist, dass man dazu sehr hardwarenahe Programmierung betreiben muss und es nur schwer nachvollziehbar ist, ob der Cache wirklich mit den richtigen Daten befüllt wird und diese Instruktionen darin fest verankert sind.

Ausführen von Kode aus dem RAM Der kritische Kode wird hierbei in den RAM Speicher ausgelagert und von dort ausgeführt.
Der Vorteil liegt darin, dass man nicht auf die Verwendung der Caches angewiesen ist und eine große Menge an Kode verwenden kann. Ein Nachteil dagegen ist, dass der Kode dann Speicherbereiche belegt, die für Variablen, dynamischen Speicher und Stack vorgesehen sind.
Diese Lösung erfordert es, den entsprechenden Kode so zu markieren, dass er in den RAM vorgeladen wird, was mit einem geeigneten GCC Attribut und einer Anpassung des Linkerskripts geschieht. Allerdings sind somit zwei Kodebereiche vorhanden: Ein Teil liegt auf dem Flashspeicher, also in den ersten 32 MiB des gesamten Adressraums. Der andere Teil liegt im RAM, der im Adressraum an der 1472 MiB-Grenze beginnt. Der Sprungbefehl des ARM-Instruktionssatzes ist allerdings auf eine Differenz von 32 MiB begrenzt, die zwischen der aktuellen Instruktion und der anzuspringenden Instruktion liegen darf. Somit lässt sich der Kode im RAM nicht direkt anspringen. Eine Lösung dazu besteht darin, die Adresse der anzuspringenden Instruktion in ein Register zu laden und den Kode über dieses Register anzuspringen.

Für diesen Umstand wurden verschiedene Makros implementiert, die den Sprung ausführen können und dabei aber auch auf Fehler hinweisen, wenn etwa falsch getypte Argumente an eine Funktion übergeben werden sollen.

Read-While-Write Der Flashspeicher unterstützt eine weitere Eigenschaft, die es erlaubt einzelne Flashbereiche in verschiedene Modi zu versetzen. Da der Bootloader derzeit in die erste Flashpartition passt, kann man die zweite programmieren, während der Kode des Bootloaders weiterhin problemlos gelesen und ausgeführt werden kann.
Kritisch sind somit nur Zugriffe, die auf die erste Partition gemacht werden, etwa das Sperren der Flashblöcke in denen der Bootloader liegt, sodass dieser nicht mehr Überschrieben werden kann, nachdem der Bootloader die Kontrolle an das Image abgibt.

In der finalen Implementierung wurden die entsprechenden Flashroutinen einerseits in den RAM ausgelagert, so wie es im vorhergehenden Abschnitt erläutert wurde. Während der Ausführung der Flashroutinen sind Interrupts weiterhin aktiviert, daher wird andererseits die Read-While-Write Möglichkeit verwendet, da ein auftretender Interrupt das Ausführen des Interrupthandlers, der im Flashspeicher liegt, veranlasst.

4.4 Anpassungen

Die Modularisierung des Bootloaders zeigt sich auch in den Dateistrukturen des Quelltexts. Der Bootloader kann somit einfach um weitere Funktionalitäten erweitert werden, ohne dass dazu große Anpassungen am bestehenden Kode nötig sind. Die Modularität soll nun anhand einiger Beispiele in der Übertragung, dem Zielspeichermedium und der Portierung auf eine andere Plattform verdeutlicht werden.

4.4.1 Übertragung

Die Datenübertragung wurde für die USB-Schnittstelle konkretisiert, jedoch können auch weitere Schnittstellen der Imote-Plattform für die Datenübertragung verwendet werden. Beispiele hierfür wären eine der seriellen Schnittstellen oder auch eine drahtlose Übertragung über den CC2420 Chip.
Über das unterliegende Kommunikationsprotokoll werden einige Annahmen vorausgesetzt. Zuerst wird eine ausreichende Sicherung der Daten durch eine Prüfsumme angenommen. Die CRC-16 Prüfsumme, die vom USB-Protokoll und auch vom IEEE 802.15.4 Standard verwendet wird, deckt nach der Bitfiltertheorie [25] jede ungerade Anzahl von Bitfehlern, insbesondere also auch 1-Bit Fehler auf. Weiter werden Bündelfehler bis zur Länge 16 und bei der maximalen Paket- bzw. Rahmengröße von USB bzw. IEEE 802.15.4 auch 2-Bit Fehler sicher erkannt. Diese Abdeckung ist für die übertragenen Daten hinreichend. Weiter wird vorausgesetzt, dass die Daten in Sendereihenfolge empfangen werden und auch verlustfrei übertragen werden. Die Daten können allerdings beliebig fragmentiert sein, da das Kommunikationssystem des Bootloaders Paketgrenzen erkennt und die Daten als Einheit an

die höhere Schicht weiterleitet.
Wird eine dieser Annahmen nicht durch die gewählte Kommunikationsart abgedeckt, so muss sie beim Implementieren nachgebildet werden. Soll also die Funkschnittstelle verwendet werden, sollte auf jeden Fall eine Verlusterkennung und -behandlung stattfinden.

Die Kommunikation selbst wird in der Quelldatei `communication_buffer.c` behandelt. Sie enthält eine Methode, die beim Empfangen von Daten aufgerufen wird und eine Methode, um Daten zu versenden. Diese Methoden müssen dann weiter angepasst werden, um den Eigenschaften des unterliegenden Protokolls zu genügen.

4.4.2 Speichermedien

Als Speichermedium wurde der Flashspeicher des Imotes angenommen. Es ist jedoch auch denkbar, Teile der Programme auf externen Speicherkarten wie SD, MMC oder Memory Stick auszulagern und auch diese über den Bootloader zu programmieren. Das Programmieren selbst geschieht in der Datei `communication.c` und muss für weitere Speichermedien entsprechend dort erweitert werden.

4.4.3 Portierung

Der Kode des Bootloaders orientiert sich an den Fähigkeiten und Registerdefinitionen der Imote-Plattform. Teile des Kodes, wie beispielsweise das Übertragungsprotokoll wurden jedoch plattformunabhängig ausgelegt und weitere Teile, wie die Behandlung des Endpunkt 0 Datenstroms sind in ihrer Art plattformübergreifend und können somit wiederverwendet werden.
Somit ist es möglich, den Bootloader auf andere Plattformen zu portieren, indem man die entsprechenden hardwarenahen Teile der Zielplattform anpasst.

4.5 Vergleich

Im Abschnitt 2.1 wurde der von Junaith Ahemed Shabadeen entwickelte TinyOS-Bootloader und einige seiner Nachteile erwähnt. Im Folgenden werden nun die Vorzüge des entwickelten Bootloaders aufgezeigt, wobei auch nochmal auf die gestellten Anforderungen an den Bootloader eingegangen wird.

Freie Adresswahl ist durch das entwickelte Übertragungsprotokoll gewährleistet. Der TinyOS-Bootloader schreibt das geladene Programm immer an die Startadresse des Flashspeichers, was nicht beeinflusst werden kann.

Maximalgröße des Programms lag u.A. durch einen Designfehler beim TinyOS-Bootloader bei 1 MiB. Wird diese Grenze überschritten, können Teile des Bootloaders überschrieben werden, wodurch der Imote unbrauchbar wird und nur durch Neuprogrammieren des Bootloaders über die JTAG-Schnittstelle wieder

verwendet werden kann.
Der entwickelte Bootloader dagegen beschränkt die hochgeladenen Daten nur durch die Größe des Flashspeichers, abzüglich der 4 MiB Untergrenze.

Programmiergeschwindigkeit Das Übertragen einer kleinen Applikation (etwa 500 KiB) dauert mit dem entwickelten Bootloader etwa 20 Sekunden, wovon die reine Übertragungs- und Programmierzeit etwa 5 Sekunden ausmacht. Den größten Teil der Zeit belegt dabei das Löschen der Flashblöcke.
Der TinyOS-Bootloader nimmt für die selbe Applikation über die dreifache Zeit in Anspruch. Dies ist vor allem auf die geänderte Übertragungs- und Programmierart zurückzuführen, da die Eigenschaften des Flashspeichers für die Programmierung besser ausgenutzt wurden. Die Löschzeit ist in beiden Bootloadern von gleicher Dauer anzusehen, daher ist bei größeren Images mit einem noch größeren Zeitvorteil zu rechnen.

Kommunikation Die Kommunikation des TinyOS-Bootloaders verwendet das Human Interface Device (HID) [31] Protokoll, das für Eingabegeräte über USB entwickelt wurde. Daher ist auch Interrupttransfer als Transferart fest vorgegeben. Sowohl das Protokoll als auch die Transferart sind nicht dazu geeignet, die zu programmierenden Imagedaten zu übertragen. Vor allem der Protokolloverhead, der durch die Kapselung in das HID Protokoll zustande kommt, schränkt auch die Geschwindigkeit der Übertragung weiter ein und durch die Verwendung des Protokolls wird kein Vorteil erreicht.

Energiemanagement Der entwickelte Bootloader deaktiviert die CPU, wenn keine weiteren Aufgaben mehr anstehen. Dies wurde besonders in Hinsicht auf ein kabelloses Programmieren in Verbindung mit einer portablen Energieversorgung implementiert. Während der Übertragung der Imagedaten und dem Programmieren auf den Flash wird die CPU andererseits hochgetaktet, um eine erhöhte Geschwindigkeit zu erzielen. Dies muss natürlich bei Verwendung einer portablen Energieversorgung überdacht werden.

Kodequalität Der Kode des entwickelten Bootloaders ist mit Doxygen [16] Kommentaren beschrieben, sodass die Dokumentation der einzelnen Routinen leicht einsehbar ist. Doxygen ist ein Werkzeug, das aus strukturierten Kommentaren eine übersichtliche Dokumentation des Kodes erstellen kann. Weiter wurde der Kode so strukturiert und untergliedert, dass sich ein logischer Aufbau der einzelnen Teile in der Dateistruktur wiederfindet.
Der Kode des TinyOS-Bootloader ist dagegen zwar dokumentiert, jedoch ist die gesamte Funktionsweise des Bootloaders auch nach Lesen der Kommentare noch nicht verständlich. Eine Strukturierung des Kodes lässt sich nur schwer beschreiben und die Verwendung von Assemblerkode und fehlende Kommentare machen den Kode stellenweise undurchsichtig und nur schwer verständlich.

Fortschrittsanzeige Die LEDs der Plattform wurden dazu genutzt, den Fortschritt der Programmierung anzuzeigen. Der TinyOS-Bootloader macht zwar auch von den Leuchtdioden Gebrauch, jedoch erhält man darüber keine Rückmeldung in einem möglichen Fehlerfall.

Kapitel 5

Kodelader

Der Kodelader bezeichnet allgemein ein Programm, welches die Gegenstelle der Kommunikation zum Übertragen eines Images auf einen Zielknoten übernimmt. Er baut die Kommunikation mit dem zu programmierenden Gerät auf, tauscht die erforderlichen Parameter zur Programmierung des Images aus und überträgt schließlich das Image.
Dies kann zum einen ein PC-Programm sein, welches die USB-Schnittstelle nutzt, um ein Image zu übertragen. Zum anderen kann es sich aber auch um ein auf einem zweiten Imote ausgeführtes Programm handeln, welches die drahtlose Schnittstelle der Imotes zur Übertragung nutzt.

5.1 Übertragungsprotokoll

Das Übertragungsprotokoll aus Abschnitt 4.1.3 erfordert eine Implementierung auf der Kodeladerseite. In Abb. 5.1 ist der Zustandsautomat für den Kodelader gezeigt. Die Implementierung selbst gestaltet sich anders als auf Bootloaderseite, da im wesentlichen nur die einzelnen Signale zum Übertragen der Imagedaten gesendet werden und nach jedem Signal der Statuskode abgewartet wird. Im Falle eines positiven Statuskodes wird das nächste Signal gesendet, ein negativer Statuskode beendet den Kodelader mit einer entsprechenden Fehlermeldung.
Beim Start des Kodeladers sendet dieser ein `Reset`-Signal, das den Kommunikationsautomaten auf Seite des Bootloaders in den `IDLE`-Zustand versetzt. Somit ist ein definierter Zustand erreicht und der Kodelader kann den die eigentliche Übertragung des Images einleiten. Das `Addr`-Signal enthält die Zieladresse und die Länge des hochzuladenden Images. Nach positiver Quittierung folgt das `Erase`-Signal, das das Löschen der entsprechenden Flashblöcke einleitet. Nachdem auch dieses Signal positiv quittiert ist, kann der eigentliche Übertragungsvorgang einsetzen, was durch blockweises Senden des Images in `Upload`-Signalen geschieht. Sind alle Daten übertragen, wird der Vorgang mit einem `UploadFinish`-Signal abgeschlossen. Danach befinden sich beide Automaten wieder im `IDLE`-Zustand und es kann eine weitere Übertragung begonnen werden. Sind keine weiteren Übertragungen mehr gewünscht, kann der Imote mit Hilfe des `Reboot`-Signals neu gestartet werden.

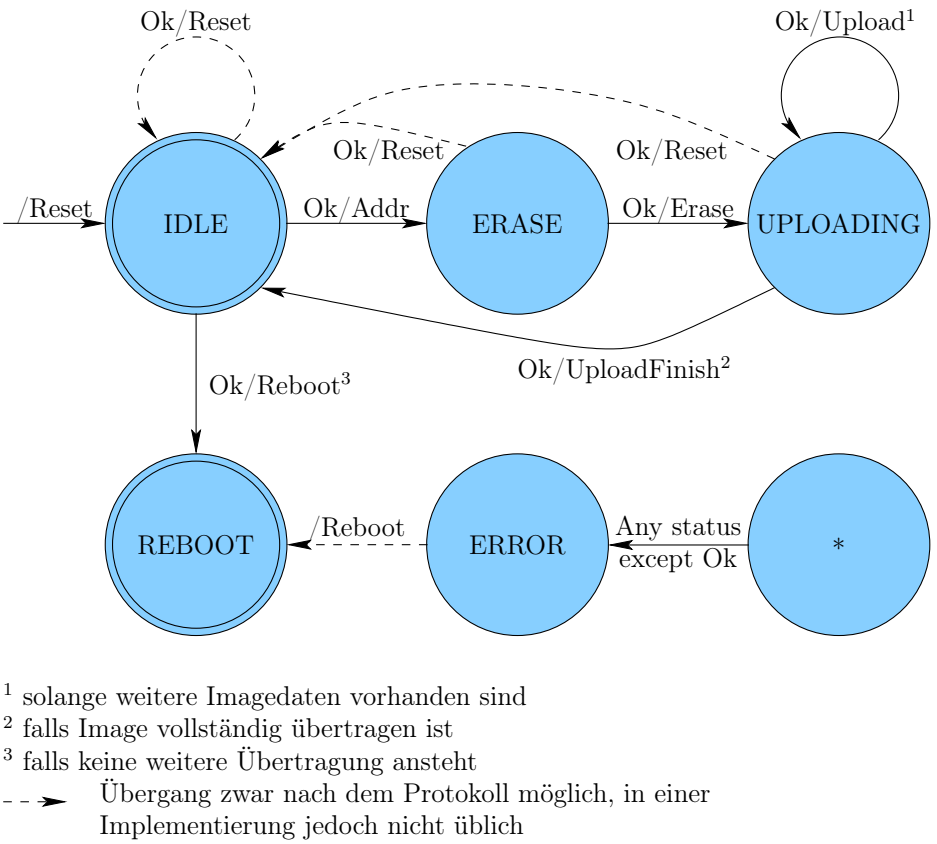

Abbildung 5.1: Übertragungsprotokoll auf Kodeladerseite als Zustandsautomat

5.2 Übertragung vom PC

Die Übertragung eines Images auf den Imote wird über die USB-Schnittstelle des PCs realisiert. Die *libusb* [8] wurde verwendet, um diese Schnittstelle möglichst einfach anzusprechen. Die Bibliothek bietet eine Abstraktion des Betriebssystems, sodass USB-Endgeräte unter Linux, BSD Varianten, Mac OS X und Windows-Systemen auf die gleiche Weise angesprochen werden können. Der Kodelader wurde in der Programmiersprache Java realisiert, um weitere Unabhängigkeit vom Betriebssystem zu erlangen.

Der Programmaufbau gliedert sich in Pakete und Klassen, die einzelne Aufgaben der Kommunikation übernehmen. Ein Teil davon realisiert das Kodieren der einzelnen Signale und ihrer zugehörigen Daten in Pakete und das Dekodieren der ankommenden Statuskodes in Datenstrukturen, die einen einfachen Zugriff auf die Daten bieten.
Eine andere Klasse ist für die USB-Kommunikation verantwortlich: In ihr wird das entsprechende Gerät gesucht und eine Verbindung aufgebaut. Weiter werden darüber die Pakete mit Hilfe der *libusb* versendet und empfangen. Wird ein Statuskode empfangen, wird geprüft, ob es sich um eine positive Quittierung oder einen Fehlerfall handelt. Im Fehlerfall wird eine entsprechende Ausnahmebehandlung eingeleitet, welche dem Benutzer eine aussagekräftige Fehlermeldung ausgibt und den Kodela-

der beendet.
Die Hauptklasse implementiert den eigentlichen Protokollablauf, indem es die genannten Klassen zur Kommunikation nutzt.

Konfiguration Die Hauptaufgabe des Kodeladers besteht darin, eine Datei auf den Imote zu übertragen. Auf Imoteseite werden die Daten an eine vom Benutzer gewählte Stelle des Flashspeichers geschrieben. Dazu muss dem Kodelader der Dateiname des zu übertragenden Images und die Zieladresse innerhalb des Flashspeichers mitgeteilt werden.
Dem Kodelader werden diese Daten per Kommandoparameter übergeben. Alternativ kann auch eine Konfigurationsdatei verwendet werden, die die Angaben enthält. Die Verwendung einer Konfigurationsdatei hat den Vorteil, dass mehrere Images mit ihren jeweiligen Zieladressen angegeben werden können. So kann ein Image den auszuführenden Programmkode und die anderen beispielsweise Konfigurationsdaten enthalten. Die einzelnen Teile werden dann sequentiell durch nur eine Ausführung des Kodeladers übertragen.
Weitere Konfigurationsangaben beziehen sich auf den Zielknoten selbst. So kann zusätzlich die Seriennummer des Imotes angegeben werden. Ein versehentliches Überschreiben des falschen Knoten ist damit ausgeschlossen. Diese Option bietet sich vor allem bei Verwendung einer Konfigurationsdatei an, da somit die Konfiguration an einen bestimmten Zielknoten gebunden wird.

5.3 Übertragung vom Imote

Der Bootloader kann, wie in Abschnitt 4.4.1 beschrieben, so erweitert werden, dass die Übertragung eines Images auch über die drahtlose Schnittstelle des Imotes erfolgen kann. So kann ein Imote verwendet werden, um einem weiteren Imote ein neues Image zu übertragen. Der sendende Imote übernimmt dabei Rolle des Kodeladers. Als Quelle des zu übertragenden Images sind verschiedene Szenarien denkbar: Beispielsweise kann das Image auf dem Flashspeicher des sendenden Imotes abgelegt sein, oder es kann das eigene Image als Zielimage versendet werden. Alternativ kann das Image auch von einem PC über die USB- oder serielle Schnittstelle des sendenden Imotes und von diesem weiter über die drahtlose Schnittstelle versendet werden. Dazu muss ein weiteres Protokoll zwischen dem MAC-Layer der IEEE 802.15.4 Basistechnologie und dem Kommunikationsautomat realisiert werden, das die fehlenden Eigenschaften des MAC-Protokolls ausgleicht, die vom Kommunikationsautomaten angenommen werden. Eine ausreichende Absicherung ist durch das MAC-Protokoll gegeben und, sofern kein Routing eingesetzt wird, kann die korrekte Empfangsreihenfolge ebenfalls angenommen werden. Anders verhält es sich bei Verlusterkennung und -behandlung. Diese muss mit einem geeigneten Protokoll nachgebildet werden. Ein solches Protokoll muss natürlich auf Seite des Bootloaders und auch auf Seite des sendenden Imotes für jeweils beide Richtungen des Datentransfers implementiert werden. Das eigentliche Übertragungsprotokoll kann unverändert über dem zusätzlichen Protokoll verwendet werden.

Kapitel 6

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein Bootloader für die Imote2 Plattform vorgestellt. Der Bootloader wurde entwickelt, um ein Image auf der Imote2 Plattform auszuführen oder ein neues Image über die USB-Schnittstelle der Plattform entgegenzunehmen. Für die Übertragung eines Images wurde ein Kommunikationsprotokoll entwickelt, über das die entsprechenden Signale ausgetauscht werden. Der Einsatz der USB-Schnittstelle erforderte die Entwicklung eines USB-Subsystems, das so ausgelegt wurde, dass es als Treiber in anderen Softwareprojekten wie dem SDL Environment Framework wiederverwendet werden kann.
Ein robuster und flexibler Bootloader und das zugehörige Kommunikationsprotokoll muss folgende Eigenschaften erfüllen, die in den Entwurf und die Implementierung eingeflossen sind: Der Bootloader überprüft die **Vollständigkeit eines Images**, bevor es ausgeführt wird. So können abgebrochene Übertragungen erkannt und ein nicht definiertes Verhalten verhindert werden, das auftritt, wenn ein unvollständige Image ausgeführt wird. Außerdem wird der **Fortschritt der Übertragung** anhand der LEDs des Imotes angezeigt. In Fehlerfällen liefern zusätzliche Ausgaben der seriellen Schnittstelle eine genauere Angabe des Zustands. Das **Programmieren beliebiger Adressen** ermöglicht es, ein Image in mehrere Teile aufzuspalten. Eine solche Aufteilung erlaubt die Verwendung von Konfigurationen, sodass verschiedene Ausprägungen des Images möglich sind, ohne dazu den Programmkode anpassen zu müssen. Der **modulare Aufbau** des Programmkodes des Bootloaders erlaubt einfache Erweiterungen. Es kann beispielsweise die vorhandene USB-Kommunikation um eine drahtlose Kommunikation erweitert werden, ohne tiefgehende Anpassungen im bestehenden Kode zu machen.
Das Schreiben effizienter Routinen zur Flashprogrammierung, die Verwendung von Techniken zur Geschwindigkeitsoptimierung während der Übertragung, und nicht zuletzt das Übertragungsprotokoll selbst trugen dazu bei, dass die Übertragung eines Images mit dem entwickelten Bootloader eine **hohe Übertragungsgeschwindigkeit** erzielt. Dies ist vor allem während der Entwicklung eines Images vorteilhaft, da durch einen kurzen Programmiervorgang störende Wartezeiten verkürzt werden.
Der Einsatz von *Doxygen* [16] erzeugt eine gute **Dokumentation**, die eine Weiterentwicklung erleichtert.

Im zweiten Teil der Arbeit wurde ein Kodelader für den Bootloader entwickelt. Der Kodelader wird auf dem PC ausgeführt und kommuniziert über eine USB-

Schnittstelle mit dem Bootloader. Er lässt sich mit Hilfe von **Konfigurationsdateien** komfortabel steuern. Der Kodelader wurde in der Programmiersprache Java implementiert und verwendet **betriebssystemunabhängige** Bibliotheken zur Kommunikation mit der USB-Schnittstelle. Dies führt dazu, dass der Kodelader auf vielen Plattformen, für die eine Implementierung der Java Virtual Machine existiert, ausgeführt werden kann.

In einer Weiterentwicklung des Bootloaders soll das Übertragen eines Images über die drahtlose Schnittstelle des Imotes realisiert werden. So kann ein Imote angewiesen werden, sein Image auf einen Imote in Funkreichweite zu replizieren. Werden die Images mit einer Kennung und Version versehen, können die Imotes rekursiv das Image immer an einen Knoten weitergeben, der in sich in Reichweite befindet. Auf diese Weise können mehrere Knoten ohne Benutzerinteraktion auf den selben Kodestand gebracht werden. Die Realisierung einer solchen Funktionalität ist zwar trivial, wurde jedoch aufgrund mangelnder Zeit nicht mehr in die Implementierung aufgenommen.

Die Aufteilung des Images ermöglicht es, den eigentlichen Programmkode getrennt von Daten, wie beispielsweise Konfigurationen, auf den Imote zu übertragen. Der Programmkode selbst kann jedoch ebenfalls aus mehreren Teilen bestehen, die beim Übersetzen im letzten Schritt durch den Linker zusammengefügt werden. Beim Übersetzen eines SDL-Systems wird beispielsweise die SDL-Spezifikation mit den Treibern des SDL Environment Frameworks (SEnF) [17], der SDL-Laufzeitumgebung SdlRE [18] und der C-Bibliothek µClibc [10] zusammengesetzt. Eine Trennung dieser Teile in das eigentliche System einerseits und die Umgebung mit den Bibliotheken andererseits hätte den Vorteil, dass diese auch getrennt ausgetauscht und übertragen werden können. Die Realisierung dieser Trennung erfordert jedoch auch, dass ein Teil des Linkervorgangs erst vorgenommen wird, wenn die Teile auf dem Imote zusammengefügt werden. Dazu ist ein definiertes Format der Imageteile nötig, die neben dem Programmkode auch die Symboltabellen enthalten, anhand derer die Schnittstellen der einzelnen Teile zusammengeführt werden. Dieser Vorgang wird durch einen Linker realisiert, der im Bootloader zu integrieren ist. Insgesamt ist diese Funktionalität sehr schwergewichtig und wurde daher nicht realisiert.

Der Bootloader selbst wird derzeit mit Hilfe der JTAG-Schnittstelle auf den Imote übertragen. Die erforderliche Hard- und Software könnte eingespart werden, da der TinyOS-Bootloader auf dem Imote vorinstalliert ist. Der TinyOS-Bootloader kann einfach verwendet werden, um den neuen Bootloader zu programmieren und somit den TinyOS-Bootloader durch den neuen zu ersetzen. Unter Ausnutzung dieser Möglichkeit kann der hier entwickelte Bootloader ohne Einsatz einer JTAG-Schnittstelle auf den Imote übertragen werden. Somit entfällt auch der Bedarf der JTAG-Hardware. Die Realisierung dieser Funktion ist im Anschluss an diese Arbeit geplant.
Der Bootloader selbst soll mit einer neueren Version überschrieben werden können. So entfällt der Einsatz der JTAG-Schnittstelle komplett. Allerdings muss dabei beachtet werden, dass der Kode, der den Flashspeicher überschreibt, selbst Teil des Bootloaders ist und somit theoretisch überschrieben wird. Dies führt im allgemeinen Fall zu einem nicht definierten Verhalten, was natürlich verhindert werden muss. Eine

Möglichkeit zur Lösung besteht darin, den Kode, wie in Abschnitt 4.3.1 beschrieben, in den Arbeitsspeicher auszulagern und von dort auszuführen.

Anhang A

OpenOCD Konfiguration und Bedienung

In diesem Kapitel soll eine kurze Übersicht über den Open On-Chip Debugger gegeben werden.
Alle Informationen beziehen sich auf die SVN Revision 1189.

A.1 Übersicht

OpenOCD ist ein Programm, welches einerseits über eine bestimmte Hardware und dem JTAG-Protokoll mit einer Zielplattform kommuniziert und andererseits Benutzerschnittstellen zur Verfügung stellt, um der Plattform die gewünschten Befehle zu geben. Somit läuft OpenOCD im Hintergrund als Daemonprozess.
Die Konfiguration von OpenOCD und die Bedienung der bedeutsamsten Schnittstellen werden in den folgenden Abschnitten vorgestellt.

A.2 Voraussetzungen

Um OpenOCD mit der hier verwendeten Hardware (Amontec JTAGKey-Tiny) zu betreiben, wird die FTDI Bibliothek benötigt. Es gibt zwei Alternativen dieser Bibliothek:

- **libftd2xx** ist eine Bibliothek, die direkt von der Firma FTDI bereitgestellt wird
- **libftdi** ist eine von der Firma intra2net bereitgestellte Bibliothek, von der auch der Quelltext offengelegt ist

Im folgenden wird von der Verwendung der libftdi ausgegangen. Erstellt man OpenOCD aus dem Quelltext, so lautet der Konfigurationsaufruf dafür `./configure --enable-ft2232_libftdi`.

A.3 Konfiguration

Im folgenden wird eine Beispielkonfiguration gezeigt, die auf der OpenOCD Konfiguration von TinyOS[1] aufbaut.
Weitere Informationen können der OpenOCD Konfiguration[2] entnommen werden.

```
#ports
telnet_port     3333
gdb_port        4444

#interface
interface ft2232
ft2232_layout jtagkey
ft2232_vid_pid 0x0403 0xcff8
jtag_speed 0

#set the jtag_ntrst_delay to the delay introduced
# by a reset circuit the rest of the needed delays
# are built into the openocd program
jtag_ntrst_delay 0

#use combined on interfaces or targets that can't
# set TRST/SRST separately
reset_config trst_and_srst separate

#jtag scan chain
# <Length> <IR Capture> <IR Capture Mask> <IDCODE>
jtag_device 7 0x1 0x7f 0x7e

#target configuration
target xscale little 0 pxa27x

#map to PXA internal RAM
working_area 0 0x5c000000 0x10000 nobackup

#flash bank <driver> <base> <size> <chip_width> <bus_width>
flash bank cfi 0x00000000 0x2000000 2 2 0

init

reset halt
```

[1] http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x-contrib/intelmote2/tools/platforms/intelmote2/openocd/arm-usb-tiny.cfg?view=markup

[2] http://openocd.berlios.de/doc/openocd.pdf

A.4 Starten von OpenOCD

Wie bereits im vorherigen Abschnitt angemerkt, ist OpenOCD ein Prozess, welcher im Hintergrund läuft. Vor dem Starten sollte man sicherstellen, dass die Hardware angeschlossen und eingeschaltet ist. Des weiteren benötigt man die erforderlichen Leserechte auf `/dev/bus/usb/` und Schreibrechte auf die entsprechende Datei in einem dort liegenden Unterverzeichnis. Der Befehl `lsusb -d 0403:ccf8` liefert dabei beispielsweise die Information

```
Bus 003 Device 002: ID 0403:cff8 Future Technology Devices [...]
```

welche nun auf den Zugriffspfad `/dev/bus/usb/003` und die darin enthaltene Gerätedatei `002` verweist.

Zum Starten verwendet man nun folgenden Befehl:

```
openocd -f <konfigdatei>
```

In manchen, nicht näher untersuchten Fällen ergibt sich das Problem, dass OpenOCD Fehlermeldungen wie die folgende ausgibt:

```
Error:  JTAG communication failure, check connection, [...]
```

In diesem Fall hilft es häufig, OpenOCD abzubrechen (`STRG+C`) und neu zu starten. Hilft auch dies nicht, sollte man die Hardware nochmals neu anschließen und den OpenOCD Daemon neustarten.

A.5 Telnet Schnittstelle

Über ein gewöhnliches Telnet Programm kann man sich mit dem OpenOCD Prozess verbinden. Der Port ist über die Konfiguration einstellbar und standardmäßig auf Port 3333 voreingestellt. Da die Kommunikation somit über Netzwerk geschieht, kann der OpenOCD Prozess auch über einen entfernten Rechner verbunden werden.

Verbinden der Schnittstelle Läuft der Prozess auf dem selben Rechner, lautet der Aufruf

```
telnet localhost 3333
```

Man erhält dann eine Eingabeaufforderung, in der man dem Daemon die gewünschten Befehle mitteilen kann. Im Folgenden werden nun die wichtigsten Befehle des OpenOCD Daemons genannt.

Allgemeine Befehle

- `shutdown`
  Beendet den OpenOCD Prozess

- `script` *`Datei`*
  führt die Befehle, die in der angegebenen Datei enthalten sind aus. Der Dateiname ist dabei relativ zu dem Pfad anzugeben, in welchem der OpenOCD Prozess gestartet wurde.

- `reset [run|halt]`
  startet die Zielplattform neu. Wenn der Parameter `halt` angegeben wurde, wird die Plattform nach dem Neustart direkt angehalten

- `halt`
  Hält die aktuelle Ausführung an

- `resume`
  setzt die Ausführung fort

Speicherbefehle

- `mdw` *`Adresse Länge`*
  Ließt die gegebene Adresse wortweise aus.
  Vorsicht: In OpenOCD tritt unter der gegebenen Hardwarekonfiguration ein Fehler auf, wenn eine bestimmte Größe (etwa 64 Worte) überschritten wird

- `reg`
  ließt die Registerinhalte des Prozessors aus

Flash Operationen

- `flash info 0`
  ließt Informationen über den Flash aus. Dieser Befehl eignet sich auch dazu festzustellen an welchen Adressen die einzelnen Flashblöcke beginnen

- `flash protect 0` *`Anfang Ende`* `off`
  entsperrt die Flashblöcke, die zwischen *Anfang* und *Ende* liegen.
  Dieser Vorgang ist nötig bevor Flashblöcke gelöscht oder beschrieben werden können.
  Anmerkung: Mit der gegebenen Hardwarekonfiguration ist es nur möglich die ersten 20 Flashblöcke zu entsperren, sonst tritt in OpenOCD ein Fehler auf.

- `flash erase_sector 0` *`Anfang Ende`*
  löscht die Flashblöcke zwischen *Anfang* und *Ende*

- `flash write_bank 0` *`Datei Anfang`*
  schreibt den Inhalt der angegebenen Datei in den angegebenen Flashblock, gegebenenfalls blockübergreifend in weitere Blöcke. Die Datei wird dabei relativ zu dem Pfad, in dem OpenOCD gestartet wurde, angegeben.
  Dazu müssen die Blöcke entsperrt und gelöscht sein.

A.6 GDB Schnittstelle

Der GNU Debugger GDB kann mit einem speziellen Protokoll auch entfernte Prozesse verbinden und steuern. Dieses Protokoll wird auch von OpenOCD unterstützt und ist ebenfalls über Netzwerk erreichbar. Standardmäßig ist der Port 4444 dafür voreingestellt, kann aber wie auch der Telnet-Port in der Konfiguration angepasst werden.

Zuerst startet man dazu den GDB, der für die Zielplattform compiliert wurde:

```
xscale-elf-gdb <Programm>
```

Der Platzhalter *Programm* wird dabei durch den Dateinamen des zu analysierenden Programms ersetzt.

Zum Verbinden gibt man GDB nun folgenden Befehl ein:

```
target remote localhost:4444
```

GDB unterbricht dann die Ausführung und kann nun wie gewohnt bedient werden.

Verwenden von Breakpoints GDB erlaubt es, so genannte Breakpoints zu verwenden, bei deren Erreichen die Kodeausführung unterbrochen wird. Mit der gegebenen Hardwarekonfiguration sollte man sich jedoch auf einen Breakpoint beschränken, da es sonst zu Fehlern in OpenOCD kommt.

Debuggen eines SDL Systems Ein vorhandenes SDL System kann ebenfalls mit Hilfe von GDB analysiert werden. Vor dem Verbinden mit OpenOCD gibt man dazu folgenden Befehl in GDB an:

```
add-symbol-file <datei> 0x00400000
```

Die Dateiangabe bezieht sich dabei auf das zu analysierende Image und ist relativ zu dem Pfad, in dem GDB gestartet wurde. Die nachfolgende Zahl gibt an, wo sich das Textsegment der Anwendung befindet.

Anhang B

Konfiguration, Übersetzung und Installation des Bootloaders

Der Bootloader kann mit mehreren Parametern vor dem Übersetzungsvorgang gesteuert werden. Der nachfolgende Übersetzungsschritt geniert dann den Programmkode, der auf den Imote ausgeführt werden soll. Dieser Programmkode muss abschließend auf den Imote übertragen werden. In diesem Kapitel sollen die erforderlichen Arbeitsschritte zum Konfigurieren, Übersetzen und Installieren des Bootloaders aus dem Quelltext beschrieben werden.

B.1 Konfiguration

Der Programmkode des Bootloaders bietet mehrere Stellen, an denen eine Konfiguration vorgenommen werden kann. Das entwickelte USB-Subsystem wird durch Angabe diverser Parameter vorkonfiguriert, die den Leistungsumfang und Speicherverbrauch des Subsystems beeinflussen. Weiter kann der Bootloader selbst durch die Konfiguration verschiedener Parameter Änderungen an der Umgebung angepasst werden. Zuletzt kann durch die Angabe verschiedener Optionen beim Übersetzungsprozess die Feinheit der Debugausgaben gesteuert werden.
Die Konfiguration des USB-Subsystems ist in der Datei `usb/usb_config.h` zusammengetragen. Alle weiteren Parameter sind in der Datei `config.h` zusammengefasst.

Konfiguration des USB-Subsystems Das USB-Subsystem ist vollständig im Unterordner `src/usb` enthalten und besteht wie in Abschnitt 4.2.1 beschrieben aus mehreren Teilen, die sich auch in der Dateistruktur wiederfinden. Die statische Konfiguration des Subsystems ist in der Datei `usb_config.h` geregelt.

Konfiguration der Kommunikation Die Parameter der Kommunikation steuern das Protokollformat, wie es in Abschnitt 4.1.3.4 vorgestellt wurde. Diese Parameter müssen natürlich mit den Definitionen im Kodelader vereinheitlicht werden.

Konfiguration der Debug-Schnittstelle Debugausgaben können mit Hilfe einer formatierenden `debugf()` Routine erstellt werden. Die Länge dieser Ausgaben ist durch die Konfiguration begrenzt. Ein höherer Wert hat einen erhöhten Speicherverbrauch zur Folge.

Konfiguration des Flashspeichers Der Flashspeicher des Imotes ist mit einigen Parametern vorkonfiguriert, die den Programmiervorgang beeinflussen. Da diese sich auf die Beschaffenheit des Flashspeichers beziehen, sind diese im Normalfall nicht zu ändern.

Konfiguration des Verhaltens Über weitere Parameter, die in der Datei enthalten sind kann das allgemeine Verhalten des Bootloaders angepasst werden. Dazu gehören Parameter wie die Einsprungadresse, die mit dem Linkerskript abzugleichen sind, unter dem die auszuführenden Images entstanden. Weiter können Wartezeiten und das Energiemanagement des Bootloaders konfiguriert werden.

B.2 Übersetzen des Bootloaders

Der Programmkode des Bootloaders wird mit Hilfe eines Makefiles übersetzt. Das Makefile enthält Variablen, die das generieren von Debugausgaben und den Übersetzungsvorgang steuern. Da je nach Subsystem des Bootloaders viele Debugausgaben generiert werden können und diese Ausgaben dann eine merkbare Prozessorlast verursachen, wird dadurch die Performanz des Bootloaders beschränkt. Daher kann man die Ausgaben der verschiedenen Subsysteme bereits beim Übersetzungsvorgang aktivieren oder deaktivieren.

B.2.1 Konfiguration

Neben den erwähnten Optionen zur Generierung der Debugausgaben gibt es noch weitere Optionen, die den Übersetzungsvorgang steuern. Die Parameter werden im folgenden genannt.

- `ARM_BARE` gibt den Ort des `arm_bare` Verzeichnisses innerhalb des SDL Environment Framework Projekts an.
- `RELEASE` regelt die Optimierung des Compilers. Ist die Option eingeschaltet, so wird effizienterer Kode erzeugt, was zu einer performanteren Ausführung des Kodes auf dem Imote führt.
- `DEBUG` gibt an, ob überhaupt Debugausgaben generiert werden sollen.
- `DEBUG_TRACE` gibt an, ob Debugausgaben generiert werden, die die Instrumentierung der Ausführung ermöglichen. Bei aktivierter Option werden bei jedem Funktionsein- und austritt Ausgaben übermittelt, mit Hilfe derer nachvollzogen werden kann, welche Routine von einer anderen ausgeführt wurden.

- `DEBUG_MEMORY` gibt an, ob Debugausgaben beim Anfordern und Freigeben von Speicher ausgegeben werden sollen.
- `DEBUG_FLASH` regelt, ob das Verwenden der Lösch- und Schreibroutinen Debugausgaben generieren sollen.
- `DEBUG_USB` gibt an, ob das USB-Subsystem Debugausgaben generiert.
- `DEBUG_DMA` gibt an, ob das DMA-Subsystem Debugausgaben generiert.
- `DEBUG_COMM` gibt an, ob das Kommunikationssystem Debugausgaben generiert.
- `USE_DMA` regelt, ob DMA oder Interrupt-Transfer verwendet werden soll.
- `OPENOCD_HOST` gibt den Namen des Hosts an, auf dem der OpenOCD Daemon gestartet wurde.

B.2.2 Ziele

Ein einfacher Aufruf von `make` ohne Angabe von Zielen erstellt die Ziele `out`, `out.bin` und `out.sym`. Die Datei `out` ist das Ergebnis des Übersetzungs- und Linkervorgangs. In dieser Datei sind noch Programmheader wie Symboltabellen und Debuginformationen enthalten, die für die Ausführung auf dem Imote nicht mehr benötigt werden. Die Datei `out.bin` enthält ausschließlich den fertigen Programmkode, der aus der Datei `out` extrahiert wird. Zum Debuggen wird noch die Symboltabelle benötigt, die in der Datei `out.sym` enthalten ist.

Mit Hilfe des Ziels `install` kann das fertige Programm, also die Datei `out.bin` auf den Imote mit Hilfe von OpenOCD übertragen werden. Dazu muss sichergestellt sein, dass OpenOCD die Datei in dem Verzeichnis vorfindet, wo es gestartet wurde. Das Verwenden symbolischer Links ist an dieser Stelle zu empfehlen.

Der Programmkode wurde mit Hilfe von Doxygen, einem Werkzeug, welches automatisch Dokumentationen generiert, dokumentiert. Um die Dokumentation zu generieren stehen zwei Ziele zur Verfügung. Das Ziel `int_doc` erstellt die gesamte Dokumentation des Bootloaders. Dabei werden auch Routinen und Variablen dokumentiert, die nach außen hin nicht sichtbar sind. Dies ist dann Hilfreich, wenn beispielsweise das USB-Subsystem weiterentwickelt wird. Das Ziel `doc` erstellt dagegen nur die Dokumentation der sichtbaren Routinen, also beispielsweise der öffentlichen Schnittstellen des USB-Subsystems.

Das Ziel `clean` löscht alle generierten Dateien.

B.3 Installation

Nach Übersetzung des Bootloaders muss die Datei `out.bin` in den Flashspeicher des Imotes übertragen werden. Dafür wurde in dieser Arbeit die JTAG-Schnittstelle mit Hilfe der Software OpenOCD verwendet. Für diese Aufgabe kann das Ziel `install` des Makefiles verwendet werden, welches dem OpenOCD Host die erforderlichen Befehle übergibt. Diese sind auch im Anhang A beschrieben.

B.4 Debug

Das Debugsystem bietet neben der Ausgabe von Textnachrichten auch die Möglichkeit, Funktionsaufrufe zu verfolgen. Normale Textnachrichten können mit Hilfe der Routinen `debug` und `debugf` versendet werden. Das Verfolgen der Funktionsaufrufe wird über die Aktivierung der Option `DEBUG_TRACE` im Makefile eingeschaltet. Dies veranlasst den Compiler während der Kodegenerierung bei jedem Funktionsein- und austritt die Instrumentierungsroutine `__cyg_profile_func_enter` bzw. `__cyg_profile_func_exit` aufgerufen. Diesen Routinen wird die Symboladresse der gerade betretenen bzw. verlassenen Funktion übergeben. Einzelne Funktionen, die nicht instrumentiert werden sollen, können unter Angabe des GCC-Attributs `__attribute__((no_instrument_function))` in ihrer Deklaration ausgenommen werden. Diese Symboladresse wird dann mit einer Markierung über die serielle Schnittstelle übermittelt.

Das Perlskript `addr.pl` im Verzeichnis des Bootloaders kann nun dazu genutzt werden, die Symboladressen in die Symbolnamen zu übersetzen. Dazu ließt das Skript beim Start die Symboltabelle aus `out.sym` ein, die beim Übersetzen des Bootloaders generiert wird. Dies bedeutet auch, dass das Skript nach einem erneuten Übersetzen neu gestartet werden muss, um die aktualisierte Symboltabelle einlesen zu können. Das Skript selbst wird mit dem Befehl `perl addr.pl < /dev/tts/USB1` ausgeführt, wobei der Pfad zur seriellen Schnittstelle natürlich entsprechend angepasst werden muss. Bevor auf diese zugegriffen werden kann, muss auch die Übertragungsgeschwindigkeit angeglichen werden. Auf Seite des Bootloaders ist sie mit 115200 Kbit/s festgelegt. Der Aufruf des Kommandos `stty -F /dev/tts/USB1 speed 115200` auf PC-Seite setzt diese Geschwindigkeit für die serielle Schnittstelle des PCs fest.

Literaturverzeichnis

[1] *ARMboot*. http://armboot.sourceforge.net/.

[2] *blob, a StrongARM boot loader*. http://sourceforge.net/projects/blob/.

[3] *Cygwin, a Linux-Like environment for Windows*. http://www.cygwin.com/.

[4] *Das U-Boot*. http://www.denx.de/wiki/U-Boot.

[5] *GCC, the GNU Compiler Collection*. http://gcc.gnu.org/.

[6] *GDB: The GNU Project Debugger*. http://www.gnu.org/software/gdb/.

[7] *GNU Binutils*. http://www.gnu.org/software/binutils/.

[8] *libusb*. http://libusb.wiki.sourceforge.net/.

[9] *LibUsb-Win32*. http://libusb-win32.sourceforge.net/.

[10] *µClibc*. http://www.uclibc.org/.

[11] *Open RFID reader*. http://www.openpcd.org/.

[12] *Openmoko*. http://wiki.openmoko.org/wiki/Main_Page.

[13] Amontec: *Amontec JTAGkey-Tiny*. http://www.amontec.com/jtagkey-tiny.shtml.

[14] Crossbow: *IIB2400 Imote2 Interface Board*. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/IIB2400_Datasheet.pdf.

[15] Crossbow: *Imote2 High-Performance Wireless Sensor Network Node*. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Imote2_Datasheet.pdf.

[16] Doxygen: *Doxygen, a source code documentation generator tool*. http://www.doxygen.org/, 2007.

[17] Fliege, I., A. Geraldy, S. Jung, T. Kuhn, C. Webel und C. Weber: *Konzept und Struktur des SDL Environment Frameworks (SEnF)*. Techn. Ber. 341/05, TU Kaiserslautern, 2005.

[18] Fliege, I., R. Grammes und C. Weber: *ConTraST - A Configurable SDL Transpiler and Runtime Environment*. In: Gotzhein, R. und R. Reed (Hrsg.): *System Analysis and Modeling: Language Profiles*, Bd. 4320 d. Reihe *Lecture Notes in Computer Science*, S. 216–228, 2006.

[19] Instruments, T.: *Texas Instruments SmartRF CC2420*. http://focus.ti.com/lit/ds/symlink/cc2420.pdf, März 2007.

[20] Intel: *Intel PXA27x Processor Family Developer's Manual*. http://int.xscale-freak.com/XSDoc/PXA27X/2800002.pdf, Oktober 2004.

[21] Intel: *Intel PXA27x Processor Family Memory Subsystem*. http://int.xscale-freak.com/XSDoc/PXA27X/30185501.pdf, 2004.

[22] Intel: *Intel XScale Core Developer's Manual*. http://download.intel.com/design/intelxscale/27347302.pdf, Januar 2004.

[23] Intel: *Intel PXA27x Processor Family Electrical, Mechanical and Thermal Specification Data Sheet*. http://int.xscale-freak.com/XSDoc/PXA27X/28000304.pdf, 2005.

[24] Intel: *Intel PXA27x Processor Family Specification Update*. http://int.xscale-freak.com/XSDoc/PXA27X/28007106.pdf, Mai 2005.

[25] Kowalk, W.: *CRC Cyclic Redundancy Check, Analyseverfahren mit Bitfiltern*. http://einstein.informatik.uni-oldenburg.de/forschung/crc/Bitfilter-Lange%20Version.pdf.

[26] libusbjava: *Java libusb/libusb-win32 wrapper*. http://libusbjava.sourceforge.net/wp/.

[27] OpenOCD: *Open On-Chip Debugger*. http://openocd.berlios.de/doc/About.html.

[28] Schmelzer, T.: *Integration des Imote2 in das SDL Environment Framework (SEnF)*. Projektarbeit, Technische Universität Kaiserslautern, Oktober 2007.

[29] Schmelzer, T.: *Entwicklung und Integration von Services und Gatewayfunktionalität auf Basis von AmICoM*. Diplomarbeit, Technische Universität Kaiserslautern, Mai 2008.

[30] Shahabdeen, J. A.: *Boot Loader Architecture*. http://tinyos.cvs.sourceforge.net/viewvc/*checkout*/tinyos/tinyos-1.x/contrib/imote2/tools/src/bootloader.doc?revision=1.1, 2005.

[31] USB IF: *Device Class Definition for Human Interface Devices (HID)*. http://www.usb.org/developers/devclass_docs/HID1_11.pdf, 2001.

[32] USB IF: *Device Class Speicification for Device Firmware Upgrade 1.1*. http://www.usb.org/developers/devclass_docs/DFU_1.1.pdf, 2004.

[33] USB IF: *Defined 1.0 Class Codes*. http://www.usb.org/developers/defined_class, März 2006.

[34] USB IF: *Universal Serial Bus Specification 2.0*. http://www.usb.org/developers/docs/usb_20_122208.zip, 2008.

[35] WASABI: *Wasabi Software Development Tools User's Guide for Intel XScale Microarchitecture*. http://download.intel.com/design/intelxscale/dev_tools/031121/Wasabi_XScale_Users_Guide1.pdf, März 2004.