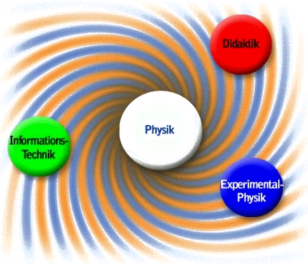
UNIVERSITÄT KOBLENZ · LANDAU
FACHBERICH INFORMATIK



Erstellung einer Softwarebibliothek zum Betrieb von Microcontrollern der Baureihe Atmel AT90USB als USB-Hostcontroller

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers im Studiengang Informatik

vorgelegt von:
Michael Fogel
Thomas Wilbert

Betreuer:
Prof. Dr. Christoph Steigner
Dr. Merten Joost

Koblenz, im März 2007

Zusammenfassung

In dieser Arbeit wird eine Softwarebibliothek zur Nutzung des USB-Hostmodus von Mikrocontrollern der Baureihe AT90USB entwickelt. Die Eigenschaften des USB werden erläutert und darauf aufbauend die Hardware des verwendeten Mikrocontrollers beschrieben. Anschließend wird die entwickelte Software und deren Funktionsweise erläutert. Abschließend werden Treiber für diverse Geräteklassen vorgestellt, die auch dem Test der entwickelten Bibliothek dienen.

Erklärungen

Hiermit versichere ich, Michael Fogel (Matrikelnr. 201210278), die Abschnitte 2.1, 2.1.1.1, 2.1.4, 2.2, 2.4, 2.4.1, 2.4.2, 2.4.3.4–2.4.3.11, 2.4.5, 2.4.5.1, 2.4.5.2, 2.4.6, 3.5, 3.6.1–3.6.4, 4.1–4.3, 4.5.5–4.5.9, 4.6 sowie 4.6.1–4.6.3 der vorliegenden Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, den _____

_____
Michael Fogel

Hiermit versichere ich, Thomas Wilbert (Matrikelnr. 201210251), die Abschnitte 2.1.1.2, 2.1.2, 2.1.2.2, 2.1.2.3, 2.1.3, 2.1.5, 2.1.5.1, 2.3, 2.4.3, 2.4.3.1–2.4.3.3, 2.4.3.12, 2.4.4, 2.4.5.3, 2.4.7, 2.5, 3, 3.1–3.4, 3.6.5, 3.6.6, 4.4, 4.5, 4.5.1–4.5.4, 4.5.10–4.5.12 sowie 4.6.4 der vorliegenden Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, den _____

_____
Thomas Wilbert

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Im Laufe der letzten Jahre macht sich eine immer weiter wachsende Technisierung unseres Alltagslebens bemerkbar. Sowohl im professionellen als auch im privaten Umfeld sind wir alle mehr und mehr von elektronisch gesteuerten Geräten umgeben, sei es nun aus der Notwendigkeit heraus, die Produktivität zu steigern oder einfach nur zur persönlichen Unterhaltung. Von Messeinrichtungen über Haushaltsgeräte bis zum Wiedergabegerät für Musik: In allen diesen Geräten arbeitet, meist vor dem Anwender gut verborgen, ein eigens für seine spezielle Aufgabe designter beziehungsweise programmierter Mikrochip.

Abhängig von der Anwendung kommt dabei eine Klasse von ICs besonders häufig zum Einsatz: die sogenannten Mikrocontroller, kurz auch als μC bezeichnet. Diese "Ein-Chip-Computersysteme" bieten sich heutzutage aufgrund ihrer Leistung und geringen Kosten für eine breite Palette von Steuer- und Regelungsaufgaben an. In der Lehre wird diesem Umstand meist mit einer Anzahl Veranstaltungen Rechnung getragen, die sich mit dem Umgang mit solchen Controllern beschäftigen. Einen Namen für kostengünstige und gut zu handhabende Mikrocontroller hat sich beispielsweise die Firma Atmel mit einer AVR genannten Baureihe gemacht. Im Fachbereich der Autoren werden diese Controller zur Zeit unter anderem zu Lehrzwecken genutzt.

Gleichzeitig mit ihrer Anzahl steigt auch das Bedürfnis, die verschiedenen, meist auch mobilen Geräte miteinander zu verbinden. Ständig gilt es, Daten von einem Gerät zum anderen zu übertragen. Eine Messeinrichtung, die ihre Daten nicht zur Weiterverarbeitung zugänglich machen kann, ist schließlich genauso unpraktisch, wie ein MP3-Player, der mangels Schnittstelle nicht mit der neuesten Musik bespielbar ist. Anschluss und Bedienung selbst sollen sich dabei natürlich möglichst schnell, komfortabel und ohne die Notwendigkeit einer eigenen Schnittstellenart für jedes neue Gerät gestalten. Eine breit unterstützte, sehr vielseitig einsetzbare Schnittstelle ist also

Abbildung 1.1: Einige Controller aus Atmels AVR-Serie

vonnöten, um Interoperabilität zu gewährleisten.

Als weit verbreiteter Standard zur Datenübertragung zwischen verschiedensten Geräten hat sich der Universal Serial Bus, kurz USB, durchgesetzt. Ursprünglich von Intel als Bussystem zum Anschluss von Peripheriegeräten an PCs entwickelt, ist mit wachsender Zahl der Embedded-Systeme auch eine direkte Verbindung zweier solcher Geräte wünschenswert.

Genau dort jedoch liegt zunächst ein Problem: Der USB-Standard unterscheidet zwei Gerätearten, im USB-Jargon "Device" und "Host" genannt. Grob umrissen handelt es sich bei Devices vornehmlich um einfache Endgeräte, wie zum Beispiel Mäuse, Tastaturen oder auch Speichersticks. Die von diesen Komponenten benötigten USB-Fähigkeiten lassen sich relativ einfach und kostengünstig herstellen. Solche Geräte sind jedoch nicht in der Lage, eigenständig untereinander eine Kommunikation über USB herzustellen. Der Benutzer bemerkt dies vor allem bei zunächst ganz simpel erscheinenden Vorhaben, wie dem Kopieren einer Musikdatei von einem MP3-Player zum anderen. Die beiden Geräte einfach mit einem geeigneten Kabel zu verbinden ist zwecklos – um die Datei wie gewünscht zu übertragen ist zusätzlich noch ein Gerät aus der Hostklasse notwendig.

Hostgeräte sind bei USB mit deutlich mehr Intelligenz ausgestattet, so dass sie den kompletten Datenverkehr auf dem Bus regeln können. Angehörige der Deviceklasse reagieren im Wesentlichen einfach nur auf Anfragen des Hosts. Geräte mit der Fähigkeit, als Busmaster zu agieren, müssen somit – sowohl was die Hardware als auch was die Software angeht – deutlich höhere Anforderungen erfüllen, was sich letztlich auch in höheren Kosten niederschlägt. Im Allgemeinen übernimmt ein PC die Rolle des Hosts, ganz so, wie es ursprünglich auch von Intel angedacht war.

Im Normalfall ist diese Einschränkung nicht weiter hinderlich, doch wie verhält es sich, wenn USB-Devices im Rahmen eigener Schaltungen als Peripherie eines Mikrocontrollers genutzt werden sollen? Ein solches Vorgehen würde – nicht zuletzt wegen des breiten Angebots an günstigen USB-Gerä-

Abbildung 1.2: Beispiele typischer, bekannter USB-Devices. Links: Webcam, liefert Videodaten per USB an den PC; Mitte: MP3-Player, kann per USB mit neuen Musikdateien versorgt werden; Rechts: Maus, ein über USB angeschlossenes Eingabegerät.

ten – eine Vielzahl neuer Möglichkeiten eröffnen. Von der einfachen Speichererweiterung mit Hilfe eines Speichersticks bis zum Anschluss von Webcams sind der Anwendung kaum Grenzen gesetzt.

Leider gestaltet sich ein solches Vorhaben mit dem Gros der am Markt befindlichen μCs schwierig bis unmöglich. Zu hoch sind die Anforderungen, die die USB-Spezifikation ([Spek]) an einen Hostcontroller stellt, als dass sie beispielsweise mit den Standardmodellen der AVR-Controller erfüllt werden könnten. Sollen AVRs oder vergleichbare Mikrocontroller als Host eingesetzt werden, so ist zwangsläufig der Einsatz zusätzlicher Hardware notwendig.

Hierzu kann zum einen ein gesonderter Chip verbaut werden, der die Buskontrolle übernimmt. Zum anderen bieten spezielle Controller jedoch auch bereits von Haus aus eine hardwareseitige Unterstützung für den Hostmodus. Waren solche Geräte in den günstigen Serien bislang kaum zu finden, so bietet Atmel seit kurzem einige weitere Modelle der AVR-Reihe an, die mit den gewünschten Fähigkeiten ausgestattet sind: die Modelle der Reihe AT90USB. Diese erlauben es, bereits vorhandene Erfahrungen mit AVR-Controllern weiter einzusetzen, dabei jedoch gleichzeitig nicht mehr auf den Anschluss von USB-Geräten verzichten zu müssen.

Für den Betrieb der neuen Chips ist eine geeignete Software – optimalerweise in Form einer Bibliothek – notwendig, wie sie bisher nicht verfügbar ist. Mit der Erstellung einer geeigneten, freien Software stände dem mit der AT90USB-Baureihe arbeitenden Entwickler eine kostenfreie und arbeitserleichternde Basis für eigene, innovative Entwicklungen zur Verfügung.

1.1 Aufgabenstellung

Die Erstellung und Dokumentation einer solchen in C geschriebenen Softwarebibliothek, die einen möglichst einfachen Betrieb über USB angeschlossener Peripheriegeräte an Controllern der Familie AT90USB erlaubt, ist Ziel der vorliegenden Arbeit.

Zum Verständnis der Funktion sowie auch der Nutzung der Software wichtige technische Fakten sollen zusammengetragen und erläutert werden. Dies betrifft neben der Hardware der Mikrocontroller selbst auch die Funktionsweise des USB, da dessen Verständnis Grundvoraussetzung für ein sinnvolles weiteres Arbeiten ist. Bei der Beschreibung des USB kann jedoch auf die Behandlung des sogenannten Highspeed-Modus verzichtet werden, da die verwendete Mikrocontroller-Familie diesen Modus nicht unterstützt.

Nach der Beschaffung geeigneter Mikrocontroller ist zunächst ein Evaluationsboard zu entwerfen und herzustellen, welches die Verbindung zu USB-Devices auf rein physikalischer Ebene erlaubt. Die so aufgebaute Schaltung dient im weiteren Verlauf dann als Testplattform für die zu entwickelnde Software. Die Nutzung der Bibliothek soll zusätzlich anhand einiger Beispiele exemplarisch erklärt werden, um zukünftigen Anwendern den Einstieg zu erleichtern.

Über diese Aufgabenstellung hinaus haben es sich die Autoren zum Ziel gesetzt, den gleichzeitigen Betrieb mehrerer Devices zu erlauben. Hierzu wird zunächst bewertet, in wie fern ein solcher Betrieb technisch überhaupt möglich ist[1]. Entsprechend der so gewonnenen Erkenntnisse wird die Software sodann für den simultanen Betrieb mehrerer Geräte entworfen und implementiert.

1.2 Vorgehen

Um sich der Entwicklung der Bibliothek zu nähern und mit der technischen Seite von USB vertraut zu werden, wird zunächst eine Einführung in das Thema gegeben. Kapitel 2 erläutert dafür Schritt für Schritt die USB-Spezifikation und liefert somit wichtige Grundlagen, die zum Verständnis des Betriebes von USB-Geräten unabdingbar sind. Darauf folgend wird in Kapitel 3 die benutzte Hardware beschrieben. Die für die Entwicklung der Software relevanten Komponenten werden erläutert und auch der Entwurf des Evaluationsboards wird hier vorgestellt. Letztlich lässt sich mit diesem Wissen dann

[1] Tatsächlich ergeben sich hierfür im weiteren Verlauf mit den zur Zeit aktuellen AT90USB-Controllern hardwareseitig einige Einschränkungen, die im Rahmen dieser Arbeit nicht zu beheben sind.

die Software verstehen, deren Beschreibung in Kapitel 4 folgt. Abschließend werden die in dieser Arbeit erreichten Ziele kurz zusammengefasst und ein Rückblick auf die Entwicklung gegeben.

Kapitel 2

Der Universal Serial Bus

2.1 Grundlagen und Begriffe

Der Universal Serial Bus (USB) ist ein 1996 von Intel eingeführtes Bussystem zur Kommunikation zwischen PC und diversen Peripheriegeräten. Ziel war es, die bereits bestehenden Schnittstellen durch eine universelle Schnittstelle zu ersetzen, um dem Kabelsalat auf dem Schreibtisch ein Ende zu bereiten.

Bis zur Einführung des USB war es bei IBM-kompatiblen PCs üblich, jedes Peripheriegerät mit einer dedizierten Schnittstelle direkt am Rechner zu verbinden. Dies brachte zum einen den Nachteil mit sich, dass für jedes Gerät eben auch eine solche Schnittstelle am PC vorhanden sein musste. Zum anderen verlief auf diese Weise aber natürlich auch für jedes kabelgebundene Gerät eine eigene Leitung zum PC. Wollte der Nutzer beispielsweise zwei Drucker anschließen, so ergab sich bei den meisten Rechnern das Problem, dass nur eine parallele Schnittstelle zur Verfügung stand[1] und somit beide Drucker nicht ohne weiteres gleichzeitig betrieben werden konnten.

Außerdem gestaltete sich der Anschluss neuer Geräte umständlich. Vor jedem An- oder Abstecken eines Peripheriegerätes musste der PC ausgeschaltet werden, um nicht das Risiko einer Beschädigung der elektronischen Bauteile einzugehen. Zusätzlich waren die Schnittstellen bei der Mehrheit der Rechner auf dessen Rückseite zu finden, so dass sie für den Anwender nur schlecht zu erreichen waren.

Mitunter waren die Schnittstellen für verschiedene Geräte auch noch mechanisch kompatibel, ein bekanntes Beispiel sind hier auch heute noch die PS2-Anschlüsse für Tastatur und Maus. Wird auf diese Weise ein Gerät mit einem ungeeigneten Port verbunden, so kann es im schlimmsten Fall zur Zer-

[1] Dies war die bis dahin gebräuchliche Schnittstelle zum Anschluss von Druckern an IBM-kompatible PCs.

störung des Peripheriegerätes oder Rechners kommen.

Wünschenswert war also ein einziges, erweiterbares System, durch das nahezu beliebige externe Geräte mit nur einem einzigen Anschluss am PC verbunden werden konnten. Es galt also, die Direktverbindungen, welche durch einzelne Kabel zwischen jedem Gerät und dem Rechner hergestellt wurden, durch ein Bussystem, an welches alle Geräte Anschluss finden können, zu ersetzen. Die Computer des Herstellers Apple boten damals (mit einigen Einschränkungen) schon ein ähnliches Konzept, welches den Anschluß von mehreren Eingabegeräten an nur eine Buchse des Computers erlaubte. Dieses System wurde damals unter dem Namen "Apple Desktop Bus" (ADB) vertrieben.

Die USB-Entwickler begannen also mit der Konzeption eines neuen Bussystems und griffen dabei einige Grundideen von ADB auf. Um die Anzahl der benötigten Adern der Anschlusskabel gering zu halten und Probleme mit unterschiedlichen Signallaufzeiten zu vermeiden wurde, wie bei ADB auch, eine bitserielle Übertragung gewählt. Auch die später noch beschriebene Klasseneinteilung von USB-Geräten stammt in ihren Grundzügen von ADB ab. Die USB-Entwickler verbesserten jedoch natürlich große Teile entscheidend. Um beim Beispiel der Übertragung zu bleiben kommt dort aus Gründen der Störsicherheit eine differentielle Form zum Einsatz, welche bei ADB nicht vorgesehen war.

Aus Gründen der Anwendbarkeit legten die Entwickler Wert darauf, dass USB-Geräte zur Laufzeit an den Rechner anschließbar sind. Für den Anwender muss ferner garantiert sein, dass ein Gerät, welches er in den Anschluss für die neu konzipierte Schnittstelle einsteckt, an dieser auch einwandfrei seine Funktion aufnimmt. Auch die Stromversorgung des Gerätes kann im Regelfall direkt über das USB-Kabel erfolgen. Eventuell weitere notwendige Konfigurationen, wie sie vor der Einführung von USB für jedes Gerät oder dessen Treiber notwendig waren, sollten nach dem Willen der Entwickler so weit wie irgend möglich entfallen und automatisiert werden. Dies würde endlich ein tatsächlich funktionsfähiges "Plug and Play" System ermöglichen, wie es schon oft vorher propagiert worden war.

Ein weiterer wesentlicher Punkt, den die Entwickler beachten mussten, betraf die Kosten für das neue Anschlusssystem: Die Herstellung und Entwicklung von Geräten muss so einfach und preisgünstig wie möglich erfolgen können. Ein typisches System, wie es sich bei den Anwendern im Einsatz befindet, besteht aus einem PC mit einer Vielzahl von Peripheriegeräten. Aus kosten- und entwicklungstechnischen Gründen ist es daher sinnvoll, den Aufbau der Geräte so einfach wie möglich zu halten. Der Hauptaufwand, welcher sich durch die neue Schnittstelle ergibt, kann vom Rechner beziehungsweise

einer dessen interner Komponenten getragen werden. Diese ist nur einmal im System vorhanden, so dass hier ein aufgrund der komplexeren Aufgabe höherer Preis vertretbar ist.

Das USB-System unterscheidet daher zwischen dem sogenannten "Host" – also der besagten im PC befindlichen Komponente – und den einfachen Peripheriegeräten.

Peripheriegeräte erfüllen jeweils eine bestimmte Funktion für den Nutzer. Eine Maus liefert beispielsweise Eingabedaten für die Position des Mauszeigers, eine Kamera liefert einen Videodatenstrom oder auch vorher geschossene Bilder als JPEG-Dateien oder ähnliches. Der Host fragt Daten im Bedarfsfall von den Geräten ab und diese reagieren ausschließlich auf die Anweisungen des Hosts. Insbesondere heißt das, dass kein Gerät eigenmächtig schreibend auf den Bus zugreift. Somit entfallen aufwändige Arbitrierungstechniken auf Seite der Geräte, was deren Konstruktion erheblich erleichtert. Solche Aufgaben werden dann vollständig durch den Host übernommen.

Dieses Verfahren bietet jedoch noch einen weitern ganz entscheidenden Vorteil: Da der Host die Kontrolle über den kompletten Datenverkehr auf dem Bus hat, ist er dazu in der Lage, mit jedem Gerät genaue Vorgaben über dessen Datenlieferung auszuhandeln. Beispielsweise kann er einem Eingabegeräte garantieren, dessen Daten in bestimmten immer gleichen Intervallen abzufragen. Gleichzeitig ist es aber auch möglich, Videogeräten eine bestimmte Übertragungsgeschwindigkeit zu garantieren, so dass in deren Videostrom keine durch den Anwender zu bemerkenden "Aussetzer" auftreten.

Zusätzlich kann davon ausgegangen werden, dass bestimmte Geräte eine wesentlich höhere Bandbreitenanforderung an den Bus stellen als andere. So benötigt ein Eingabegerät wie die Maus eine deutlich niedrigere Datenrate als dies bei der Videoanwendung der Fall ist. Die Entwickler sahen daher bei USB die Möglichkeit vor, verschiedene Geräte mit verschiedenen Datenraten zu betreiben. In den Standards 1.0 sowie 1.1 sind dabei der so genannte Lowspeed-Modus mit 1,5Mbit/s sowie ein mit Fullspeed bezeichneter Modus mit 12Mbit/s definiert. Geräte, welche nur den Lowspeed-Modus nutzen, können Signale auf dem Bus beispielsweise mit einer geringeren Abtastrate erfassen als solche, die im Fullspeed-Modus operieren. Somit lassen sich die Kosten der Geräte weiter minimieren.

Mit dem im Jahr 2000 verabschiedeten USB-2.0 Standard fügten die Entwickler dem Bus eine weitere Geschwindigkeitsoption hinzu. Ein "Highspeed"-Modus ermöglicht dort die Datenübertragung mit bis zu 480Mbit/s. Damit war endgültig auch eine effiziente Handhabung von externen Massenspeichern und dergleichen möglich.

USB hat sich seit seiner Einführung aufgrund seiner eindeutigen Vorteile

gegenüber den bis dahin genutzten Schnittstellen weit verbreitet. Zwischenzeitlich haben andere Hersteller von Rechensystemen die Vorteile von USB erkannt. So dient der USB bei Produkten aus dem Hause Apple nun beispielsweise als Ersatz für den vorher verwendeten ADB. Aber auch bei Geräten, welche nicht direkt dem Umfeld eines Desktop-Rechners zuzuordnen sind, wird USB eingesetzt: Moderne Videospielkonsolen sind ein gutes Beispiel hierfür.

Im Laufe der Zeit wurde USB somit zu einem plattformübergreifenden Standard zum Anschluss von Peripheriegeräten an Rechensysteme aller Art.

Neben der eigentlichen Spezifikation zu USB-2.0, welche sich stark auf den Anschluss von Peripherie an den PC bezieht, existieren noch weiterführende Dokumente, die sich zum Beispiel mit dem Anschluss von USB-Geräten untereinander beschäftigen. Die sogenannte "On the Go"-Erweiterung (OTG) bietet diese Möglichkeit. Hierbei übernimmt ein eigentlich als Peripherie für den PC vorgesehenes Gerät die Aufgabe des Hosts. Somit kann zum Beispiel eine USB-Kamera direkt mit einem USB-Drucker verbunden werden, ohne dass der Anwender einen Computer benötigt. Um die Komplexität der Anforderung an das Gerät im Hostmodus zu verringern, muss dieses bei OTG nicht alle in der USB-2.0-Spezifikation angegebenen Anforderungen an einen Host unterstützen. Insbesondere die Möglichkeit zum Betrieb im Highspeed-Modus ist nicht gefordert[2].

Der AT90USB1287 ist ein solches OTG-Gerät. Er ist sowohl als einfaches USB-Gerät als auch in einem Modus betreibbar, welcher ihn als Busmaster arbeiten lässt. Gegenüber einem vollständig spezifikationskonformen Host sind dabei seine Fähigkeiten jedoch leicht eingeschränkt – das OTG-Supplement erlaubt wie erwähnt diverse Einschränkungen bei Nutzung von OTG-Geräten im Hostmodus.

Der AT90USB1287 unterstützt in beiden Anwendungsfällen ausschließlich den Fullspeed- und den Lowspeed-Modus. Der Highspeed-Modus ist nicht zugänglich.

Wie schon in der Aufgabenstellung festgehalten, beschränkt sich daher die weitere Detailbeschreibung von USB auf die Charakteristika der beiden unterstützten Modi.

Im Folgenden gilt es, den grundsätzlichen Aufbau und die Funktionsweise des USB zu erfassen, um mit dieser Kenntnis die weitere Entwicklungsarbeit leisten zu können.

[2]Siehe u.a. [Speh], Abschnitt 3.2

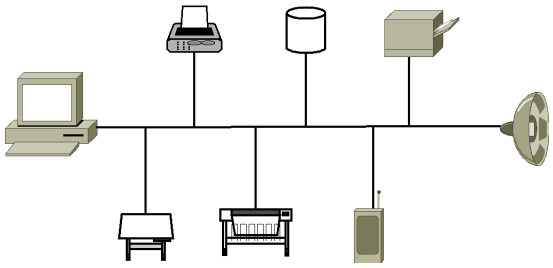


Abbildung 2.1: Typische Busstruktur

2.1.1 Topologie des USB

2.1.1.1 Physische Topologie

Üblicherweise stellt sich eine Busstruktur wie in Abbildung 2.1 gezeigt dar. Mehrere Geräte befinden sich physikalisch direkt in Verbindung mit einem gemeinsamen Medium, welches zur Signalübertragung genutzt wird. Auf diese Weise bildet sich eine einfache Strang-Topologie. Ein bekanntes Beispiel für eine solche Struktur ist das Ethernet über Koaxkabel, bei welchem Geräte direkt mittels T-Stücken oder gar per Schneidklemme ("Vampirklemme") an den Bus angeschlossen werden. Dort ist die physikalische Verbindung der Geräte zum Übertragungsmedium sehr offensichtlich.

Dieses Vorgehen bringt aber aus Sicht der Robustheit des Busses einen großen Nachteil mit sich. Erleidet einer der Busteilnehmer eine Fehlfunktion und bringt falsche Signale auf das Medium, so ist sofort der ganze Bus in Mitleidenschaft gezogen. Es ist dann keine Datenübertragung mehr möglich.

Für USB wurde daher eine modernere Topologievariante gewählt, bei der die verschiedenen Busteilnehmer elektrisch gesehen nur noch Punkt-zu-Punkt-Verbindungen eingehen – Um einen Bus handelt es bei einer solchen Struktur nur noch auf logischer Ebene. Ein ähnliches Vorgehen findet sich auch bei moderneren Ethernetvarianten.

Im Fall von USB entsteht so eine baumartige Struktur, in dessen Wurzel sich der Host befindet. Die einzelnen Verästelungen – also sternartige Anschlussstellen für zusätzliche Geräte – werden durch sogenannte Hubs erreicht. Diese dienen als Kontrollinstanz zwischen den einzelnen Ästen und leiten nur valide Signale aus einem Ast in den nächsten weiter.

Jeder Host besitzt nach der USB-Spezifikation einen sogenannten Root-Hub, welcher permanent direkt mit dem Host verbunden ist. Root-Hub-Logik und Host-Controller sind oftmals auf einem einzigen IC realisiert. Das Verhalten eines Root-Hubs ist jedoch – abgesehen von seiner untrennbaren

Verbindung zum Host – identisch mit dem üblicher externer Hubs.

An Hubs können weitere Hubs oder Endgeräte angesteckt werden, wobei USB den gleichzeitigen Betrieb von bis zu 127 Hubs und Geräten erlaubt. Jedoch erhöht sich mit jedem Hub, der zwischen ein Endgerät und den Host geschaltet ist, der Abstand zwischen Host und Endgerät in der Topologie. Geräte mit gleichem Abstand zum Host befinden sich auf einer sogenannten "Stufe" (engl. "Tier"), eine solche Topologie wird daher auch stufige Stern-Topologie genannt. Abbildung 2.2 zeigt das Schema eines solchen Aufbaus. Die USB-Spezifikation beschränkt die maximal verwendbare Tiertiefe auf sieben, wobei der Host (beziehungsweise dessen Root-Hub) als Stufe 1 gezählt wird.

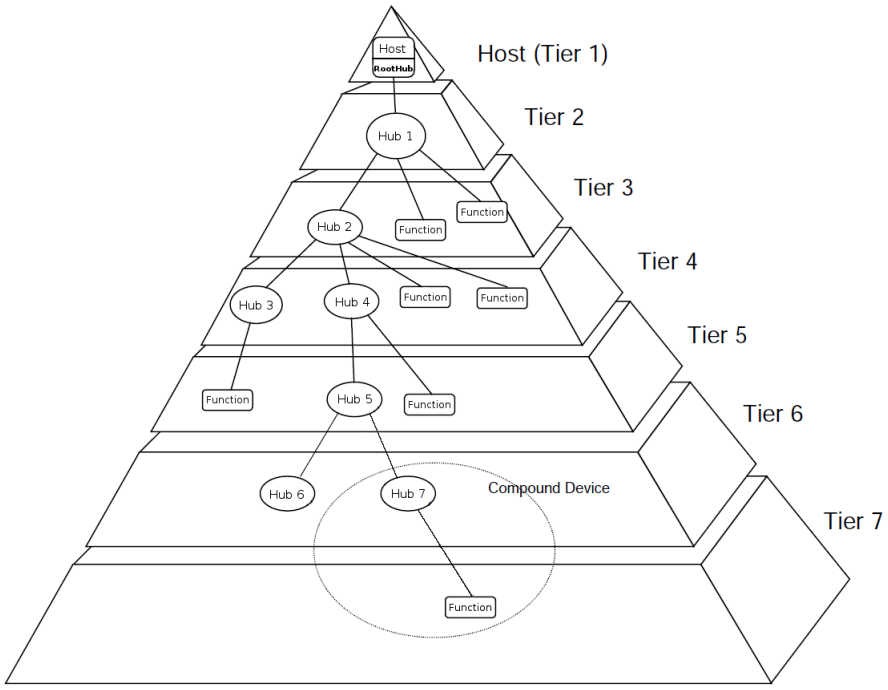


Abbildung 2.2: USB-Topologie (vergleiche [Spek]).

USB-Hubs finden sich heute zum einen als alleinstehende Geräte, zum anderen jedoch aber auch in einer Vielzahl von Peripherie, die auf einem moderenen Schreibtisch zu finden ist. Zusätzliche Anschlussmöglichkeiten für USB-Geräte existieren so an Monitoren, Tastaturen oder sonstigen Geräten, welche für den Anwender leicht erreichbar auf dem Tisch (und nicht mehr darunter) stehen.

2.1.1.2 Logische Topologie

Obwohl es sich bei der physischen Vernetzung der Geräte mit dem Host um eine stufige Stern-Topologie handelt, erscheinen alle Geräte aus logischer Sicht als an einen üblichen Bus (in Strang-Form) angeschlossen. Was den Kommunikationsprozess angeht, verhält sich USB für den Host so, als wären alle Geräte direkt mit ihm verbunden. Tatsächlich sind auch Hubs aus Sicht des Hosts einfache Geräte, die jedoch die Sonderfunktion des Anschlusses zusätzlicher Geräte mit sich bringen. Als eigenständige USB-Geräte nehmen Hubs an der regulären Buskommunikation teil. Der Host interagiert mit angeschlossenen Hubs und steuert so deren Funktion[3].

Der Host muss sich zu jedem Zeitpunkt über die physische Struktur des Busses bewusst sein, damit er geeignet auf das Entfernen von Geräten reagieren kann. Wird ein Hub aus der Topologie entfernt, so bedeutet dies implizit auch ein Entfernen aller über diesen Hub verbundenen weiteren Geräte. Abbildung 2.3 zeigt die logische Topologie aus Sicht des Hosts.

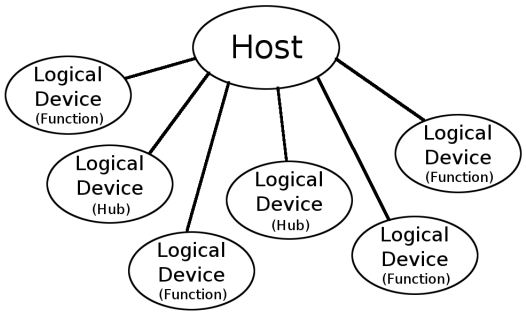


Abbildung 2.3: Logische Topologie des USB. Quelle [Spek].

Wie bei Bussen allgemein üblich werden angeschlossene Geräte mittels einer eindeutigen Adresse identifiziert. Die USB-Spezifikation sieht hierbei einen Adresspool von 128 Adressen vor (0–127), aus welchem den Geräten dynamisch eine Adresse zugeordnet wird. Wichtig hierbei ist, dass der Adresse Null besondere Bedeutung zukommt, da neu angeschlossene Geräte zunächst automatisch diese Adresse besitzen. Für einen dauerhaften Betrieb muss jedem Gerät eine der Adressen 1–127 zugeteilt werden – somit lassen sich maximal 127 Geräte gleichzeitig über USB betreiben.

[3]Die Funktion von Hubs wird näher in Abschnitt 2.4.7 beschrieben.

2.1.2 Schichtenmodell

Wird ein komplexes System entworfen, so kann nach dem Prinzip "teile und herrsche" vorgegangen werden. Nach der Aufteilung in kleinere Teilprobleme ist es einfacher, das Gesamtsystem zu überblicken und geeignete Lösungen für die Teilprobleme zu finden. Kommunikationssysteme sind daher üblicherweise in Schichten gegliedert, welche verschiedene, aufeinander aufbauende Funktionen bieten. Auch im Fall von USB wurde die gesamte Kommunikation zwischen Host und Gerät in solchen Schichten organisiert. Die Spezifikation sieht drei Schichten vor, Abbildung 2.4 zeigt das Ebenenmodell.

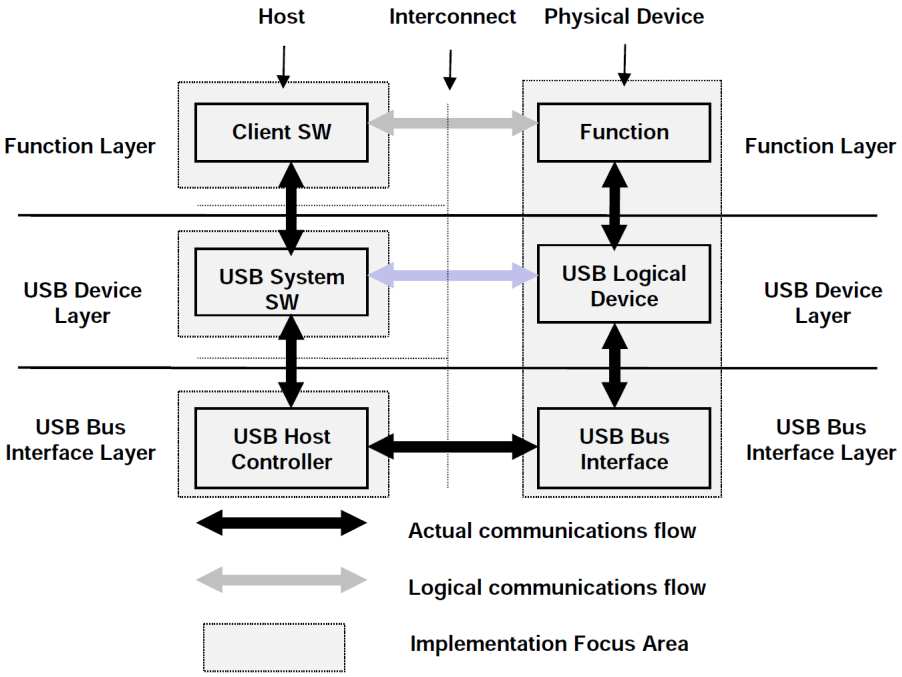


Abbildung 2.4: USB Schichtenmodell. Quelle [Spek]

2.1.2.1 Function Layer

Auf der obersten Ebene befindet sich auf Hostseite die auf eine Gerätefunktion zugreifende Client-Software. Dabei handelt es sich normalerweise um einen für das Gerät geeigneten Gerätetreiber. Dieser kommuniziert aus seiner Sicht direkt mit der für ihn relevanten Funktionseinheit des Gerätes. Ein Maustreiber kann sich beispielsweise direkt die Positionsdaten der durch ihn

bedienten Maus liefern lassen, ohne sich um die Details der dafür ablaufenden Datenübertragung kümmern zu müssen.

Da diese Schicht sich mit der Kommunikation zwischen Hostsoftware und konkreten Gerätefunktionen beschäftigt, wird sie in der Spezifikation mit *Function Layer* bezeichnet. Der Begriff *Function* steht dabei auch als Synonym für USB-Endgeräte im Allgemeinen. Er grenzt die Endgeräte von den Hubs ab, die ja keinerlei zugreifbare Funktion für eine Anwendungssoftware bieten.

Zu beachten ist, dass ein Endgerät durchaus auch mehrere Fähigkeiten zur gleichen Zeit bereitstellen kann. Jeder dieser Fähigkeiten kann dann ein eigener Treiber zugeordnet werden. Bei einer Kamera könnte zum Beispiel durch einen Treiber ein Live-Videostrom abgerufen und an eine Software zur Medienwiedergabe weitergereicht werden. Ein weiterer Treiber könnte aber auch einen Datenspeicher auf der Kamera ansprechen, um die dort befindlichen Daten einem Dateimanager (oder besser gesagt: einem Dateisystemtreiber) zur Verfügung zu stellen.

Endgeräte – also "Functions" – mit solchen, unabhängig voneinander nutzbaren Fähigkeiten werden *Composite Devices* genannt[4].

2.1.2.2 Device Layer

Unterhalb des Function Layers ist der Device Layer angesiedelt. Auf dieser Ebene kommuniziert die sogenannte USB-Systemsoftware mit den an den USB angeschlossenen Geräten. Die USB-Systemsoftware steuert dabei auf Hostseite die Konfiguration und Initialisierung der einzelnen Geräte. Dabeibei handelt es sich um einen von der eigentlichen Gerätefähigkeit unabhängigen Vorgang, der lediglich das Gerät selbst als Busteilnehmer einsatzfähig macht. Die USB-Systemsoftware ist üblicherweise Teil des Betriebssystems und verwaltet die einzelnen Kommunikationsvorgänge auf dem Bus. Insbesondere sorgt sie für eine korrekte Behandlung der in Abschnitt 2.1.3 beschrieben Transferarten, so dass geltende Zeit- und Bandbreitenbedingungen eingehalten werden.

Auf Seite des Hosts sieht die USB-Spezifikation sogenannte I/O-Request-Packets (IRP) als Schnittstelle für den Informationsaustausch zwischen dem Function- und Device Layer vor. Hierbei handelt es sich um Datenstrukturen, mit deren Hilfe die Client-Software der USB-Systemsoftware mitteilt, dass Daten mit einem USB-Gerät auszutauschen sind.

[4]Daneben kennt USB noch den Begriff des *Compound Device*, bei welchen einfach mehrere Functions fest mit einem Hub verbunden in einem einzelnen Gehäuse verbaut werden (beispielsweise Tastatur mit Touchpad). Logisch gesehen sind dies mehrere Geräte.

Die konkrete Implementation dieses Konzeptes ist Aufgabe des Betriebssystems, so verwalten beispielsweise sowohl Windows als auch Linux Übertragungswünsche in Form von Konstrukten namens "USB Request Block" (URB).

2.1.2.3 Bus Interface Layer

Als unterste Ebene tritt der Bus Interface Layer in Erscheinung. Hier befinden sich das Businterface des Hosts (Host-Controller) sowie auch das Businterface des USB-Gerätes. Diese beiden Komponenten tauschen nun tatsächlich Signale über eine physische Verbindung aus. Details über die physikalischen Vorgänge der Datenübermittlung folgen in Abschnitt 2.3.

2.1.3 Transferarten

Wie eingangs schon gesagt, können verschiedene Busteilnehmer bezüglich der Datenübertragung ganz unterschiedliche Anforderungen an den Bus stellen. Hierzu zählen Zeitgarantien für Echtzeitanwendungen, wie sie beispielsweise für Eingabegeräte interessant sind, aber auch Bandbreitengarantien für Streaminganwendungen und dergleichen.

Auch existieren verschiedene Anforderungen bezüglich der Verlässlichkeit der Übertragung: Während für die oben genannten Transfers ein eventueller Verlust eines Datums bei der Übertragung noch verschmerzt werden könnte, so existieren jedoch auch Anwendungen, bei denen eine vollständige und korrekte Übermittlung unbedingt erforderlich ist.

Bei der Streaminganwendung wird dem Anwender das Fehlen eines einzelnen Pixels für die Dauer eines Bildes wenig auffallen, ebenso verhält es sich wohl mit dem Verlust einiger weniger Audiodaten.

Im Gegensatz dazu wäre ein Verlorengehen von Daten bei einer Festplattenanbindung jedoch fatal. Der Einsatz eines externen Speichermediums stellt also ein Beispiel für die Gruppe der Anwendungen dar, welche eine korrekte und vollständige Datenübertragung benötigen.

Weiterhin ist es sinnvoll, eine spezielle Transferart für die Gerätekonfiguration zur Verfügung zu haben, auf welche Geräte garantiert zu jedem Zeitpunkt reagieren müssen. Im Fehlerfall genügt es dann für ein Gerät, nur noch solche Transfers zu behandeln. Auch hier muss natürlich die Integrität der Daten gewährleistet sein, damit es nicht zu Fehlkonfigurationen kommt.

Auf Basis dieser Überlegungen wurde der Datentransfer über USB konzipiert. Es existieren vier verschiedene Transferarten, welche sich im Wesentlichen durch die oben genannten Eigenschaften auszeichnen.

Control Transfers sind eine Transferart zur Übertragung wohldefinierter Daten, mit deren Hilfe unter anderem die Gerätekonfiguration vorgenommen wird.

Isochronous Transfers sind Transfers, für welche das Gerät mit dem Host eine Bandbreitengarantie ausgehandelt hat.

Interrupt Transfers ermöglichen es Geräten, dem Host eine Garantie bezüglich eines zeitlich konstanten Abfrageintervalls für Daten abzuverlangen.

Bulk Transfers schließlich dienen der gesicherten Übertragung von Daten, für welche jedoch keine weitere Latenz- oder Bandbreitengarantie gilt.

Diese Übersicht dient mehr der grundsätzlichen Einführung der Begrifflichkeiten, im Weiteren wird genauer auf die Details der Realisierung und der Übertragungsabläufe eingegangen.

2.1.4 Endpoints

Bei USB werden Daten jeweils zwischen Host und einem so genannten Endpoint – oder auch Endpunkt – eines Gerätes ausgetauscht[5]. Endpoints sind unverwechselbare Ein- und Austrittsstellen eines jeden über den Bus übertragenen Datums. Für jeden Endpoint existiert im Gerätespeicher ein entsprechender Bereich, der Daten einer Übertragung aufnehmen, beziehungsweise Daten für den Versand bereitstellen kann. Jeder Endpunkt ist dabei nur zu einer Simplexverbindung fähig: Es können immer nur entweder Daten vom Gerät zum Host (upstream) oder vom Host in Richtung Gerät (downstream) transferiert werden.

Ein Gerät kann mehrere Endpunkte besitzen, welche vollständig unabhängig voneinander arbeiten. Jedem Endpunkt des Geräts ist eine Nummer zugewiesen, wobei das Tupel aus Richtung und Nummer dann einen Endpunkt (bezogen auf das Gerät) eindeutig identifiziert. Aus Sicht des kompletten Bussystems ist das Tupel aus Geräteadresse, Endpunktnummer und Richtung des Datentransfers für jeden Endpunkt einzigartig.

Geräteendpunkte haben weitere Eigenschaften, welche für den Kommunikationsprozess von Wichtigkeit sind. Über jeden Endpunkt lassen sich Übertragungen mit genau einer Transferart abwickeln. Ferner besitzen die Busteilnehmer (beziehungsweise deren USB-Controller) immer nur eine begrenzte Menge Speicher, so dass der für jeden Endpunkt zur Verfügung stehende

[5]Im deutschen Sprachraum hat sich eine Übersetzung des Begiffs eingebürgert. Daher werden auch in der vorliegenden Arbeit beide Begriffe synonym verwendet.

Speicher begrenzt ist. Jeder Endpunkt kann daher nur eine gewisse maximale Datenmenge pro Übertragung aufnehmen.

Die USB-Spezifikation gibt jedem Fullspeed-Gerät die Möglichkeit, für die Kommunikation mit dem Host bis zu 16 Endpunkte jeder Übertragungsrichtung zur Verfügung zu stellen. Die Anzahl der Endpunkte, welche tatsächlich bei einem speziellen Gerät vorhanden sind, ist im Rahmen dieser Möglichkeiten jedoch frei vom Hersteller wählbar. Ein Entwickler kann ein bestimmtes Endgerät also durchaus mit beispielsweise zehn Upstream- und zwei Downstream-Endpoints ausstatten, ganz so, wie es der jeweilige Einsatzzweck des Gerätes erfordert. Für Lowspeed-Geräte begrenzt die Spezifikation die Zahl der möglichen Endpunkte jedoch auf zwei je Übertragungsrichtung.

2.1.4.1 Endpoint 0

Eine besondere Bedeutung kommt bei jedem Gerät den Endpunkten der Nummer Null zu. Jedes USB-Gerät besitzt zwei solcher Endpunkte, wobei einer als Upstream- und der andere als Downstream-Endpunkt ausgelegt ist. Im Gegensatz zu anderen Endpunkten dienen diese beiden Endpunkte als direkte Schnittstelle zwischen der Gerätefirmware und der USB-Systemsoftware.

Beide Endpunkte müssen dem Transfertyp “Control” zugeordnet und sofort nach dem Anschließen des Gerätes (beziehungsweise dessen Reset) kommunikationsbereit sein. Die tatsächlich frei nutzbare Endpunktanzahl je Richtung reduziert sich damit auf 15 bei Fullspeed- beziehungsweise nur einen bei Lowspeed-Geräten.

2.1.5 Pipes

Eine USB-Pipe stellt eine Beziehung zwischen der Host-Software und Geräteendpunkten her. Über Pipes können Daten zwischen Host und USB-Geräten ausgetauscht werden.

Grundsätzlich unterteilen sich Pipes in zwei Klassen: Message Pipes und Stream Pipes. Die Kommunikation über Message Pipes erfolgt bidirektional nach einem durch die USB-Spezifikation festgeschrieben Format, welches einem “Request/Data/Status”-Paradigma folgt. Hierbei stellt der Host zunächst eine (im Rahmen der Spezifikation zulässige) Anfrage an das Gerät. Daraufhin reagiert das Grät und kann – abhängig von der gestellten Anfrage – Daten mit dem Host austauschen. Abschließend folgt eine Statusübermittlung, mit deren Hilfe die Kommunikationsteilnehmer die korrekte Behandlung des Requests bestätigen.

Im Gegensatz zur Message Pipe ist der Stream Pipe keine feste Struktur zugeordnet. In die Klasse der Message Pipes fallen nach der USB-Spezifikation solche Pipes, welche Control Transfers behandeln. Die übrigen Transferarten werden über Stream Pipes abgewickelt.

Eine Pipe kann bidirektional (im Falle einer Message Pipe) oder unidirektional in Up- oder Downstreamrichtung (im Falle einer Stream Pipe) sein. Stream Pipes sind dabei immer mit genau einem Endpunkt beliebiger Richtung und Nummer assoziiert, während Message Pipes eine Verbindung zu genau zwei Endpunkten gleicher Nummer aber entgegengesetzter Richtung darstellen.

Die Eigenschaften einer Pipe werden noch weiter durch die Charakteristika der Endpunkte, mit welchen sie in Beziehung steht, beeinflusst. Hierzu gehört zum Beispiel die maximal mögliche Größe der Nutzdaten pro Übertragung (Payload). Zusätzlich kann ein Gerät gegenüber dem Host Wünsche bezüglich der für bestimmte Endpunkte nötigen Abfrageintervalle oder Übertragunsbandbreiten äußern. Einer zum Endpunkt hin geöffneten Pipe werden bei ihrer Erzeugung dann auch diese Eigenschaften zugewiesen.

2.1.5.1 Default Control Pipe

Da alle Geräte im Sinne der USB-Spezifikation das Vorhandensein der beiden Endpunkte mit Nummer Null garantieren, ist hierüber ein standardisiertes Verfahren möglich, um Geräte zu identifizieren und zu konfigurieren. Der Host beziehungsweise die USB-Systemsoftware hat zum Zeitpunkt des Anschließens des Gerätes noch keinerlei Informationen über das Gerät – lediglich von der Existenz der Endpunkte mit Nummer Null kann ausgegangen werden.

Sofort nachdem besagte Endpunkte kommunikationsbereit sind, erstellt die USB-Systemsoftware eine Message Pipe zur Kommunikation mit den beiden Endpunkten. Diese für alle Geräte zwingend vorhandene Pipe wird "Default Control Pipe" genannt.

Detailinformationen über das Gerät sowie dessen zusätzlich zu den Endpunkten der Nummer Null vorhandener Endpunkte werden vom Host über die Default Control Pipe abgefragt. Neben dem Informationsabruf zur Geräteidentifikation läuft wie bereits angedeutet auch die Konfiguration von Geräten über deren Default Control Pipe. Aufgaben wie die Adressvergabe, deren Behandlung für alle Geräte notwendig sind, werden also ebenfalls über die garantiert vorhandenen Endpunkte der Nummer Null abgewickelt.

Genauere Informationen über die tatsächlich ausgetauschten Daten und die ablaufenden Kommunikationsvorgänge finden sich später in Abschnitt 2.4.4.4 sowie 2.4.5.1 und 2.4.5.2.

Während der gesamten Betriebszeit des Gerätes bleibt die Default Control Pipe aktiv und wird direkt durch die USB-Systemsoftware verwaltet. Auf dem Host laufende Client-Software ist jedoch über den Mechanismus der bereits beschriebenen IRPs ebenfalls zu einem Datenaustausch über die Default Control Pipe in der Lage.

Nachdem in den vorangegangen Abschnitten einige Grundlagen und Anforderungen bezüglich USB erläutert wurden, wird im Folgenden die konkrete Realisierung des Busses betrachtet. Hierzu zählt die mechanische sowie elektrische Verbindung der Geräte, Details der Datenübertragung sowie auch darauf aufbauende Konstrukte und Verfahren zur Gerätekonfiguration und -Identifikation.

2.2 Mechanische Geräteverbindung

2.2.1 Kabel

Verbindungen zwischen USB-Geräten werden über vieraderige Kabel hergestellt. Über zwei der Adern kann die Stromversorgung von Klein- und Kleinstgeräten erfolgen, hierzu wird eine 5V-Gleichspannungsquelle vom Host zur Verfügung gestellt. Die mit dem Pluspol der Spannungsversorgung verbundene Ader bezeichnet die USB-Spezifikation als *VBus*, die Masseader wie üblich mit *GND*. Das zweite Adernpaar dient der eigentlichen Signalübertragung, die Adern wurden hierbei $D+$ beziehungsweise $D-$ benannt.

Für die Anbindung von Full- und Highspeed-Geräten sind zusätzlich eine Schirmung des Kabels sowie verdrillte Adernpaare vorgeschrieben, da hier im Vergleich zu Lowspeed-Geräten eine höhere Anforderung bezüglich der Qualität der Signalübertragung an das Kabel gestellt wird. Für Lowspeed-Geräte sind diese Maßnahmen nicht unbedingt notwendig, werden aber von der Spezifikation empfohlen.

Besitzt das verwendete Kabel eine Schirmung, so sollte diese Massepotential führen.

Abbildung 2.5 zeigt den Schnitt durch ein USB-Kabel.

2.2.2 Steckverbinder

Neben den Kabeln sind auch die zur Verbindung genutzten Steckverbinder durch die USB-Spezifikation genauestens beschrieben. Um Schleifen oder sonstige Fehlverbindungen zu verhindern, wurden zwei verschiedene Arten von Steckern und Buchsen spezifiziert. Das zur Hostseite hin gerichtete Kabelende wird immer mittels Stecker und Buchse des sogenannten Typs “A”

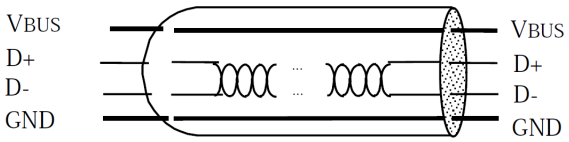


Abbildung 2.5: Schnitt durch ein USB-Kabel (vergleiche [Spek]).

verbunden (Verbindung mit einem Downstreamport). Dem entgegen kommen am gegenüberliegenden Ende (Verbindung zum Upstreamport) Stecker und Buchse des Typs "B" zum Einsatz. Hierbei sind laut Spezifikation die Geräte immer mit Buchsen und die Kabel mit den entsprechenden Steckern zu versehen. Bei Lowspeed-Geräten entfällt die geräteseitige USB-konforme Steckverbindung. Das Kabel ist hier fest (oder zumindest nicht mit einem USB-Steckverbinder) mit dem Endgerät verbunden.

Abbildung 2.6 zeigt die in der USB-Spezifikation vorgesehenen Standard-Steckverbinder.

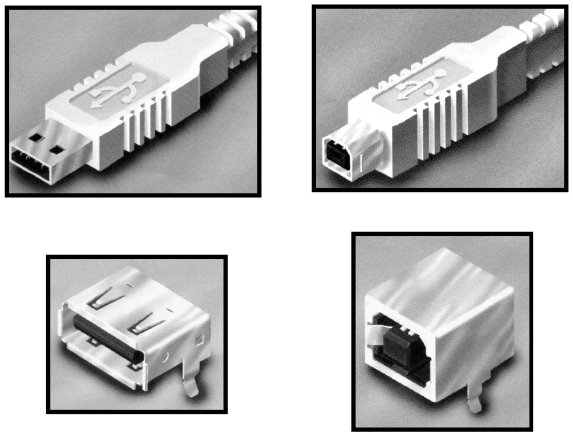


Abbildung 2.6: Steckverbinder der USB-Spezifikation. Links: Typ A, Rechts: Typ B.

Damit bei Verbindung eines Gerätes mit dem Bus dessen Signalleitungen von Anfang an definierte Pegel führen und um die Störsicherheit zu erhöhen, muss dem Gerät eine Spannungsversorgung zur Verfügung gestellt werden, bevor die Signalleitungen Kontakt zum Bus aufnehmen. Hierfür wurden die für VBus und GND zuständigen Verbindungskontakte der Stecker gegenüber den Kontakten für D+ und D− um 1mm verlängert ausgeführt, wodurch

sie beim Anstecken bereits vor den Signalleitungen einen Kontakt der Spannungsversorgung zum Gerät herstellen.

Durch [Spef] ist zusätzlich zu den ursprünglich vorgesehen Konnektoren eine Steckverbindung des Typs "Mini-B" definiert. Diese stellt einen einheitlichen Standard für die Verbindung von Kleingeräten dar, bei deren Entwurf der Einsatz der Standard-B-Verbinder wegen zu geringer Gehäuseabmessungen unpraktikabel war. Ebenfalls erweitert wurde die Palette an verwendbaren Steckern und Buchsen durch das OTG-Supplement, um weitere Steckverbinder für Kleinstgeräte anzubieten.

Genaue Schemazeichnungen und Definitionen für Abmessungen und ähnliches sind der Spezifikation zu entnehmen.

Zwischenzeitlich sind auf dem freien Markt auch eine ganze Reihe Kabel und Verbinder zu finden, welche mechanisch zwar mit USB kompatibel sind, deren Einsatz in Kombination mit USB jedoch zur Fehlfunktion oder gar Zerstörung anderer USB-Hardware führen kann. Die durch USB definierten Steckverbinder werden wegen ihrer kompakten Form und hohen mechanischen Belastbarkeit durch diverse Hersteller gerne zweckentfremdet, auch wenn dies dem eigentlichen Gedanken von USB zuwider ist. Für den Einsatz mit USB vorgesehene Kabel sind mit dem USB-Logo (siehe Abbildung 2.7) zu versehen, welches dem Anwender die einwandfreie Verwendbarkeit mit USB anzeigt.

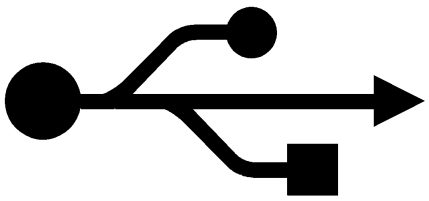


Abbildung 2.7: Logo für USB-Hardware

Jede mit diesem Logo versehene Hardware hat die durch das "USB-Implementers-Forum" definierten Kompatibilitätstests bestanden, wodurch eine störungsfreie Nutzung garantiert ist. Eine Liste der geforderten Kenngrößen und Anforderungen an Kabel und Stecker ist Kapitel 6.6 und 6.7 der Spezifikation ([Spek]) zu entnehmen. Hierbei werden auch solche Anforderungen betrachtet, welche sich auf die elektrischen Eigenschaften des Mediums beziehen oder aus diesen resultieren[6]. Solche Eigenschaften werden im Bedarfsfall in den folgenden Abschnitten näher erläutert.

[6]Insbesondere wird hierdurch die verwendbare Kabellänge eingeschränkt.

2.3 Elektrische Verbindung

Im Rahmen dieser Arbeit werden einige elektrische Eigenschaften des USB betrachtet, deren Kenntnis im Laufe der weiteren Entwicklung notwendig beziehungsweise hilfreich ist. Im Besonderen sind dies die Eigenschaften der Spannungsversorgung, die Signalübertragung und die dabei genutzten Pegel. Weniger interessant sind hier die sich bei jeder Nachrichtenübertragung ergebenden Einschränkungen durch physikalische Effekte. Im Weiteren ist die verwendete Hardware vorgegeben und es ist davon auszugehen, dass die elektrotechnisch kritischen Aspekte der Spezifikation von den Geräteentwicklern beachtet wurden. Einige diesbezüglich wichtige Punkte werden an geeigneter Stelle dennoch behandelt.

2.3.1 Spannungsversorgung

Die über die VBus- und GND-Adern zur Verfügung gestellte Spannungsversorgung ist für den Betrieb kleinerer Geräte ausreichend. Bei einer Spannung von 5V($\pm$0,25V) muss jeder Host sowie jeder Hub mit eigener netzgespeister Stromversorgung dazu in der Lage sein, einen Strom von 500mA über jeden Downstream-Port zu liefern. Downstream-Ports solcher Geräte werden auch Highpower-Ports genannt.

Batterie- beziehungsweise akkubetriebene Root-Hubs können wahlweise auch nur 100mA pro Port zur Verfügung stellen. Für solche Lowpower-Ports reduziert sich ferner die untere Toleranzgrenze der gelieferten Spannung auf 4,4V.

Keinem Gerät ist es erlaubt, Strom für andere Busteilnehmer über seinen Upstreamport bereit zu stellen.

Wird ein Gerät mit dem Bus verbunden, so darf die durch es verursachte Stromaufnahme 100mA nicht überschreiten. Eine Erhöhung der Stromaufnahme des Gerätes vom Bus auf bis zu 500mA kann jedoch während des Konfigurationsvorganges durch die USB-Systemsoftware erlaubt werden. Benötigt ein Gerät die Bereitstellung einer höheren elektrischen Leistung als momentan über den jeweiligen Downstream-Port zur Verfügung gestellt werden kann, so hat die USB-Systemsoftware den weiteren Betrieb des Gerätes zu unterbinden.

2.3.2 Signalübertragung

Bei USB handelt es sich um eine bitserielle Spannungsschnittstelle. Die Datenübertragung auf elektrischer Ebene erfolgt durch Anlegen definierter Pegel an die D+ beziehungsweise D$-$ Leitung.

Jedes USB-Gerät verfügt über eine Sendeeinheit mit zwei Treibern, welche mit den Signalleitungen verbunden sind. Diese müssen die jeweilige Signalleitung zum einen mit einem definierten Lowpegel V_{OL}, zum anderen mit einem definierten Highpegel V_{OH} beschalten können. Die USB-Spezifikation schreibt für V_{OL} das Erreichen einer Spannung zwischen 0V und 0,3V vor, welche der Treiber auch dann gewährleisten muss, wenn die Signalleitung über einen Widerstand von 1,5kΩ gegen ein Potential von 3,6V verbunden wird. Für V_{OH} ist das Erreichen einer Spannung zwischen 2,8V und 3,6V einzuhalten und zwar auch dann, wenn die Leitung mit einem Widerstand von 15kΩ gegen das Massepotential geschaltet wird. Alle Spannungsangaben beziehen sich hierbei auf das beiden Geräten gemeinsame Massepotential, welches auch von der GND-Ader geführt wird. Die Treiber arbeiten demnach als Single-Ended-Transmitter.

Aus elektrotechnischer Sicht gibt die Spezifikation zudem eine weitere Vorgabe für den Entwurf von Lowspeed-Bustreibern. Diese werden unter der Annahme konzipiert, dass Kabel sowie auch angeschlossene Geräte durch eine einfache Kapazität approximiert werden können. Die kleinste spezifikationskonforme Ersatzkapazität für Kabel und Geräte ist hierbei 200pF, die größtmögliche 600pF. Bei Anschluss dieser Kapazitäten muss jeder Lowspeed-Treiber eine Anstiegs- und Abstiegszeit (Rise/Fall-Time) bei Pegelwechseln zwischen 75ns und 300ns gewährleisten.

Für Fullspeed-Geräte ist die zulässige Rise/Fall-Time ebenfalls festgeschrieben, sie beträgt hier zwischen 4ns und 20ns bei Beschaltung mit 50pF.

Neben den Transmittern besitzt jedes Gerät zum Empfang zwei Single-Ended-Receiver, welche jeweils eine der beiden Datenleitungen überwachen und entsprechend das auf dieser Leitung gegenüber GND herrschende Potential feststellen. Gemessene Spannungen unterhalb von 0,8V werden dabei dem Interval V_{IL} zugeordnet und eingangsseitig als Lowpegel interpretiert. Spannung, welche 2,0V überschreiten, werden V_{IH} zugeordnet und gelten dementsprechend als Highpegel.

Zusätzlich zu den Single-Ended-Komponenten ist jedes Gerät mit einem differentiellen Empfängerbaustein ausgestattet, welcher das Differenzensignal zwischen D+ und D− auswertet. Hierbei wird eine Spannungsdifferenz D+ − D− oberhalb von 200mV als "differtielle 1", eine solche unterhalb von −200mV als "differentielle 0" erkannt. Im Allgemeinen kann davon ausgegangen werden, dass eventuell von außen auf die Übertragungsstrecke einwirkende Störungen sich auf beide zur Differenzenbildung genutze Adern gleichermaßen auswirken. Somit sorgt die Verwendung eines Differenzensignals im Vergleich zu Single-Ended-Komponenten für eine höhere Robustheit der Signalübertragung, da das Differenzensignal bei gestörter und ungestörter

Übertragung gleich sein sollte.

Für die Datenübertragung ergeben sich auf elektrischer Seite für USB drei aktiv genutzte Zustände. Neben der "differentiellen 0" und "differentiellen 1" wird der sogenannte "SE0"-Zustand genutzt, bei welchem beide Signalleitungen Lowpegel führen[7].

2.3.3 Einsteck- und Geschwindigkeitserkennung

Wird ein Gerät mit dem Bus verbunden, so muss sowohl das Anstecken an sich, als auch die durch das Gerät beherrschte Übertragungsrate (Low- oder Fullspeed) zur Laufzeit erkannt werden.

Hostseitig sind dazu die beiden Datenleitungen jeweils mit einem Pull-Down-Widerstand zwischen 14,25kΩ und 24,8kΩ gegen Masse verbunden, was in Abwesenheit eines Gerätes für einen definierten Lowpegel auf beiden Leitungen sorgt. Es herrscht dann also der SE0-Zustand.

Dementgegen wurde bei Lowspeed-Geräten D−, bei Fullspeed-Geräten jedoch D+ mit einer PullUp-Beschaltung versehen, mit welcher beim Einstecken des Gerätes die jeweilige Leitung auf Highpegel gebracht wird. Hierzu kann ein Widerstand von 1,5kΩ genutzt werden, genauere Angaben sind der Änderungsnotiz [Spei] zu entnehmen. Der Host kann so zweifelsfrei die vom Gerät beherrschte Geschwindigkeit erkennen.

Gleichzeitig dienen die verbauten Widerstände auch der Terminierung der Leitungen, wodurch Signalreflektionen am Kabelende gedämpft werden. Genauere Angaben zu Eingangs- und Leitungsimpedanzen sind ebenfalls der Spezifikation zu entnehmen.

Durch die (passiven) PullUp-Widerstände ergeben sich auf den Datenleitungen Pegel, welche als Ruhezustand des Busses interpretiert werden. Der Ruhezustand bei Betrieb eines Lowspeed-Gerätes ist also invers zu dem eines Fullspeed-Gerätes.

Wird ein Gerät angeschlossen, so wechselt der Bus von SE0 in den Ruhezustand. Nach 2,5μs stabiler Zustandsdauer erkennt der Host das Einstecken des Gerätes. Umgekehrt ist ein Abziehen eines Gerätes festzustellen, wenn der Buszustand für einen Zeitraum von mindestens 2,5μs den SE0-Zustand einnimmt, ohne dass der Host diesen aktiv erzeugt.

Abbildung 2.8 zeigt beispielhaft den Pegelverlauf bei Anschluss eines Lowspeed-Gerätes.

[7] Der durch die USB-2.0-Spezifikation definierte SE1-Zustand (D+ und D− führen Highpegel) stellt im Rahmen einer Full- beziehungsweise Lowspeed-Übertragung zu keinem Zeitpunkt ein valides Signal dar.

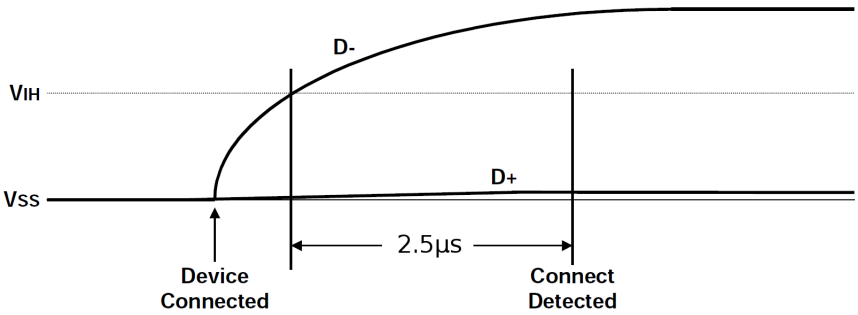


Abbildung 2.8: Signalverlauf bei Anschluss eines Lowspeed-Gerätes (vergleiche [Spek]).

2.3.3.1 Abstraktion vom physikalischen Differenzensignal

Da die Ruhepegel von Low- und Fullspeed-Geräten zueinander invers sind – empfängerseitig wird hier bei Lowspeed-Geräten eine "differentielle 0", bei Fullspeed-Geräten jedoch eine "differentielle 1" erkannt – wird im Folgenden eine geeignete Abstraktion für die Darstellung des Differenzensignals benötigt.

Die USB-Spezifikation sieht hierfür die Zustände J sowie K vor, wobei J für die "differentielle 0" bei Lowspeed- und die "differentielle 1" bei Fullspeed-Geräten steht. K bezeichnet den dazu inversen Fall[8].

Im weiteren Verlauf werden Low- und Fullspeed-Geräte im Text soweit es die Signale angeht gleich behandelt, indem jeweils nur die Darstellung der differentiellen Signale mittels J und K genutzt wird.

2.3.4 Gerätereset

USB-Geräte (beziehungsweise deren USB-Einheit) lassen sich über ein Signal auf dem Bus zurücksetzen. Tatsächlich muss jedes Gerät nach Verbindung mit dem Bus durch den Host zunächst zurückgesetzt werden, bevor es seine eigentliche Arbeit aufnehmen kann[9].

Erzeugt wird ein USB-Reset durch einen 10ms andauernden SE0-Zustand des Busses. Da es sich elektrisch gesehen bei USB nur um Punkt-zu-Punkt-Verbindungen handelt, kann so jedes Gerät einzeln durch den Host zurückgesetzt werden. Der Host weist dazu den Hub, über welchen das fragliche Gerät mit dem Bus verbunden ist, zur Durchführung des Resets an.

[8] Hiermit liegt der Zustand J senderseitig sehr nahe am Ruhezustand des Busses.
[9] Siehe hierzu auch 2.4.5.3.

Geräteseitig darf ein USB-Reset nach 2,5μs dauerndem SE0-Zustand angenommen werden. Die USB-Einheit des Gerätes muss das Rücksetzen noch während des Andauerns des SE0 garantieren, so dass es sofort nach der 10ms währenden Reset-Zeit des Hosts empfangsbereit ist.

2.3.5 Suspend und Resume

Ein USB-Gerät, welches für eine Dauer von mindestens 3ms einen ununterbrochenen Ruhezustand festellt, muss sich in einen "Suspended" genannten Zustand begeben, in welchem es eine geringere Leistungsaufnahme benötigt. Die Spezifikation schreibt Geräten hierbei vor, nicht länger als 7ms für den Übergang in diesen Zustand zu benötigen.

Geräte erwachen aus einem herrschenden Suspended-Zustand, wenn auf ihrem Upstream-Port kein Ruhezustand mehr herrscht. Diese Wiederaufnahme der normalen Funktion wird auch als "Resume" bezeichnet. Zusätzlich erlaubt die USB-Spezifikation auch das Aufwachen von Geräten ohne Aktivität auf dem Bus. Angemerkt sei an dieser Stelle, dass aufgrund der Eigenschaften des USB der Übergang von Geräten in den Suspend-Modus explizit vom Host zugelassen werden muss. Weitere Details zu Suspend und Resume finden sich in den Abschnitten 7.1.7.6 sowie 7.1.7.7 der Spezifikation [Spek].

2.3.6 Codierungsverfahren

Die Übertragung logischer Werte erfolgt bei USB über die Zustände J und K nach dem NRZI-Verfahren (Non Return to Zero Invert).

Hierbei entspricht jeder Zustandswechsel einer logischen Null, ein Gleichbleiben des Zustands repräsentiert eine logische Eins. Im Falle langer Ketten von Nullen in den zu übertragenden Daten erzeugt dieses Verfahren optimal viele Pegelwechsel und macht so die Erkennung von Bitgrenzen für den Empfänger sehr einfach. Problembehaftet ist jedoch eine Übertragung langer Eins-Ketten, da hier bei reinem NRZI keinerlei Pegelwechsel auftreten. Um die Synchronisation zwischen Sender und Empfänger zu gewährleisten, wird daher zusätzlich Bitstuffing angewandt.

Nach jeweils sechs aufeinanderfolgend zu sendenden Eins-Werten wird senderseitig automatisch eine logische Null in den Datenstrom eingefügt, wodurch mindestens alle sieben Takte ein Zustandswechsel zwischen J und K erzwungen wird. Der Empfänger entfernt solche Nullen automatisch wieder aus dem Datenstrom. Hierbei ist es wichtig, dass eine Null nach sechs Einsen auch dann eingefügt wird, wenn der nächste zu sendende Bitwert eine Null ist und damit schon von sich aus einen Zustandswechsel erzwingen würde.

Bitstuffing ist ein mit einem einfachen Automaten realisierbares Verfahren zur Erleichterung der Synchronhaltung von Sender und Empfänger. Die benötigte Technik ist daher relativ simpel und kostengünstig. Durch die Kombination aus NRZI und Bitstuffing kann insbesondere bei zuverlässiger Bitgrenzenerkennung auf eine zusätzliche Taktleitung verzichtet werden.

Wie später noch zu sehen sein wird, ist die Wahl von NRZI als Codierungsverfahren sinnvoll, da längere Ketten von Nullen durch das bei USB verwendete Protokoll vermehrt auftreten.

2.3.7 Übertragungsrate, Jitter und Delay

Wie bereits erwähnt beherrscht der für diese Arbeit zu verwendende Controller sowohl den Lowspeed- als auch den Fullspeed-Modus, weshalb die Kenngrößen für beide Betriebsarten hier betrachtet werden müssen.

Die Übertragungsrate für den Lowspeed-Modus ist durch die Spezifikation auf 1,5Mbit/s festgeschrieben, was in Verbindung mit dem bereits beschriebenen NRZI-Verfahren eine maximale Signalfrequenz von 750kHz ergibt. Die zeitliche Länge eines Bits entspricht $666,\overline{6}$ns.

Bei Fullspeed-Geräten beträgt die Übertragungsrate 12Mbit/s, was einer maximalen Signalfrequenz von 6MHz beziehungsweise einer Bitdauer von $83,\overline{3}$ns entspricht.

Zusammen mit einigen weiteren Überlegungen lassen sich Aussagen zu den maximal tolerierbaren Signalleitzeiten von Kabeln und damit dann auch zu deren maximaler Länge treffen. Die Spezifikation schreibt als Berechnungsgrundlage für alle verwendeteten Kabel ein Propagation-Delay von nicht mehr als $5,2\frac{ns}{m}$ vor.

Wie erwähnt werden Lowspeed-Bustreiber unter der Annahme konzipiert, dass Geräte und Kabel mittels einer einfachen Kapazität approximiert werden können. Diese Abschätzung ist jedoch nur dann zuverlässig, wenn die Signalreflexion noch während der ersten Hälfte der Flankenwechselzeit von minimal 70ns (F_{min}) erfolgt. Um dies zu erreichen, darf eine bestimmte Kabellänge K_{max} nicht überschritten werden. Es gilt:

$$\frac{F_{min}}{2} = \frac{2 * K_{max}}{5,2\frac{ns}{m}} \Rightarrow K_{max} \approx 3,37m$$

Tatsächlich schreibt die Spezifikation für Lowspeed-Kabel ein Propagation-Delay von maximal 18ns vor, das für die Weiterentwicklung und Verbreitung von USB zuständige USB-Implementers-Forum (USB-IF) begrenzt jedoch die Länge nachträglich auf 3m ([FAQ]).

Für Fullspeed-Übertragungen ergeben sich keine Restriktionen aus der Flankensteilheit der Pegelwechsel, jedoch ist aus übertragungstechnischer

Sicht auch hier darauf zu achten, dass auftretende Signalreflektionen noch während der Sendedauer des reflektierten Bits ausreichender Dämpfung unterliegen. Die USB-Spezifikation schreibt daher eine Signallaufzeit von nicht mehr als 26ns für jedes im Fullspeed-Modus verwendete Kabel vor. Dies entspricht genau einer maximalen Kabellänge von 5m.

Hier liegt auch der Grund dafür, dass spezifikationskonforme Lowspeed-Geräte niemals eine USB-konforme Buchse besitzen dürfen, sondern in aller Regel mit einem fest verbundenen Kabel ausgeliefert werden. Hierdurch wird verhindert, dass ein unbedarfter Nutzer ein zu langes Kabel nutzt, um das Gerät mit dem Host beziehungsweise einem Hub zu verbinden. Insbesondere verbietet die Spezifikation auch die auf dem freien Markt jedoch sehr beliebten "Verlängerungen" (A-Stecker auf A-Buchse), da der Anwender diese ebenfalls zu einer spezifikationsverletzenden Längenerhöhung des Kabelweges nutzen könnte.

Zusätzlich zu der durch den Kabelweg verursachten Signalverzögerung von maximal 26ns erlaubt die USB-Spezifikation weitere Verzögerungszeiten für die Signalleitung zu den Anschlussbuchsen innerhalb der Geräte selbst. Für Downstream-Ports sind hierbei 3ns, für Upstream-Ports 1ns vorgesehen. Somit ergibt sich ein zulässiges Propagation-Delay von 30ns von einer USB-Einheit zur nächsten. Bei Verwendung eines Hubs ist für diesen eine maximale Signalleitzeit von 70ns inklusive des upstreamseitig verwendeten Anschlusskabels vorgesehen.

Die Signalübertragung unterliegt ferner einem gewissen Grad an Jitter, also zeitlichen Variationen in der Signallaufzeit. Die USB-Spezifikation gibt die einzuhaltenden Toleranzen sowohl für die Sender- als auch für die Empfängerseite vor[10]. In der Praxis wird dem Problem des Jitter auf Empfängerseite durch ein vierfaches Oversampling bei der Signalabtastung Einhalt geboten. Entsprechend des Nyquist-Shannon-Abtasttheorems könnte ein Fullspeed-Signal bereits mit 12MHz Abtastfrequenz erfasst werden – Ein vierfaches Oversampling wird also durch Abtasten mit einer Frequenz von 48MHz erreicht.

Zu beachten ist dabei, dass Jitter sich aufgrund seiner Eigenschaft als Variation in der Signallaufzeit im zeitlichen Mittel selbst ausgleicht. Gemessene Werte für die Abweichung der Bitdauer vom Sollwert müssen sich also bei ausreichend großem zeitlichen Betrachtungsinterval zu Null aufsummieren. Sollte dies nicht der Fall sein, so handelt es sich bei der Störgröße nicht um Jitter, sondern um einen dauerhaften Fehler im Sendetakt.

[10] Siehe hierzu [Spek] Kapitel 7.1.13.1 sowie 7.1.15.1.

2.4 Protokoll-Schicht

Die Übertragung von Daten erfolgt natürlich auch bei USB nach den Regeln eines festgeschriebenen Protokolls, an welches sich alle Busteilnehmer halten müssen. Hierbei ist – wie bei vielen anderen Systemen auch – eine paketbasierte Datenübermittlung vorgesehen. Die in den vorangegangenen Abschnitten beschriebenen (elektrischen) Zustände werden so zu logischen informationstragenden Einheiten zusammengruppiert. Mit den einzelnen Paketen lassen sich dann wiederum einzelne Transaktionen bewerkstelligen, durch deren unterschiedliche Abfolge und Gruppierung letztlich die verschiedenen von USB gebotenen Transferarten realisiert sind.

Dieses Kapitel beschreibt Einzelheiten zu Paketaufbau und Transferabwicklung, sofern diese für den Full- und Lowspeed-Modus relevant sind.

2.4.1 Bit- und Byteorder

Die Datenübertragung erfolgt bei USB von LSb nach MSb, so dass das niederwertigste Bit der zu übertragenden Daten zuerst auf den Bus gelegt wird. Sind im Folgenden Pakete im Text repräsentiert, so entspricht die Leserichtung von links nach rechts der Reihenfolge, in welcher die Bits auf den Bus gebracht wurden. Dies ist zu beachten, da diese Darstellung gerade nicht der üblichen Zahlendarstellung, bei welcher die höchstwertige Stelle links angegeben wird, entspricht.

Im Falle von mehrbytigen Feldern, wie sie im weiteren Verlauf noch auftauchen werden, erfolgt die Übertragung der einzelnen Bytes nach der Little-Endian-Byteorder, es wird also das niedertwertigste Byte zuerst übertragen.

2.4.2 Bauteile von Paketen

2.4.2.1 Paketanfang, SYNC-Feld und Paketende

Die Übertragung eines Paketes beginnt mit dem Senden des sogenannten SYNC-Feldes. Die dabei übertragene Bitfolge 0x80 ergibt eine Zustandsfolge von $KJKJKJKK$ auf dem Bus. Somit entsteht eine maximale Anzahl von Pegelwechseln, mit deren Hilfe sich der Empfänger auf das empfangene Signal einstellt. Ein Paket beginnt demnach immer mit einem Wechsel des Busses von Ruhezustand nach K. Dieser Zustandsübergang wird im Folgenden mit Start-of-Paket (SOP) bezeichnet.

Durch das doppelt auftretende K-Signal am Ende des SYNC-Feldes erkennt der Empfänger auch dann noch sicher den Anfang des nächsten Paketfeldes, wenn nicht die gesamte Synchronisationsfolge empfangen wurde.

Nach einer Reihe weiterer Felder wird jedes Paket mit einem definierten Signal beendet, welches als End-of-Packet (EOP) bezeichnet wird. Hierbei handelt es sich um ein SE0, dessen Dauer mindestens eine Bitzeit beträgt und welchem sich ein *J*-Zustand anschließt. Danach kann der Bus wieder den Ruhezustand einnehmen.

Neben SOP, SYNC und EOP sind Pakete noch aus weiteren Feldern zusammengesetzt, deren Aufgabe und Format in den folgenden Abschnitten beschrieben ist.

2.4.2.2 Packet Identifier (PID)

Jedes Paket besitzt ein PID-Feld, über welches der Pakettyp und damit der weitere Aufbau des Paketes eindeutig identifiziert ist. Pakettypen werden dabei durch eine Sequenz von vier Bits repräsentiert, wobei das PID-Feld zusätzlich zu diesen vier Bit auch noch deren Einerkomplement als Prüfsumme enthält. Abbildung 2.9 zeigt das Format des PID-Feldes.

PID_0	PID_1	PID_2	PID_3	$\overline{PID_0}$	$\overline{PID_1}$	$\overline{PID_2}$	$\overline{PID_3}$

Abbildung 2.9: Format des PID-Feldes

Jeder Busteilnehmer muss das PID-Feld decodieren und interpretieren können. Sollte dabei – beispielsweise durch eine fehlerhafte Übertragung – eine unbekannte oder fehlerhafte PID erkannt werden, so muss der Empfänger das Paket ignorieren. Gleiches gilt auch, falls die PID zwar valide, der durch sie angezeigte Pakettyp für den Empfänger jedoch nicht verarbeitbar ist.

Tabelle 2.1 gibt eine vollständige Übersicht der für diese Arbeit relevanten PIDs beziehungsweise Pakettypen.

2.4.2.3 Geräte-Adressfeld (ADDR)

Diverse Pakettypen enthalten ein Adressfeld, welches den Empfänger des Paketes identifiziert. Das ADDR-Feld ist eine Folge aus sieben Bit, welche genau der Geräteadresse (0–127) entspricht.

Abbildung 2.10 zeigt das Format des ADDR-Feldes.

PID Typ	PID Name	Code[3 : 0]	Bitfolge	Beschreibung
Token	OUT IN SOF SETUP	0001B 1001B 0101B 1101B	10000111B 10010110B 10100101B 10110100B	Ausgehende Kommunikation Eingehende Kommunikation Start-Of-Frame Paket Setup-Verbindungen
DATA	DATA0 DATA1	0011B 1011B	11000011B 11010010B	Gerades Datenpaket Ungerades Datenpaket
Handshake	ACK NACK STALL	0010B 1010B 1110B	01001011B 01011010B 01111000B	Erfolgreich Nicht erfolgreich Anhalten der Verbindung
Special	PRE	1100B	00111100B	Lowspeed-Präambel

Tabelle 2.1: Paketarten und zugeordnete PIDs beziehungsweise deren Bitfolgen.

$ADDR_0$	$ADDR_1$	$ADDR_2$	$ADDR_3$	$ADDR_4$	$ADDR_5$	$ADDR_6$

Abbildung 2.10: Format des ADDR-Feldes

2.4.2.4 Endpunktnummer (ENDP)

Zusätzlich zur Geräteadresse wird der Paketempfänger über die Endpunktnummer genauer identifiziert[11]. Hierzu steht das ENDP-Feld zur Verfügung, welches die vierbittige Endpunktnummer aufnimmt. Abbildung 2.11 zeigt das Format des ENDP-Feldes.

$ENDP_0$	$ENDP_1$	$ENDP_2$	$ENDP_3$

Abbildung 2.11: Format des ENDP-Feldes

[11] Vergleiche Abschnitt 2.1.4.

2.4.2.5 Feld der aktuellen Framenummer (FrameNumber)

Dieses Feld enthält die elfbittige Nummer des aktuellen sogenannten Frames. Frames sowie deren Anwendung werden später in Abschnitt 2.4.3.4 erläutert.

2.4.2.6 Data-Feld

Das Datenfeld nimmt beliebige zu übertragende Daten auf, wobei die Anzahl der zu sendenden Bytes immer ganzzahlig sein muss. Für Lowspeed-Übertragungen gilt ferner eine maximale Länge des Datenfeldes von acht Byte, bei Fullspeed-Übertragungen sind bis zu 1023 Byte in diesem Feld möglich. Die zu sendenden Daten werden byteweise LSb-first in das Feld und somit auf den Bus gebracht.

2.4.2.7 CRC-Checksummenfeld

Einige Pakete beziehungsweise einige deren Felder sind mit einer CRC-Checksumme gegen Übertragungsfehler gesichert. Der verwendete CRC-Typ wird dabei so gewählt, dass durch die Checksumme alle ein- und zweibittigen Übertragungsfehler erkannt werden können. Je nach Paket findet daher eine CRC5- oder eine CRC16-Prüfsumme Anwendung.

Die für CRC5 und CRC16 genutzten Generatorpolynome lauten:

$$G_5(X) = X^5 + X^2 + 1$$

$$G_{16}(X) = X^{16} + X^{15} + X^2 + 1$$

Das PID-Feld eines Paketes ist nie Teil der CRC-Checksummenbildung, da der PID-Wert durch das mitgelieferte Einerkomplement schon selbst ausreichend gesichert ist.

Bei der Implementation ist darauf zu achten, dass aufgrund der verwendeten Variante des CRC-Verfahrens der empfängerseitig verbleibende Rest bei erfolgreicher Übertragung ungleich Null ist. Im Fall von CRC5 gibt ein Rest von 01100B eine erfolgreiche Übertragung an, bei CRC16 muss der Restwert 1000000000001101B entsprechen. Zu beachten ist ebenfalls, dass das Übertragen der CRC-Checksumme entgegen dem Vorgehen bei anderen Feldern MSb-first erfolgt, so dass das höchstwertige Bit jeweils zuerst auf den Bus gebracht wird. Sollte der Empfänger eine unpassende Checksumme vorfinden, so muss er alle mit dieser gesicherten Felder und damit üblicherweise das komplette Paket ignorieren.

2.4.3 Pakete auf dem Bus

Die bis hierhin beschriebenen Paketfelder werden zu kompletten Paketen zusammengesetzt, welche in den folgenden Abschnitten vorgestellt werden. Eine Instanz eines jeden Pakettyps wurde hierzu mit Hilfe eines Digitaloszilloskops aufgenommen und dient zur Veranschaulichung des Aufbaus des gezeigten Typs.

2.4.3.1 Paketgattungen

Die verschiedenen Pakete sind jeweils einer der Gattungen "Token", "Data" oder "Handshake" zugeordnet. Wie schon in Tabelle 2.1 zu sehen ist, lässt sich die Gattung eines Paketes durch die ersten beiden Bits des jeweiligen PID-Feldes erkennen. Ein PRE-PID dient dabei, wie noch zu sehen sein wird, nie als eigenständige PID eines Paketes. Tokenpakete werden ausschließlich vom Host erzeugt und dienen allein der Steuerung des Busses beziehungsweise der angeschlossenen Geräte. Beispielsweise wird die Zuteilung von Buszeit an ein bestimmtes Gerät mit Hilfe von Tokenpaketen realisiert.

Datenpakete enthalten die eigentlichen zu übertragenden Nutzdaten und können sowohl durch Host als auch durch Geräte erzeugt werden.

Handshakepakete dienen der Quittierung der empfangen Daten sowie der Flusskontrolle. Auch diese Pakete können grundsätzlich von Host- und auch von Geräteseite auf den Bus gebracht werden.

2.4.3.2 Transaktionen und Transfers

Letztlich dienen Pakete dem Zweck, Nutzdaten zwischen Kommunikationspartnern auszutauschen. Datenaustausch ist bei USB nicht über den Versand eines einzelnen Paketes möglich, sondern ergibt sich aus dem Zusammenspiel mehrerer Pakete der verschiedenen oben genannten Gattungen.

Eine Paketfolge, welche notwendig ist, um Nutzdaten von Gerät zu Host oder von Host zu Gerät zu transportieren, wird als Transaktion bezeichnet. Eine solche Paketfolge besteht aus einem Tokenpaket zur Buszeitzuteilung, einem Datenpaket mit den eigentlichen Nutzdaten und einem darauffolgenden Handshakepaket als Empfangsquittung. Transaktionen sind daher – entsprechend der Gattung der beteiligten Pakete – in drei sogenannte Phasen aufgeteilt: Die Token-, Data- sowie die Handshake-Phase. Je nach gewählter Transferart kann die Handshake-Phase auch entfallen.

Eine einzelne Transaktion kann entweder im Gesamten erfolgreich sein oder aber fehlschlagen. Kommt es also zum Verlust oder zur Beschädigung eines einzelnen an der Transaktion beteiligten Paketes, so ist die komplette Transaktion zu verwerfen und gegebenenfalls zu wiederholen.

Wie noch zu sehen sein wird, ist die Anzahl der Bytes, welche ein einzelnes Datenpaket als Nutzdaten (Payload) aufnehmen kann, limitiert. Überschreitet die zu übertragende Datenmenge die Payloadgröße eines Paketes, so ist es notwendig, die Daten auf mehrere Datenpakete und damit auf mehrere Transaktionen aufzuteilen.

Ein kompletter Transfer – also die Übertragung einer beliebigen Datenmenge – kann daher (eingeschränkt nur durch die Transferart) aus mehreren Transaktionen bestehen. Der Empfänger erhält die gesamten übertragenen Daten dann durch Zusammensetzen der Nutzdaten der einzelnen übertragenen Datenpakete.

2.4.3.3 Framing

Um den am Bus teilnehmenden Geräten eine einheitliche Zeitbasis zur Verfügung zu stellen, nutzt USB das Konzept der sogenannten Frames. Bei Full- und Lowspeed-Übertragungen handelt es sich bei einem Frame um einen Zeitraum von 1ms. Mit dem Ablauf eines Frameintervals startet jeweils sofort das darauf folgende Frame, wobei jedem Frame eine elfbittige Zahl zueigen ist, welche von Frame zu Frame inkrementiert wird.

Durch das Framing ist die Buszeit in kleine, gut verwaltbare Intervalle eingeteilt, auf welche sich später die verschiedenen Transferarten beziehen können.

Der Host kann während der Dauer eines Frames beliebige Transaktionen vornehmen. Hierbei achtet er bei der Transaktionsplanung jedoch darauf, dass keine Transaktion über eine Framegrenze hinaus andauert. Pakete einer solchen Transaktion würden mit der nachfolgend beschriebenen Signalisierung der Framegrenzen für die Geräte kollidieren.

Sollte – beispielsweise durch eine Gerätefehlfunktion – dennoch Aktivität auf dem Bus in enger zeitlicher Nähe zu einem Frameende auftreten, so ist es Aufgabe der Hubs, den fehlerhaften Ast der Topologie vom Bus zu trennen, so dass sich der restliche Bus zu Beginn des nächsten Frames im Ruhezustand befindet.

Zur Klarstellung sei gesagt, dass einzelne Transfers sich durchaus über die Framegrenzen hinaus erstrecken können. Mehrere Transaktionen, welche innerhalb verschiedener Frames stattfinden, können also ein und dem selben Transfer angehören. Zusätzlich können die Transaktionen beliebiger Transfers auch in beliebiger Folge verplant werden. Es ist also nicht notwendig einen bestimmten Transfer abzuschließen, bevor ein weiterer Transfer starten kann[12].

[12] Für einen bestimmten Endpunkt ist dabei natürlich zu jedem Zeitpunkt immer nur ein Transfer aktiv.

Abbildung 2.12 gibt ein Beispiel eines möglichen Ablaufs von Transaktionen innerhalb der Frames, wobei auch die nachfolgend beschriebenen Start-Of-Frame Pakete berücksichtigt wurden.

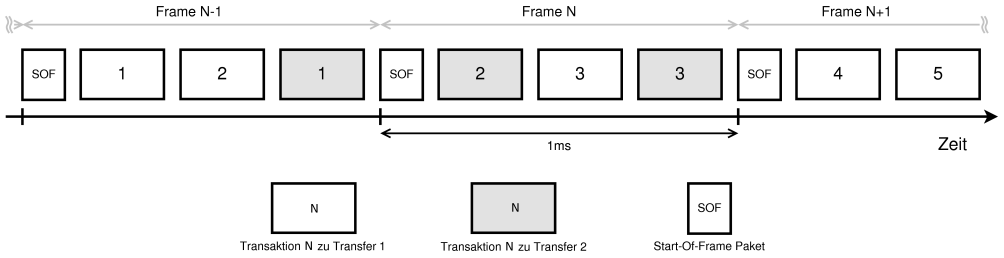


Abbildung 2.12: Möglicher Transaktionsfolge zweier Transfers.

2.4.3.4 Start Of Frame (SOF) und Keep-Alive

Um den Busteilnehmern die Framegrenzen mitzuteilen, bringt der Host bei jedem Neubeginn eines Frames (also jeweils nach Ablauf einer Millisekunde) für Fullspeed-Geräte das sogenannte Start-Of-Frame-Paket (SOF) auf den Bus.

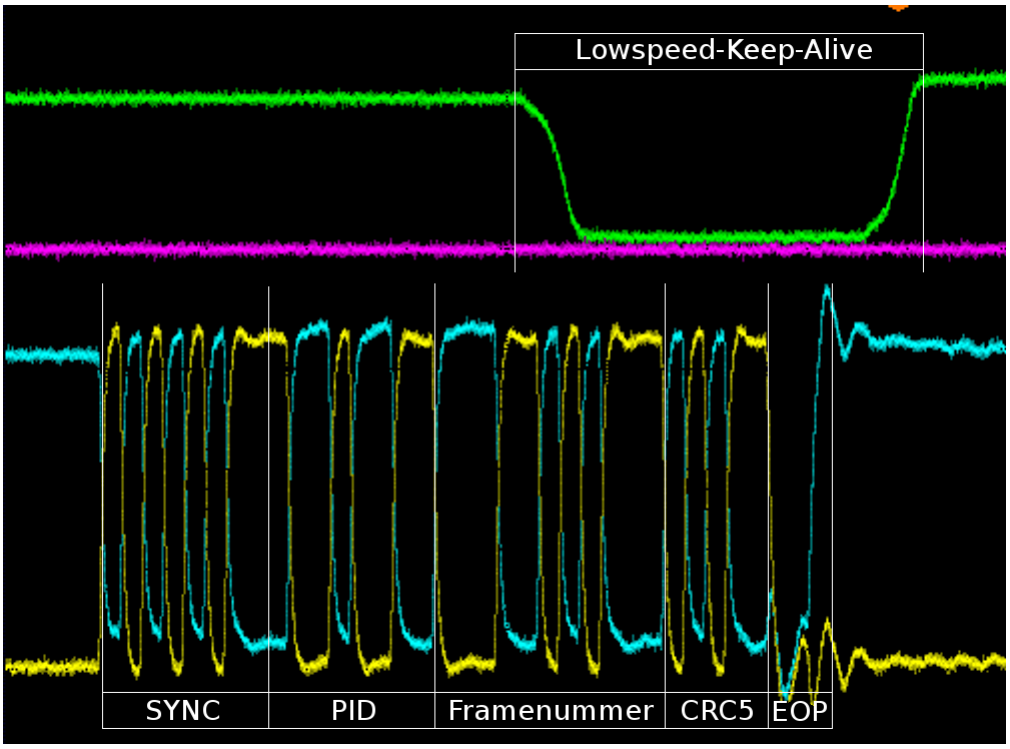
Ein SOF-Paket besteht aus einem entsprechenden PID-Feld, dem elfbittigen Framenummern-Feld und einer abschließenden CRC5-Checksumme, welche die Framenummer sichert.

Für Lowspeed-Geräte wird aus Zeitgründen kein SOF-Paket, sondern ein einfaches EOP – in diesem Fall dann Keep-Alive genannt – verwendet. Gegebenenfalls setzt ein Hub auf seinem Upstream-Port eintreffende SOF-Pakete in Keep-Alives für die downstreamseitig an ihn angeschlossenen Lowspeed-Geräte um; dieser Fall ist auch in Abbildung 2.13 zu sehen.

Wie an der PID zu erkennen ist, gehört das SOF-Paket zur Gattung der Tokenpakete. Für die Framenummer ergibt sich im abgebildeten Fall 1558, aufgrund der maximalen Länge der Framenummer von elf Bit ist der maximale Zählwert auf 2047 begrenzt. Erreicht der Zähler diesen Wert, so kommt es zu einem einfachen Überlauf und die Zählung beginnt erneut bei Null.

2.4.3.5 SETUP-Paket

Ebenfalls zur Gattung der Tokenpakete gehört das SETUP-Paket. Dieses besteht aus der PID, einem ADDR- und einem ENDP-Feld, welche den Empfän-



$KJKJKJKK$	$KJJKJJKK$	$JJJKKJKJKKK$	$JKJKK$
00000001	10100101	01101000011	00001
$SYNC$	PID	$FrameNumber$	$CRC5$

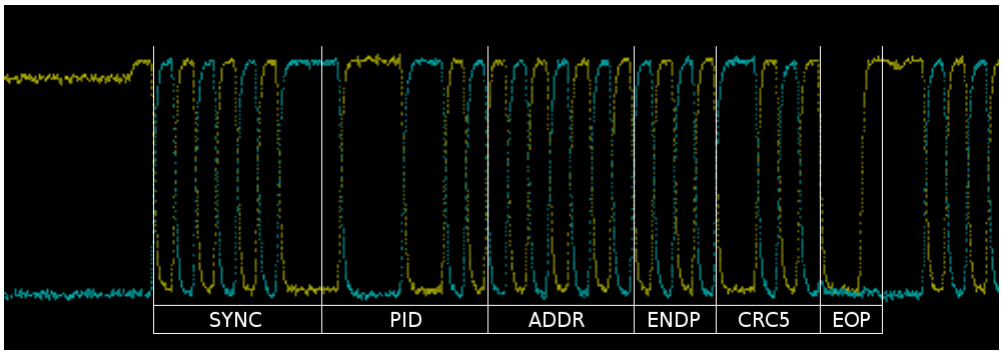
Abbildung 2.13: Fullspeed SOF (unten) mit entsprechender Keep-Alive-Umsetzung (oben), gemessen auf Upstream- und auf Downstreamseite eines Hubs im Fullspeed-Modus.

ger definieren sowie einer abschließenden CRC5-Summe, die sich über ADDR und ENDP erstreckt.

Ein Beispiel für ein SETUP-Paket ist in Abbildung 2.14 zu sehen.

Vorgreifend sei gesagt, dass SETUP-Pakete immer den Beginn eines neuen Control Transfers anzeigen.

Die Bearbeitung von SETUP-Paketen beziehungsweise der durch diese eingeleiteten Transaktionen, hat für jedes Gerät die höchstmögliche Priorität. Etwaige laufende Aktionen sind für die Reaktion auf ein SETUP zu unterbrechen, so dass jedes am Bus teilnehmende Gerät zu jeder Zeit dazu in der Lage ist, ein solches Paket zu behandeln. SETUP-Pakete werden jedoch nur von Endpunkten, welche für Control Transfers zuständig sind, behandelt. SETUP-Pakete, welche bei Endpunkten eintreffen, deren Typ nicht



$KJKJKJKK$	$KJJJKKJK$	$JKJKJKJ$	$KJKJ$	$JKJKJ$
00000001	10110100	0000000	0000	10000
$SYNC$	PID	$ADDR$	$ENDP$	$CRC5$

Abbildung 2.14: SETUP Token Paket

"Control" entspricht, müssen vom Gerät immer ignoriert werden.

2.4.3.6 IN-Paket

Ein weiteres Tokenpaket ist das IN-Paket, welches der Host dazu nutzt, um Geräten Buszeit zuzuteilen, so dass diese Daten an den Host senden können.

IN-Pakete folgen dem selben Aufbau wie SETUP-Pakete. Die eindeutige Adressierung durch Geräteadresse und Endpunktnummer sorgt dafür, dass tatsächlich immer nur genau ein Gerät (beziehungsweise dessen angesprochener Endpunkt) in Reaktion auf das IN den Bus belegen wird.

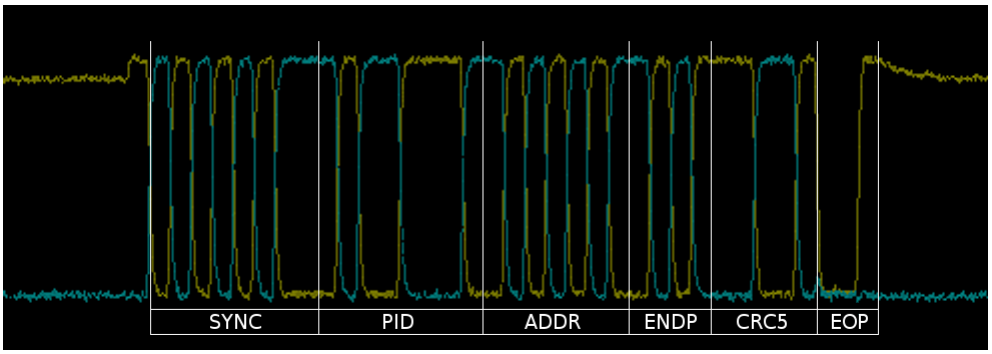
Abbildung 2.15 zeigt eine Instanz eines IN-Paketes, wobei hier Endpunkt 0 bei Geräteadresse 1 adressiert ist.

2.4.3.7 OUT-Paket

Das letzte noch ausstehende Tokenpaket ist das OUT-Paket, durch welches der Host einem Gerät eine nachfolgende Datensendung anzeigt.

Der Paketaufbau folgt wiederum dem der SETUP- und IN-Pakete.

Abbildung 2.16 zeigt ein Beispiel eines OUT-Paketes, diesmal ist Endpunkt 2 des Gerätes 2 adressiert.



KJKJKJKK	*KJKKJJJK*	*KJKJKJK*	*KJKJ*	*JJKKJ*
00000001	10010110	1000000	0000	11010
SYNC	*PID*	*ADDR*	*ENDP*	*CRC*5

Abbildung 2.15: IN Token Paket

2.4.3.8 DATA0- und DATA1-Paket

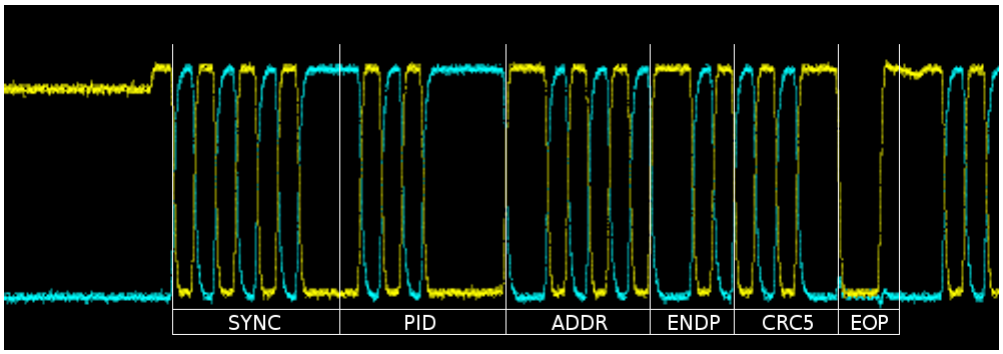
DATA0- und DATA1-Pakete gehören der Gattung der Datenpakete an. Solche Pakete bestehen aus ihrer PID und einem darauf folgenden DATA-Feld zur Aufnahme der zu übertragenden Nutzdaten. Hinter dem DATA-Feld befindet sich zusätzlich eine CRC16-Checksumme, welche die Nutzdaten sichert.

Abbildung 2.17 zeigt exemplarisch den Aufbau eines DATA0-Paketes, wobei aus Gründen der Paketlänge die abschließende CRC16-Summe nicht vollständig abgedruckt wurde.

Zur Klarstellung sei gesagt, dass Aufbau und Eigenschaften der DATA0- und DATA1-Pakete – abgesehen von einem einzigen Bit innerhalb ihrer PID – tatsächlich vollkommen identisch sind. Die Notwendigkeit verschiedener Typen von Datenpaketen ergibt sich erst aus dem später betrachteten Handshake-Verfahren des USB-Protokolls[13].

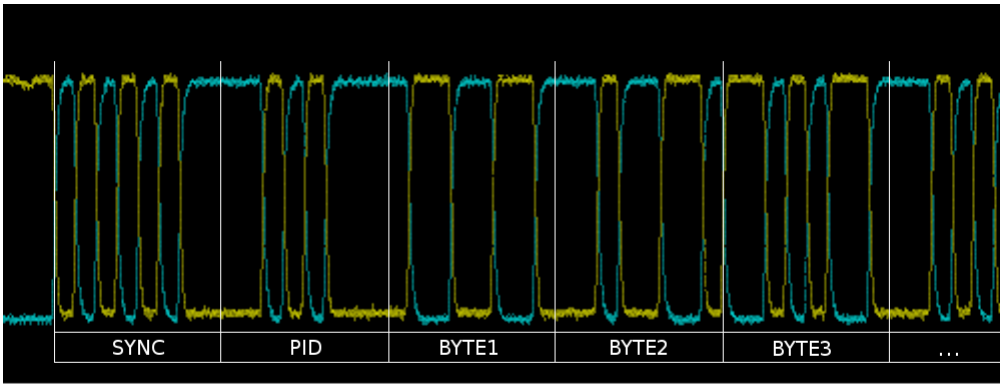
Auffällig ist jedoch das Fehlen von Adressierungsinformationen innerhalb eines DATA-Pakets. Diese sind in solchen Paketen nicht notwendig, da Sender beziehungsweise Empfänger bereits im Rahmen der Buszeitzuteilung durch ein vorangehendes SETUP-, OUT oder IN-Paket (mit entsprechenden Adressinformationen) festgelegt wurden[14].

[13] Siehe hierzu 2.4.4.1.
[14] Siehe hierzu Abschnitt 2.4.4.



KJKJKJKK	*KJKJKKKK*	*JJKJKJK*	*JJKJ*	*KJKJJ*
00000001	10000111	0100000	0100	00001
SYNC	*PID*	*ADDR*	*ENDP*	*CRC5*

Abbildung 2.16: OUT Token Paket



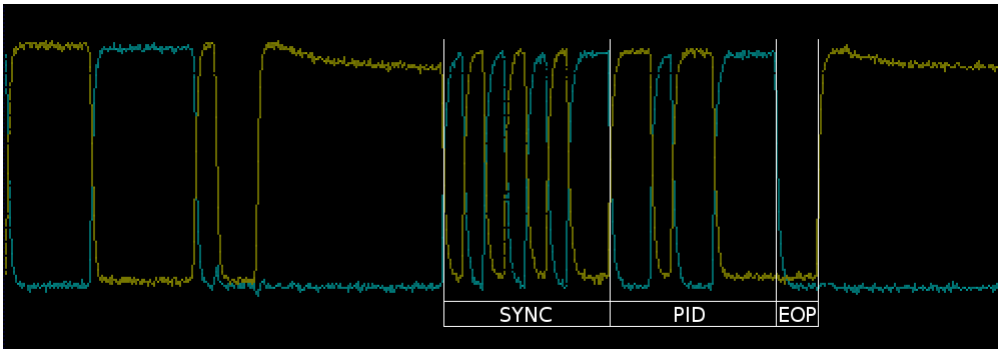
KJKJKJKK	*KKJKJKKK*	*KJJKKJJK*	*KKJKKJJK*	*JJKJKJJK*	...	
00000001	11000011	10101010	11001010	01000010	...	
SYNC	*PID*	*BYTE1*	*BYTE2*	*BYTE3*	...	*CRC16*

Abbildung 2.17: Beginn eines "geraden" Datenpaketes (DATA0)

2.4.3.9 ACK-Paket

Das ACK-Paket ist das erste betrachtete Paket aus der Gattung der Handshakepakete. Ein ACK-Paket besteht nur aus seiner PID und signalisiert später die erfolgreiche Übertragung eines DATA-Pakets. Abbildung 2.18 zeigt den

Aufbau und das Aussehen eines ACKs auf dem Bus.



$KJKJKJKK$	$JJKJJKKK$
00000001	01001011
$SYNC$	PID

Abbildung 2.18: ACK Handshake Paket

ACK-Pakete enthalten keinerlei Informationen über das Datenpaket, zu dessen Quittierung sie genutzt werden. Ihre Zugehörigkeit ergibt sich aus dem verwendeten Handshake-Verfahren und den Eigenschaften des USB.

2.4.3.10 NAK-Paket

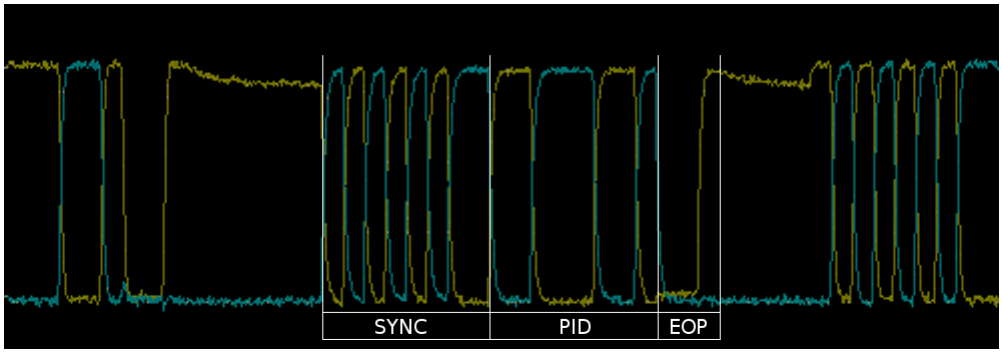
Wie das ACK-Paket besteht auch das in Abbildung 2.19 gezeigte NAK lediglich aus der ihm zugeordneten PID. NAK-Pakete können ausschließlich auf Geräteseite erzeugt werden und signalisieren dem Host eine temporäre Unfähigkeit des Gerätes, Daten zu transferieren. Besonders betont sei hierbei, dass es sich um eine *temporäre* Unfähigkeit des Gerätes handelt, die aktuelle Anfrage des Hosts zu behandeln – Ein Gerät, welches auf eine Anfrage des Hosts ein NAK sendet, wird also von allein wieder in einen transferbereiten Zustand gelangen.

Ein Beispiel ist hier eine Kamera, welche gerade mit der Aufnahme eines neuen Bildes beschäftigt ist. Sie hat für die Dauer der Aufnahme keine neuen Daten an den Host zu liefern. Sobald die Aufnahme komplett ist, wird das Gerät dem Host jedoch die Bilddaten zugänglich machen können.

Ein NAK zeigt also insbesondere keinen Fehlerzustand des Gerätes an, welchen der Host in irgendeiner Weise gesondert (neben einer neuen Datenanfrage an das Gerät) behandeln müsste.

Angemerkt sei, dass ein Gerät ein eingehendes SETUP-Paket beziehungsweise ein darauf folgendes Datenpaket niemals mit einem NAK beantwortet.

Dies ergibt sich aus der Anforderung heraus, dass Geräte zu jedem Zeitpunkt zur Behandlung eines SETUP in der Lage sein müssen.



KJKJKJKK	*JJKKKJJK*
00000001	01011010
SYNC	*PID*

Abbildung 2.19: NAK Handshake Paket

2.4.3.11 STALL-Paket

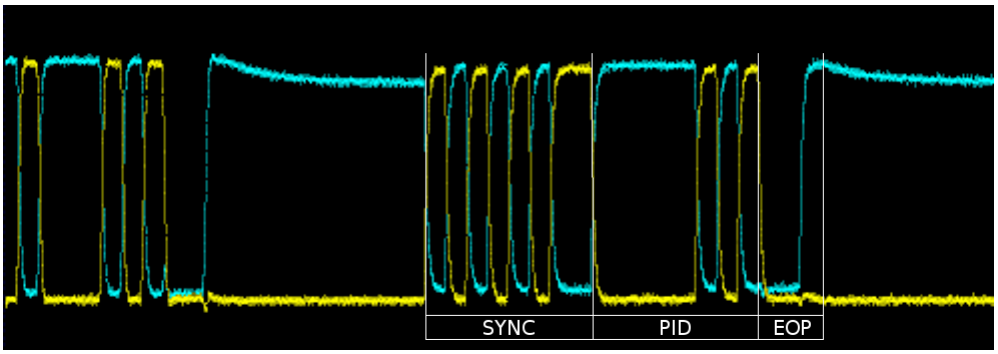
Auch das STALL wird ausschließlich geräteseitig generiert und signalisiert dem Host im Gegensatz zum NAK das Auftreten eines Fehlers, dessen Behebung ein Eingreifen von Seiten des Hosts erfordert. Zu einem solchen Fehler kann es beispielsweise kommen, wenn die Kommunikation mit dem Host einem bestimmten (nicht durch die USB-Spezifikation, sondern durch weitere Dokumente oder den Gerätehersteller selbst) definierten Protokoll folgen muss, dieses Protokoll jedoch durch die aktuelle Transaktion verletzt wird.

Welche Aktionen der Host zur Behebung des Fehlers ergreifen muss, ist nicht festgeschrieben. Die Reaktion auf ein STALL ist also gerätespezifisch.

Angemerkt sei auch hier, dass ein Gerät niemals mit einem STALL auf ein SETUP oder ein auf dieses folgende DATA-Paket antworten kann. Eine Instanz eines STALLs und dessen Aufbau ist in Abbildung 2.20 zu sehen.

2.4.3.12 Präambel für Lowspeed-Transfers

In der Vorstellung der Pakete wurden bis hierhin mit einer einzigen Ausnahme alle in Tabelle 2.1 eingetragenen PIDs verwendet. Bei dem nicht verwendeten PID handelt es sich um das PRE-PID, dessen Bedeutung im Folgenden noch erläutert werden muss.



KJKJKJKK	*JJJJJKJK*
00000001	01111000
SYNC	*PID*

Abbildung 2.20: STALL Handshake Paket

Zum Verständnis der Aufgabe des PRE-PIDs muss betrachtet werden, wie der USB den gleichzeitigen Betrieb von Geräten unterschiedlicher Geschwindigkeiten (Low- oder Fullspeed) ermöglicht. Sind Low- und Fullspeed-Geräte an den Bus angeschlossen, so ergibt sich das Problem, dass Lowspeed-Geräte den auf dem Bus stattfindenden Fullspeed-Datenverkehr (durch ein zu niedriges Abtastintervall) fehlinterpretieren könnten. In der Folge könnten sie dann eine nicht durch den Host beabsichtige Aktion durchführen, was letztlich die Funktion des gesamten Busses stören würde.

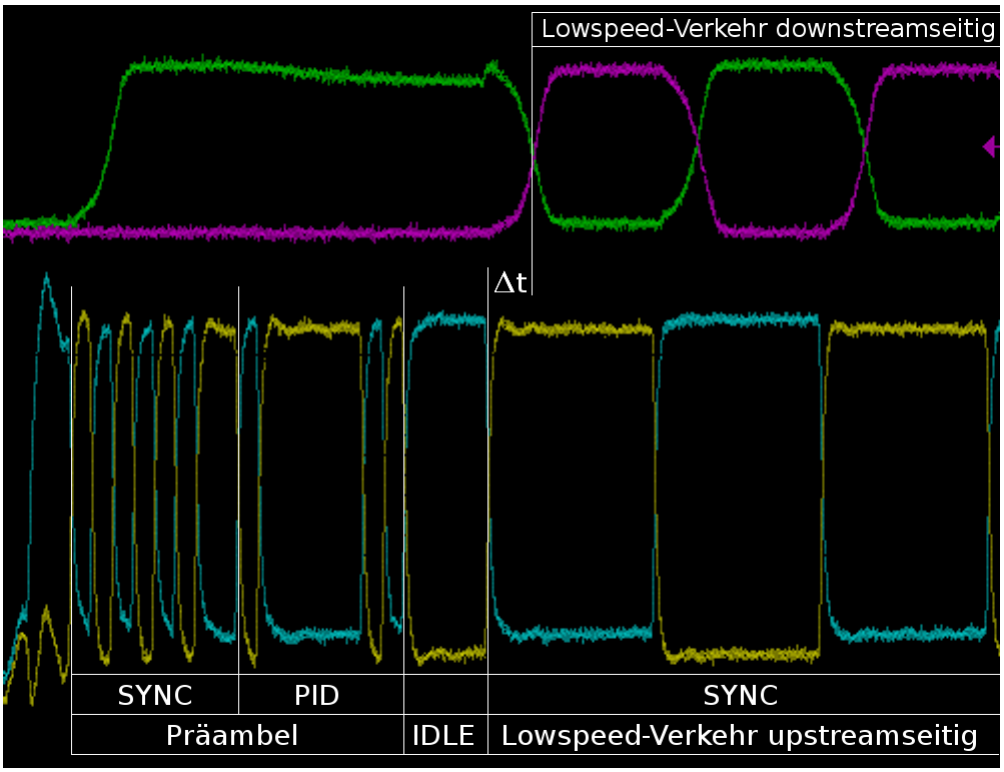
Es muss also dafür gesorgt werden, dass Lowspeed-Geräte während der Dauer von Fullspeed-Transaktionen den Bus lediglich im Ruhezustand vorfinden, so dass sie niemals mit Fullspeed-Signalen konfrontiert werden. Diese Aufgabe der "Abschottung" von Lowspeed-Geräten wird bei USB durch ein Zusammenspiel von Hubs und dem Host gelöst.

Der Host ist sich zu jedem Zeitpunkt über die Fähigkeiten eines jeden angeschlossenen Gerätes bewusst und kennt damit insbesondere auch dessen unterstützte Übertragungsgeschwindigkeit. Ist ein Paket zu einem Lowspeed-Gerät zu übertragen, so bringt der Host vor diesem Paket ein zusätzliches SYNC-Feld mit nachfolgender PRE-PID als Präambel auf den Bus.

Auch Hubs sind sich darüber im Klaren, ob ein mit ihnen verbundenes Gerät den Low- oder aber den Fullspeed-Modus unterstützt. An Ports, an welchen gerade ein Lowspeed-Gerät angeschlossen ist, werden nur solche Pakete weitergeleitet, denen die beschriebene Präambel vorangeht. Die Präambel sorgt also dafür, dass der Hub die Treiber der Ports im Lowspeed-Modus aktiviert und überhaupt Datenverkehr über diese weiterleitet. Mit

dem nächsten EOP (also dem Ende des auf die Präambel folgenden Paketes) deaktiviert der Hub den Downstreamport dann wieder und es findet (bis zur nächsten Präambel) keine weitere Signalleitung über diesen statt[15].

Der Betrieb von Low- und Fullspeed-Geräten zur gleichen Zeit ist folglich nur unter Zuhilfenahme des PRE-PIDs möglich.



KJKJKJKK	*JKKKKKJK*	*JJJJ*	*KJK*
00000001	00111100	0111	000
SYNC	*PID*	*IDLE*	*SYNC*

Abbildung 2.21: Präambel zur Weiterleitung von Lowspeed-Verkehr

Abbildung 2.21 zeigt eine Präambel für Lowspeed-Datenverkehr und die darauf folgende Reaktion eines Hubs auf dem Bus. Hierbei sind in der unteren Hälfte des Bildes die am Upstream-Port des Hubs eintreffenden Signale zu sehen. Die obere Hälfte zeigt die daraufhin erzeugten Pegel an einem der Downstream-Ports, an welchem gerade ein Lowspeed-Gerät angeschlossen ist.

[15]Eine Ausnahme ist die Umsetzung von SOF nach Keep-Alive, welche immer ohne Präambel vorgenommen wird.

Zunächst trifft am Upstream-Port die Präambel ein, woraufhin der Hub den betrachteten Downstream-Port aktiviert. Der Downstream-Port muss laut USB-Spezifikation innerhalb von vier Fullspeed-Bitzeiten in einen sendebereiten Zustand gebracht werden. Diese Wartezeit räumt der Host dem Hub also ein, bevor er mit dem Senden des Lowspeed-Paketes beginnt. Im Bild wurde der Zeitraum als IDLE markiert.

Gut zu erkennen ist auch, dass der Hub eine gewisse Verzögerung in der Signalleitung verursacht: Am Upstream-Port eingehende Signale erscheinen nicht sofort am Downstream-Port, sondern erst eine gewisse Zeit später, welche der Hub für die Weiterleitung benötigt. Diese Verzögerung ist als Δt markiert.

Durch das hier vorgestellte Verfahren ergibt sich insbesondere, dass Hubs keine Geschwindigkeitsumsetzung für Datenverkehr vornehmen. Es wird auf Basis der Präambel lediglich die Signalweiterleitung zu bestimmten Downstream-Ports unterbunden beziehungsweise erlaubt. Der Host muss selbst dafür Sorge tragen, Pakete für Lowspeed-Geräte mit den korrekten Bitzeiten zu generieren.

Hubs achten jedoch darauf, die logischen Zustände J und K korrekt zwischen Up- und Downstream-Port zu vermitteln: Ein Fullspeed-J wird also bei Weiterleitung durch den Hub in ein Lowspeed-J (bei K entsprechend) überführt. Es findet also während des Lowspeed-Verkehrs keine Invertierung der Pegel beim Signalverlauf zwischen Hub und Host statt.

2.4.4 Transferarten

An dieser Stelle folgt eine genauere Beschreibung der von USB zur Verfügung gestellten Transferarten, deren Eigenschaften und Realisierung.

2.4.4.1 Bulk Transfer

Mit Hilfe eines Bulk Transfers können Daten auf einfache Weise über den Bus übertragen werden, wenn diese Daten in größeren Mengen und mit hoher zeitlicher Variation auftreten. Ein Bulk Transfer garantiert die vollständige und korrekte Übertragung, wobei jedoch keinerlei Zeit- oder Bandbreitengarantien gegeben werden. Bei einem stark belasteten Bus kann die Abwicklung einer Datenübertragung mittels Bulk Transfers also einige Zeit in Anspruch nehmen.

Ein typisches Anwendungsbeispiel für diese Transferart ist die Anbindung eines externen Massenspeichers wie einer Festplatte.

Der Zugriff auf Bulk Transfers steht lediglich im Full- und im Highspeed-Modus zur Verfügung, Lowspeed-Geräte können also keine Datenübertragung

im Bulk-Modus betreiben. Die USB-Systemsoftware wickelt Bulk Transfers immer dann ab, wenn keine anderen Transferarten Buszeit beanspruchen. Sollten mehrere Bulk Transfers auf Komplettierung warten, so teilt die USB-Systemsoftware die zur Verfügung stehende Buszeit in fairer Weise auf die verschiedenen wartenden Bulk Transfers auf.

Transaktionsabwicklung Die Abwicklung von Bulk Transaktionen erfolgt nach einem Schema, welches in Abbildung 2.22 dargestellt ist.

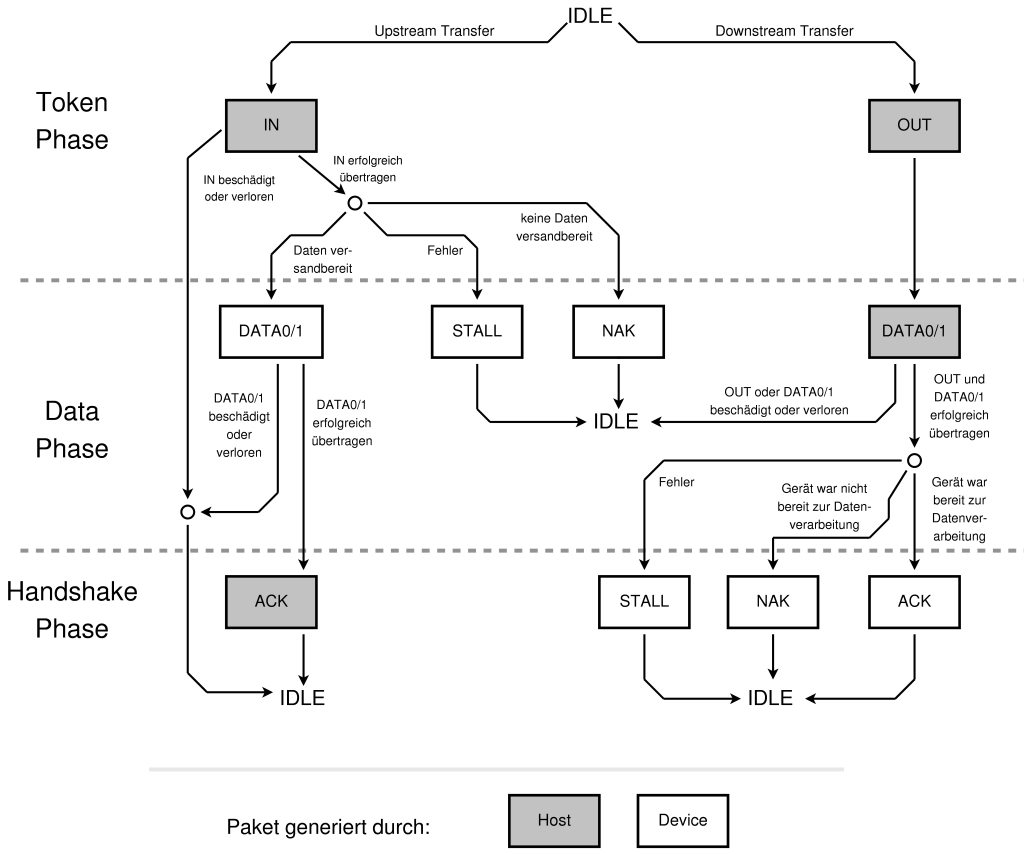


Abbildung 2.22: Ablaufdiagramm einer Bulk Transaktion

Zunächst ist zu entscheiden, ob Daten in Up- oder aber in Downstreamrichtung zu übertragen sind. Falls die Übertragung von Host zu Gerät erfolgen soll, so bringt der Host ein OUT-Paket auf den Bus, welches den empfangenden Endpunkt genau identifiziert. Anschließend sendet der Host ein Datenpaket, welches die zu übertragenden Daten enthält. Bei korrektem Empfang quittiert das Gerät den Erhalt des Paketes mit einem ACK, falls

sein aktueller Zustand die Bearbeitung der Daten zulässt. Daraufhin ist die Transaktion komplett und der Host kann nach Belieben mit weiteren Übertragungen fortfahren.

Sollte der Empfang zwar korrekt erfolgt sein, das Gerät aber aufgrund von mangelndem Speicherplatz oder ähnlichen temporären Problemen nicht zur Datenbearbeitung in der Lage sein, so zeigt es dies dem Host mit einem NAK an. Die Transaktion ist dann fehlgeschlagen und der Host muss diese daraufhin zu einem späteren Zeitpunkt wiederholen.

Soll die Datenübertragung von Gerät zu Host erfolgen, so fordert der Host die Daten vom Geräteendpunkt mittels eines mit entsprechender Adressierung versehenen IN-Paketes an. Sind auf Geräteseite Daten zum Versand bereit, so reagiert das Gerät mit einem Datenpaket, welches die zu übertragenden Daten enthält. Der korrekte Empfang dieses Paketes auf Hostseite wird dann durch den Host mit einem ACK quittiert und die Transaktion war erfolgreich.

Sind zum Zeitpunkt des Eintreffen des IN beim Gerät gerade keine Daten versandfertig, so antwortet das Gerät dem Host mit einem NAK – Auch hier gilt die Transaktion als fehlgeschlagen und der Host fordert die Daten zu einem späteren Zeitpunkt erneut an.

Im Falle eines Fehlers auf Geräteseite wird durch das Gerät ein STALL als Reaktion auf ein eintreffendes IN beziehungsweise die OUT/DATA-Kombination erzeugt. Die Transaktion ist dann fehlgeschlagen und der Host muss geeignete, gerätespezifische Maßnahmen treffen, um den Gerätefehler zu behandeln.

Funktion des Data-Toggle Eine in Bezug auf Datenverluste gesicherte Übertragung erfordert ein erneutes Senden von Paketen, welche nicht oder nur fehlerbehaftet beim Empfänger eintreffen. Schlägt eine einzelne Transaktion fehl, so muss diese also wiederholt werden. Gleichzeitig muss dabei aber auch die Reihenfolgeerhaltung der Pakete gewährleistet sein.

Hierzu findet bei USB ein einfaches Stop-and-Wait-Verfahren Anwendung. Erhält der Sender auf ein versandtes Datenpaket kein ACK, so geht er davon aus, dass das Paket verloren gegangen ist. Er muss die im Paket enthaltenen Daten daher zu einem späteren Zeitpunkt erneut versenden. Sollte bei der Übertragung jedoch nicht das Paket selbst, sondern nur das daraufhin vom Empfänger erzeugte ACK verloren gegangen sein, so könnte dieser bei einem einfachen Neuversand das wiederholte Paket nicht von einem tatsächlich neuen unterscheiden. Somit könnte es zu einer Verdopplung der eigentlich zu übertragenden Daten kommen.

Die einfache Lösung besteht hier in einer einbittigen Sequenznummer,

durch welche die Reihenfolgeerhaltung im Stop-and-Wait-Fall garantiert werden kann.

Die Sequenznummer befindet sich dabei etwas versteckt innerhalb der PID der Datenpakete. Jedes durch ein ACK als erfolgreich übertragen markierte Paket führt zu einer Änderung der PID von DATA0 nach DATA1 beziehungsweise umgekehrt. Das sich hierbei ändernde einzelne Bit, welches prinzipiell die Sequenznummer darstellt, wird wegen seines "Kipp-Verhaltens" auch Data-Toggle genannt.

Da der Host den Geräten jeweils einzeln Buszeit zuteilt und auch die maximal mögliche Round-Trip-Time für Pakete auf dem Bus durch die Spezifikation genau bekannt ist, kann auf eine Angabe der Sequenznummer im ACK problemlos verzichtet werden. Nähere Angaben zu den fraglichen Zeitbedingungen sind in Abschnitt 2.4.4.5 zusammengefasst.

Einschränkungen und Transferabschluss Jeder Host muss für eine Kommunikation mit Bulk-Endpunkten Payloadgrößen von 8, 16, 32 oder 64 Byte unterstützen. Kleinere oder größere Pakete können zwar durch den Host unterstützt werden, dies muss allerdings nicht der Fall sein. Im Falle der Unterstützung größerer Pakete ist die maximale Datenfeldlänge von 1023 Byte für Fullspeed-Datenpakete zu beachten.

Ist eine Datenmenge mit einem Bulk Transfer zu übertragen, welche nicht innerhalb eines Paketes unterzubringen ist, so werden die Daten auf mehrere Pakete und damit auf mehrere Transaktionen aufgeteilt. Während dies in Downstreamrichtung keine Probleme verursacht, da der Host die genaue zu sendende Datenmenge kennt und demnach eine passende Anzahl von OUT-Tokens (und darauf folgender Datenpakete) erzeugen kann, so ist die Erzeugung einer genau passenden Anzahl von IN-Tokens bei Übertragung in Upstream-Richtung nicht immer möglich.

Der Abschluss eines Transfers in Upstreamrichtung tritt immer dann ein, wenn der Host entweder die von ihm maximal gewünschte Datenmenge vom Gerät erhalten hat oder aber das Gerät den Abschluss der Übertragung mit einem sogenannten Underlength-Packet signalisiert. Hierbei handelt es sich um ein Paket einer Länge, welche die maximale Payloadlänge für den angesprochenen Endpunkt unterschreitet. Insbesondere gehört auch das Paket der Payloadlänge 0 zur Gruppe solcher Pakete. Ein Paket der Länge 0 wird im Folgenden als ZLP (Zero Length Packet) bezeichnet.

2.4.4.2 Interrupt Transfers

Bei Interrupt Transfers stellt der Host dem Gerät eine Latenzgarantie bereit, so dass das Gerät kleinere Datenmengen in genau definierten Intervallen vom

Host erfragen oder zu diesem übertragen kann.

Es ist zu beachten, dass es sich bei Interrupt Transfers aufgrund der Funktionsweise des USB trotz der Namensgebung dieser Transferart nicht um einen interruptgenerierenden Mechanismus handelt, bei welchem ein Gerät von sich aus einen Transfer beginnen würde, um den Host auf das Vorliegen neuer Daten hinzuweisen. Auch bei den Interrupt Transfers muss der Host in seiner Eigenschaft als Busverwalter die Datenübertragung anstoßen. Durch diese Transferart ist es jedoch für Geräte garantiert, dass sie eventuell zur Übertragung bereitstehende Daten mit einer definierten Latenz zum Host übertragen oder aber in festen Zeitabständen Daten von diesem erhalten können.

Gängige Beispiele für Geräte, welche Interrupt Transfers benötigen, sind Eingabegeräte wie Mäuse oder auch Gamepads. Diese müssen dem Host ihre Eingabedaten in Echtzeit – also gerade mit definierter Latenz – bereitstellen können. Bei Beginn des Betriebs eines solchen Gerätes handelt der Host ein Abfrageintervall mit dem Gerät aus, welches dann für den Rest der Betriebsdauer eingehalten wird[16]. Typisch ist bei derartigen Eingabegeräten beispielsweise eine Intervalldauer von 10ms, so dass im Rahmen der Wahrnehmung des Anwenders sofort eine Reaktion auf dessen Eingaben erfolgt.

Transaktions- und Transferabwicklung Zur Latenzeinhaltung bringt der Host in dem vorher mit dem Gerät fest ausgehandelten Zeitintervall ein IN- oder OUT-Token (eventuell gefolgt von den für das Gerät bestimmten Daten) auf den Bus, um dem Gerät eine Übertragungsmöglichkeit zu bieten.

Die Gerätereaktion auf dieses Token kann genauso wie bei Bulk Transaktionen erfolgen – ACK, NAK und STALL haben also die gleiche Funktion wie im Bulk-Fall. Die Hostreaktion auf ein vom Gerät geliefertes NAK führt jedoch nicht zu einer Neuanfrage von Daten im gleichen Intervall. Vielmehr wird in diesem Fall davon ausgegangen, dass das Gerät im fraglichen Intervall keine Daten zu liefern hat und eine weitere Abfrage erst im nächsten Intervall erfolgen muss.

Transferablauf und Fehlerbehandlung Die Transferabwicklung verhält sich weitgehend identisch zu der bei Bulk Transfers, mit der Ausnahme, dass es bei einer fehlschlagenden Transaktion erst im nächtsten Intervall zu einem neuen Übertragungsversuch kommt. Bei Interrupt Transfers mit Full- und Lowspeed-Geräten achtet die USB-Systemsoftware darauf, dass niemals

[16] Dies bezieht sich auf die zur Verfügung stehende Buszeit. Sollte der das Gerät bedienende Treiber beispielsweise wegen Fehlfunktion keine Daten vom Gerät entgegennehmen wollen oder keine an dieses zu liefern haben, so findet auch keine Übertragung statt.

mehr als eine Transaktion pro Frame stattfindet. Ein Transfer kann jedoch ebenfalls aus mehreren Transaktionen bestehen, welche dann in entsprechendem zeitlichen Abstand erfolgen.

Insbesondere findet jedoch auch hier das Verfahren des Data-Togglings Anwendung, der Transferabschluss bei höherer Datenmengen wird ebenfalls identisch zu Bulk gehandhabt.

Buszugriffsregeln und Einschränkungen Interrupt Transaktionen stehen Geräten aller Geschwindigkeitsklassen zur Verfügung. Die für Lowspeed-Geräte maximal zur Verfügung stehende Payloadgröße pro Paket beträgt 8 Byte, für Fullspeed-Geräte sind 64 Byte vorgesehen[17].

Fullspeed-Geräte können dem Host Abfrageintervalle zwischen einer und 255ms abverlangen, bei Lowspeed-Geräten sind Intervalle zwischen 10 und 255ms vorgesehen. Die USB-Systemsoftware berechnet aufgrund der maximalen Paketgröße des Interrupt-Endpoints die voraussichtlich erforderliche Buszeit und sorgt dafür, dass niemals mehr als 90% eines Frames für periodische Transfers verplant werden. Anzumerken ist dabei, dass zu den periodischen Transfers neben denen des Typs "Interrupt" auch noch solche des Typs "Isochronous" gehören. Stellt die USB-Systemsoftware fest, dass sich die durch das Gerät verlangten Intervalle nicht einhalten lassen, so wird der Betrieb des entsprechenden Endpunktes verweigert.

Generell handelt es sich bei der Berechnung der USB-Systemsoftware jedoch nur um die oberere Grenze der Busauslastung, da die tatsächlich zu übertragende Datenmenge erst zu Transferbegin durch den Sender festgelegt wird. Die Pakete können also kleiner sein, als die maximale Paketgröße des Endpunktes es erlauben würde.

Der USB-Systemsoftware steht es frei, das Abfrageintervall für Geräte nach unten zu korrigieren, die Angabe des Gerätes bezieht sich also nur auf die maximal akzeptierbare Latenz. Die USB-Systemsoftware wird jedoch niemals ein Abfrageintervall vorsehen, welches die Zeitdauer eines Frames unterschreitet.

2.4.4.3 Isochronous Transfers

Isochronous Transfers garantieren dem Gerät die ständige Verfügbarkeit einer bestimmte Bandbreite zur Datenübertragung.

Transaktionsabwicklung Abbildung 2.23 zeigt den möglichen Ablauf von Isochronous Transaktionen. Je nach gewünschter Datenrichtung bringt der

[17] Highspeed-Geräte sind auf 1024 Byte eingeschränkt.

Host ein IN- oder OUT-Token auf den Bus. In Abhängigkeit dieses Tokens antwortet das angesprochene Gerät entweder mit einem DATA-Paket oder aber empfängt ein durch den Host generiertes DATA-Paket.

Wie die Abbildung zeigt, ist keine Handshake-Phase vorgesehen. Auf eine garantierte Vollständigkeit der übertragenen Daten wird bei Isochronous Transfers zu Gunsten der durch das Auslassen der Handshake-Pakete freiwerdenden Buszeit (und damit Bandbreite) verzichtet.

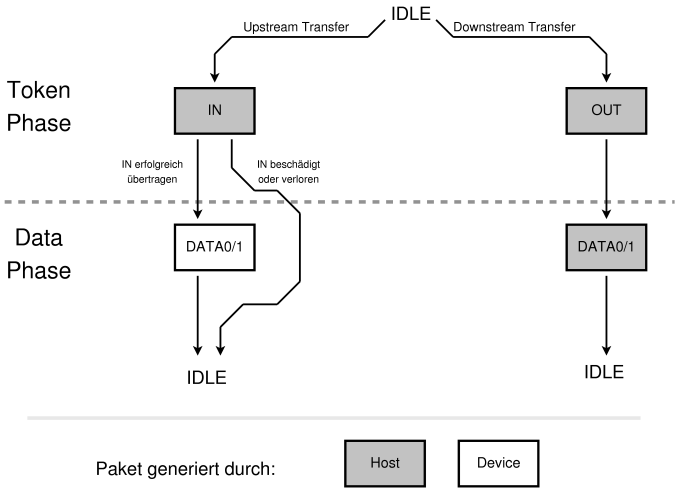


Abbildung 2.23: Ablaufdiagramm einer Isochronous Transaktion

Buszugriffsregeln und Einschränkungen Isochronous Transfers stehen nicht für Lowspeed-Geräte zur Verfügung. Für Fullspeed-Geräte ist eine maximale Payloadgröße pro Paket von 1023 Byte vorgesehen. Das Gerät handelt zunächst ein Abfrageintervall mit dem Host aus, wobei hier jedoch ein zu den Interrupt Transfers verschiedener Mechanismus zum Einsatz kommt[18]. Der Host ermittelt auf Basis der Periode und maximalen Paketgröße die zu reservierende Bandbreite. Dabei achtet die USB-Systemsoftware wieder darauf, die 90%-Grenze für periodische Transfers nicht zu überschreiten. Kann dies nicht garantiert werden, so wird der Betrieb des fraglichen Endpoints unterbunden.

Isochronous-Endpunkte müssen dazu in der Lage sein, mit einem kürzeren Abfrageintervall als dem von ihnen verlangten umzugehen. Ein Isochronous-

[18]Siehe hierzu auch Seite 60 (Felder *wMaxPacketSize* und *bInteval* des Endpoint Descriptors).

IN-Endpunkt, der zum Abfragezeitpunkt keine Daten zu liefern hat, muss dies dem Host immer durch das Senden eines ZLP anzeigen.

Das Verfahren des Data-Togglings findet keine Anwendung, beide Kommunikationspartner müssen zu jedem Zeitpunkt sowohl DATA0- als auch DATA1-Pakete als gültige Datenpakete annehmen. Die Spezifikation empfiehlt jedoch, beim Versand ausschließlich DATA0-Pakete zu verwenden.

2.4.4.4 Control Transfer

Control Transfers werden hauptsächlich durch die USB-Systemsoftware genutzt, um Status- und Konfigurationsinformationen mit Geräten auszutauschen. Ein Beispiel für Aufgaben, welche mit Hilfe von Control Transfers erledigt werden, ist die Adressvergabe an Geräte. Aufgrund der Wichtigkeit dieser Aufgaben ergeben sich für Control Transfers spezielle Anforderungen.

Der Control Transfer ist der komplexeste der betrachteten Transfertypen und auch der einzige, der über eine Message Pipe abgewickelt wird. Hierdurch ergibt sich nicht nur für den Ablauf von Transaktionen und Transfers sondern auch für das Format der zu übertragenden Daten eine komplette Definition durch die USB-Spezifikation. Insbesondere sei daran erinnert, dass es sich bei Message Pipes um bidirektionale Pipes handelt, so dass Daten während eines Transfers sowohl in Upstream- als auch in Downstreamrichtung übertragen werden können.

Transferaufbau und Transaktionsablauf Control Transfers bestehen aus zwei oder drei Stufen, welche sich jeweils wieder in die bereits gesehenen Token-, Data- und Handshake-Phasen aufteilen. Auch wenn die Namensgebung der Stufen und Phasen (aufgrund der verwendeten Pakete) sehr ähnlich sind, sei an dieser Stelle klar gesagt, dass Phasen und Stufen im Rahmen von USB zwei verschiedene Begrifflichkeiten sind, deren Bedeutung klar zu unterscheiden ist. Phasen sind Unterteilungen einzelner Transaktionen, in welchen jeweils ein Paket über den Bus versendet wird. An einer Stufe wiederum sind ein oder mehrere Transaktionen beteiligt, wobei sich die Art der durchgeführten Transaktionen je nach Stufe unterscheiden.

Die erste Stufe eines jeden Control Transfers ist die sogenannte Setup-Stufe, in welcher der Host den Verlauf des Transfers genau spezifiziert und dem Gerät bekannt macht. Abhängig von den in der Setup-Stufe übertragenen Daten folgt dann eine Daten-Stufe, in welcher weitere den Transfer betreffende Daten mit dem Gerät ausgetauscht werden. Abgeschlossen wird jeder Control Transfer durch die sogenannte Status-Stufe, in welcher die erfolgreiche Behandlung des Transfers durch das Gerät bestätigt wird.

Setup-Stufe In der Setup-Stufe findet eine einzelne Transaktion statt, welche mit Hilfe eines SETUP-Paketes eingeleitet wird. Das Schema einer solchen Transaktion ist in Abbildung 2.24 gezeigt.

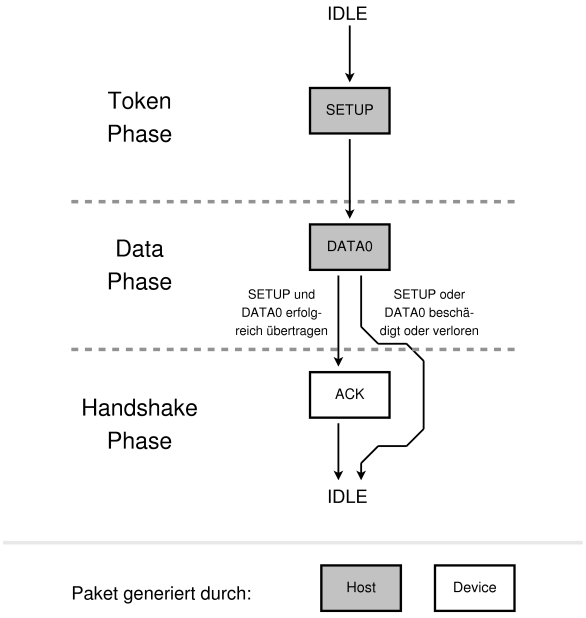


Abbildung 2.24: Schema einer Setup-Transaktion

Das SETUP-Paket ist ein analog zum OUT-Token eingesetztes Paket, in welchem der Kommunikationspartner des Hosts für den Control Transfer adressiert ist. Control Transfers werden mit Hilfe des Data-Toggles gesichert, wobei das SETUP-Paket für eine Initialisierung des Toggles mit Null sorgt. Das in der Transaktion unmittelbar nach SETUP folgende Datenpaket ist daher immer vom Typ DATA0. Die in diesem Datenpaket enthaltenen Request-Informationen belegen immer eine Payload von acht Byte; Aufbau und Funktion dieser Informationen werden später in 2.4.5.3 beschrieben.

Abgeschlossen wird die Transaktion mit einem ACK-Handshakepaket, welches den korrekten Empfang des Datenpaketes auf Geräteseite anzeigt. Die Setup-Stufe ist mit Beenden dieser einzelnen, durch das SETUP-Paket eingeleiteten, Transaktion ebenfalls vollendet.

Wie beschrieben kann das Gerät in der Setup-Stufe nicht mit einem STALL oder NAK reagieren. Sollten die in der Setup-Stufe zu übertragenden Daten fehlerhaft oder gar nicht beim Gerät angekommen sein, so bleibt das ACK einfach aus.

Daten-Stufe In Abhängigkeit des durch die Setup-Stufe vorgegebenen weiteren Verlaufs folgt nun die Daten-Stufe. Hierbei wird eine Anzahl Bytes, welche in der Setup-Stufe spezifiert wurde, entweder in Upstream- oder in Downstreamrichtung übertragen. Die Datenübertragung folgt dabei den Regeln des Bulk Transfers und besteht – abhängig von der zu übertragenden Datenmenge – aus ein oder mehreren mittels IN- oder OUT-Paket eingeleiteten Transaktionen. Anzumerken ist, dass das erste versandte Datenpaket der Daten-Stufe immer vom Typ DATA1 ist.

Control Transfers, in welchen die Daten-Stufe in Downstreamrichtung erfolgt, werden auch Control Writes genannt. Erfolgt die Daten-Stufe in Upstreamrichtung, so heißt der Transfer entsprechend Control Read. Control Transfers ohne Daten-Stufe werden als No-Data Control bezeichnet.

Status-Stufe Der Beginn der Status-Stufe wird durch eine Umkehr der Datenrichtung im Verhältnis zur vorangegangenen Stufe angezeigt und besteht immer aus der Übertragung eines ZLP.

War die Daten-Stufe zum Gerät hin gerichtet (OUT) oder nicht vorhanden, so sendet der Host also ein IN-Token, woraufhin das Gerät bei erfolgreicher Abwicklung des Transfers mit einem ZLP antwortet. Sollte das Gerät in einen Fehlerfall geraten sein, so antwortet es mit einem STALL. Ist es zum Zeitpunkt des Eintreffens des IN noch mit der Behandlung der durch den Control Transfer angestoßenen Aktion beschäftigt, so sendet es ein NAK. Die Daten-Stufe dauert darauf hin an und der Host wiederholt das IN zu einem späteren Zeitpunkt.

Handelt es sich bei dem Control Transfer um einen Control Read, so sendet der Host ein OUT-Token und bringt daraufhin sofort ein ZLP auf den Bus. Die Reaktion des Gerätes auf das ZLP gibt, in zum Control Read beziehungsweise No-Data-Control identischer Weise, den Status des Gerätes wieder.

Das in der Status-Stufe übertragene ZLP ist immer ein Paket des Typs DATA1. Tabelle 2.2 fasst die möglichen Gerätereaktionen noch einmal zusammen.

Buszugriffsregeln und Einschränkungen Bezüglich der zu unterstützenden Paketgrößen gilt für Fullspeed-Geräte auch bei Control Transfers eine Einschränkung auf 8, 16, 32 oder 64 Byte Payloadgröße. Lowspeed-Geräte sind auf 8 Byte Nutzdaten pro Paket limitiert.

Für Control Transfers stehen immer mindestens 10% der gesamten Buszeit zur Verfügung. Sind durch periodische Transfers weniger als 90% der Framezeit reserviert, so können Control Transfers auch in dieser unzugewie-

Statusmeldung	Control Write	Control Read
Erfolgreich	Zero Length Paket	ACK Handshake
Fehler	STALL	STALL
Beschäftigt	NAK	NAK

Tabelle 2.2: Status Reports

senen Zeit abgehandelt werden. Insbesondere hat die Abwicklung von Control Transfers Vorrang vor der Abwicklung von Bulk Transfers, welche nur die Restzeit des Busses beanspruchen können.

Abbildung 2.25 zeigt abschließend den Ablauf aller möglichen Arten von Control Transfers.

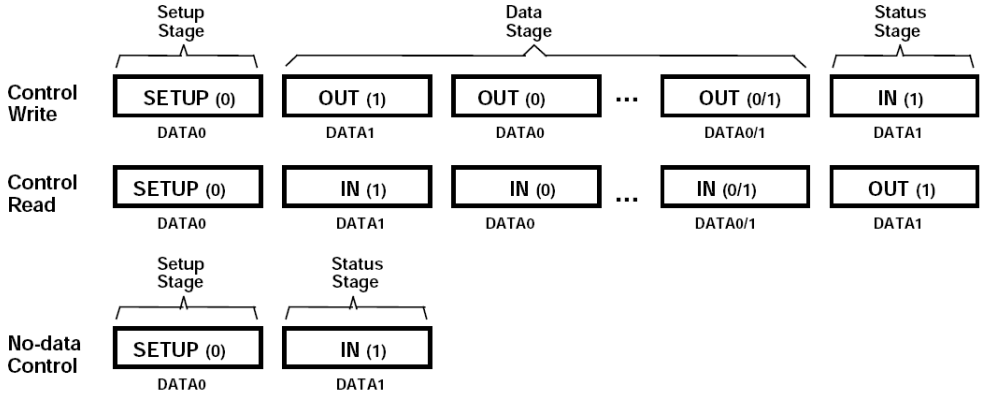


Abbildung 2.25: Ablauf verschiedener Control Transfers

2.4.4.5 Wartezeiten

Sowohl Geräte als auch der Host müssen den Bus zwischen zwei Paketen für mindestens zwei Bitzeiten der aktuellen Transfergeschwindigkeit im Ruhezustand belassen. Das Delay zwischen zwei Paketen einer Transaktion darf jedoch bei Geräten mit fest installiertem Kabel 7,5 Bitzeiten, bei allen anderen Geräten 6,5 Bitzeiten nicht überschreiten[19].

[19] Jeweils gemessen an der ersten von außen zugänglichen Stelle, also der Steckverbindung.

Bei Paketen, auf die eine unmittelbare Antwort vom Kommunikationspartner erwartet wird, kann ein wartendes USB-Gerät bereits nach 16 Bitzeiten, muss jedoch nach 18 Bitzeiten ohne Eintreffen der erwarteten Antwort vom Host von einem Verlust des Paketes ausgehen. Bleibt auf Hostseite eine erwartete Antwort eines Gerätes aus, so wird der Host nach 18 Bitzeiten eine Neuübertragung des dann als verloren anzusehenden Paketes starten, falls der Transfertyp dies erfordert.

Im Fullspeed-Fall entsprechen 16 Bitzeiten einer Dauer von $16 * \frac{1}{12*10^6} s = 1,\overline{3}\mu s$. Bei einer maximalen Tiertiefe von sieben und Worst-Case Annahmen für die durch Kabel und Hubs verursachten Delay-Zeiten ergibt sich die in Abbildung 2.26 dargestellte Situation.

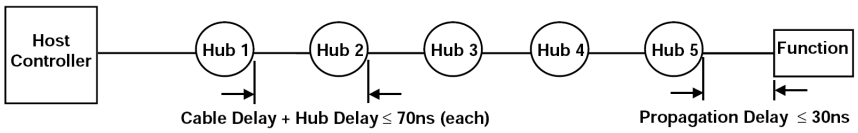


Abbildung 2.26: Worst-Case Situation für den Busaufbau

Unter der Annahme, dass jeder Hub inklusive Kabel ein Propagation Delay von 70ns erzeugt und die damit noch nicht berücksichtigten Kabel- und Anschlussstrecken 30ns betragen, ergibt sich ein maximal mögliches Delay von $5*70ns+30ns = 380ns$. Die Round-Trip-Time beträgt damit $2*380ns = 760ns$ zuzüglich der Reaktionszeit des Gerätes beziehungsweise des Hosts. Diese kann dabei im gezeigten Fall bei einem Fullspeed-Gerät maximal $6,5 * \frac{1}{12*10^6} s = 541,\overline{6}ns$ betragen. Damit ergibt sich eine Worst-Case Round-Trip-Time, welche mit $1301,\overline{6}ns$ unterhalb des 16 Bitzeiten Timeouts liegt, so dass nach Ablauf dieses Timeouts tatsächlich sicher von einem Paketverlust ausgegangen werden kann.

2.4.5 Geräte und Geräteverwaltung

Nachdem vorangehend der Ablauf verschiedener Transfers und damit der Ablauf der Datenübertragung über den Bus erläutert wurde, müssen nun Geräte und deren mögliche Zustände hinsichtlich der USB-Funktionalität betrachtet werden. Wichtigkeit hat hier insbesondere der Mechanismus, über welchen Geräte dem Host ihre Fähigkeiten und Status sowie ihre Anforderungen an den Bus (beispielsweise Bandbreite) mitteilen. Darauf aufbauend kann der Host schließlich das Gerät konfigurieren und in Betrieb nehmen.

2.4.5.1 Descriptoren

Die Fähigkeiten und Eigenschaften eines USB-Geräts werden in wohldefinierten Datenstrukturen aufgezeichnet, welche das Gerät dem Host auf Anfrage mitteilt. Diese Datenstrukturen werden Descriptoren genannt.

Verschiedene Informationen sind – entsprechend ihrer logischen Zusammengehörigkeit – in verschiedenartigen Descriptoren zusammengefasst. Jeder Descriptor besitzt dabei mehrere Felder, deren Inhalt über den Feldnamen zugreifbar ist (vergleiche auch: C-Struktur oder relationale Datenbank).

Auch die Descriptoren selbst können zueinander in Bezug stehen. Üblicherweise gruppiert dabei ein bestimmter Descriptor ein oder mehrere andere Descriptoren zu weiteren logischen Einheiten zusammen, wodurch eine baumartige Struktur entsteht. Um vorweg einen Eindruck dieses Konzepts zu vermitteln, zeigt Abbildung 2.27 eine Gruppierung, wie sie mit Hilfe einiger durch die USB-Spezifikation definierter Descriptoren möglicht ist.

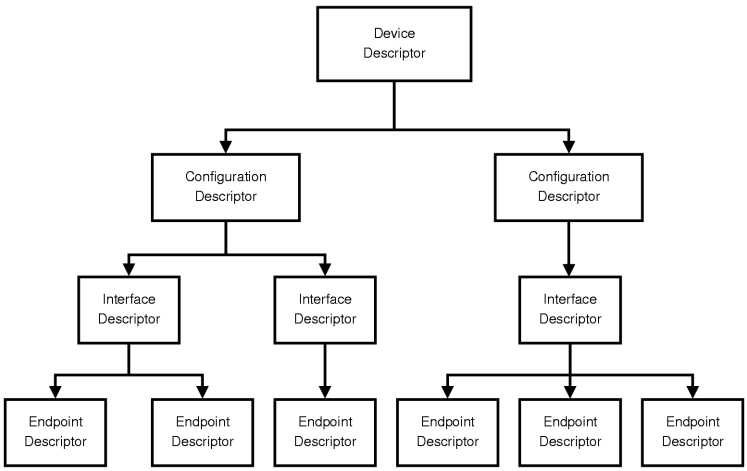


Abbildung 2.27: Beispielhafter Descriptorbaum

Die einzelnen verwendeten Descriptoren werden im Folgenden erläutert. Dabei wird nur auf die durch die USB-Spezifikation definierten Standard-Descriptoren eingegangen. Neben diesen können weitere Descriptoren durch weiterführende Dokumente definiert werden.

Descriptor-Header Allen diesen Descriptoren gemeinsam ist der Descriptor-Header, welcher Informationen über den Descriptor-Typ und die Anzahl der im Descriptor enthaltenen Bytes enthält. Fragt der Host Descriptoren vom Gerät ab, so wird in der Mehrheit der Fälle nicht nur ein einzelner,

Offset	Feld	Bytes	Wert
0	bLength	1	Nummer
1	bDescriptorType	1	Konstante
2	bEndpointAddress	1	Nummer
3	bmAttributes	1	Bitfeld
4	wMaxPacketSize	2	Nummer
6	bInterval	1	Nummer

Tabelle 2.3: Format des Endpoint Descriptors

sondern eine ganze Reihe von Descriptoren in einem Bytestrom übertragen. Die Längenangabe macht es möglich, das Offset des nächsten Descriptors innerhalb des Bytestroms direkt zu bestimmen, auch wenn der Typ des gerade vorgefundenen Descriptors unbekannt ist. Dies ist beispielsweise dann der Fall, wenn es sich bei dem fraglichen Descriptor um eine durch besagte weitere Dokumente definierte Struktur handelt, welche von der auf dem Host ausgeführten Software nicht unterstützt wird.

Endpoint Descriptor

Informationen über die Endpunkte eines Gerätes sind in Endpunkt Descriptoren zusammengefasst. Eine Ausnahme bilden hier allein die Endpunkte 0, deren Charakteristika dem Host auf andere Weise bekannt gemacht werden.

Alle Eigenschaften eines Endpunktes, wie Transfertyp, Puffergröße, Nummer und Richtung sind im Endpunkt Descriptor enthalten. Tabelle 2.3 zeigt den Aufbau von Endpunkt Descriptoren, die enthaltenen Felder sind nachfolgend beschrieben.

bLength Im Falle des Endpoint Descriptors beträgt die Descriptorlänge sieben Byte.

bDescriptorType Endpunkt Descriptoren sind durch den Wert 0x05 gekennzeichnet.

bEndpointAddress Dieses Feld gibt neben der Nummer des Endpunktes auch dessen Datenrichtung an. Dabei gilt:

Bit	**Funktion**
3..0	Endpunkt-Nummer
6..4	Reserviert, immer 0
7	Datenrichtung: 0: Downstream ("OUT-Endpunkt") 1: Upstream ("IN-Endpunkt")

Zu beachten ist, dass sich die Datenrichtungsangabe immer auf die Sichtweise des Hosts bezieht.

Ein OUT-Paket kündigt immer ein folgendes Datenpaket von Host zu Gerät an. Dementsprechend ist ein OUT-Endpunkt immer ein Endpunkt, über welchen Daten zum Gerät hin transferiert werden, auch wenn die Namensgebung in Verbindung mit dem Gedanken, dass ein Endpunkt auf Geräteseite angesiedelt ist, genau das Gegenteil suggeriert. Analog zu diesem Sachverhalt wird ein IN-Endpunkt immer genutzt, um Daten vom Gerät zum Host zu übertragen.

Im Folgenden ist auf diese Art der Namensgebung besonders zu achten.

bmAttributes Dieses Feld gibt den Endpunkttyp an. Zu dessen Codierung werden nur die zwei niederwertigsten Bit genutzt.

00	Control	**10**	Bulk
01	Isochronous	**11**	Interrupt

Für alle Transferarten mit Ausnahme von Isochronous müssen die restlichen Bits dieses Feldes auf Null gesetzt sein.

Für Isochronous Endpunkte sind durch Bit3..2 zusätzliche Optionen vorgesehen, um dem Host Hinweise zur Synchronisation der Datenrate mit dem Endpunkt zu geben. In diesem Zusammenhang bieten Bit5..4 ferner die Möglichkeit, verschiedene Isochronous Endpunkte zusammenzugruppieren. Einer der an der Gruppenbildung beteiligten Endpunkte kann dann dazu genutzt werden, um dem Host weitere Informationen bezüglich der durch die restlichen Endpunkte der Gruppe erzeugten Datenraten zu liefern.

Diese Mechanismen bleiben im Weiteren ungenutzt. Diesbezügliche Detailinformationen können der USB-Spezifikation in Abschnitt 5.12.4 sowie 9.6.6 entnommen werden.

wMaxPacketSize Für Full- und Lowspeed-Geräte sind nur die unteren elf Bit dieses Feldes zu verwenden. In diesen ist die maximale verwendbare Paketgröße für den Endpunkt angegeben.

Bei einem Isochronous-Endpunkt wird dieser Wert (in Verbindung mit *bInterval*) auch für die Bandbreitenreservierung genutzt.

In jedem Fall können Pakete jedoch auch kleiner als die hier angegebene Puffergröße sein.

bInterval Für Bulk- und Control-Endpunkte ist dieses Feld ohne Belang.

Bei Interrupt-Endpunkten geben Geräte hier das längste tolerierbare Pollingintervall in Millisekunden an, welches der Host für den Endpunkt einhalten muss. Für Fullspeed-Geräte sind Werte zwischen 1 und 255, für Lowspeed-Geräte solche zwischen 10 und 255 möglich.

Bei Isochronous-Endpoints dient das *bInterval*-Feld der Bandbreitengarantie für den Endpunkt. Es enthält ebenfalls ein Pollingintervall für Kommunikation mit dem Endpunkt, hierbei jedoch in der Form $maxInterval = 2^{bInterval-1}ms$. Für Isochronous-Endpoints gilt eine Beschränkung des *bInterval*-Feldes auf Werte zwischen 1 und 16. Die Datenrate, welche der Host für die Nutzdaten eines Fullspeed-Isochronous-Endpunktes garantiert, errechnet sich damit wie folgt:

$$maxBitRate = 8 * wMaxPacketsize * \frac{1000}{2^{bInterval-1}} bit/s$$

Interface Descriptor

Endpunkte sind ihrer logischen Funktion nach zu sogenannten Interfaces zusammengefasst.

Jeder Endpunkt ist zunächst für sich allein gesehen eine unabhängige Ein- und Austrittsstelle für Daten des Gerätes, jedoch erfordert eine sinnvolle Anwendung eines Gerätes meist das Zusammenspiel mehrerer Endpunkte. Erst durch Interfaces ergibt sich eine Zusammengruppierung von verschiedenen Endpunkten, durch deren Gesamtheit das Gerät dem Host seine Fähigkeiten zur Verfügung stellt.

Als Beispiel zum besseren Verständnis dieses Prinzips können die sogenannten Mass-Storage-Geräte herangezogen werden, also Festplatten, Flashspeicher und dergleichen mit USB-Anschluss. Bei allen diesen Geräten ist für einen ordnungsgemäßen Betrieb sowohl ein Datenstrom in Up- als auch in Downstreamrichtung notwendig, wobei die verschiedenen dazu genutzten Endpunkte geräteintern zusammen arbeiten müssen. Fragt der Host beispielsweise über einen OUT-Endpoint Daten von einem externen Massenspeicher

an, so müssen diese mittels eines korrespondierenden IN-Endpunktes zum Host gesendet werden. Ein Endpunkt für sich gesehen wäre in diesem Fall wertlos, erst beide zusammen ergeben eine Funktion des Gerätes. Daher ist es sinnvoll, diese beiden Endpunkte zu einem Interface zusammen zu fassen.

Neben Endpunkt Descriptoren können weitere Descriptorarten einem Interface angehören, die dann jedoch durch weiterführende Dokumente spezifiziert sind.

Im Bytestrom, welchen der Host bei der Descriptor-Abfrage vom Geräte erhält, sind alle Endpunkt- sowie auch alle nicht direkt durch die USB-Spezifikation definierten Descriptoren einem vorangegangen Interface Descriptor zuzurechnen. Die Reihenfolge der einem Interface Descriptor folgenden Descriptoren ist nicht näher festgeschrieben.

Geräte können verschiedene Interfaces besitzen, welche unabhängig voneinander eine eigene Funktion bereitstellen und dabei gleichzeitig aktiv sind. Ein Beispiel dafür ist eine Kamera, die einen Videostrom zur Verfügung stellt, es dem Anwender aber gleichzeitig ermöglicht, über eine Mass-Storage-Funktionalität auf vorher gespeicherte Bilder zuzugreifen.

Möglicherweise bietet die Kamera aber auch verschiedene Kompressionsstufen oder Auflösungen für den Live-Videostrom, so dass sich je nach Betriebsmodus verschiedene Bandbreitenanforderungen oder ähnliches für das Gerät ergeben. Funktion und Komponenten sind in diesem Fall jeweils identisch, nur die Anforderung an den Bus beziehungsweise an die Datenüberragung ändern sich. Insbesondere ist es nicht sinnvoll, gleichzeitig zwei verschiedene Kompressions- oder Bandbreitenforderungen zu stellen, so dass eine Erstellung eines eigenständigen Interfaces für jede Betriebsart hier nicht möglich ist. Die damit geschaffenen Interfaces würden prinzipiell gleichzeitig aktiv sein, was zu Fehlfunktionen führt.

Solche exklusiv nutzbaren Interfaces können zu mehreren Alternativkonfigurationen eines einzelnen Interfaces zusammengefasst werden. Zu jedem Zeitpunkt ist dann nur genau eines der Alternativinterfaces aktiv. Dabei ist es jedoch möglich, das aktive Alternativinterface während der Betriebszeit des Gerätes zu wechseln.

Alternativinterfaces können sich nach Belieben in Endpunktzahl und deren Konfiguration unterscheiden. In der Praxis kommen Alternativinterfaces jedoch meist nur zur Bandbreitenselektion bei Isochronous Transfers zum Einsatz.

Tabelle 2.4 zeigt das Format eines Interface Descriptors

bLength Im Falle des Interface Descriptors beträgt die Descriptorlänge neun Byte.

Offset	Feld	Bytes	Wert
0	bLength	1	Nummer
1	bDescriptorType	1	Konstante
2	bInterfaceNumber	1	Nummer
3	bAlternateSetting	1	Nummer
4	bNumEndpoints	1	Nummer
5	bInterfaceClass	1	Klasse
6	bInterfaceSubClass	1	SubKlasse
7	bInterfaceProtocol	1	Protokoll
8	iInterface	1	Index

Tabelle 2.4: Format des Interface Descriptors

bDescriptorType Interface Descriptoren sind durch den Wert 0x04 gekennzeichnet.

bInterfaceNumber Gibt die Nummer des Interfaces an. Laut Spezifikation ist die Interfacenummer nullbasiert zu vergeben, wobei sie für jedes weitere Interface inkrementiert wird. Leider scheinen einige Hersteller in der Praxis von diesem Vorgehen abzuweichen, so dass die Zählung mitunter bei Eins beginnt oder die Nummerierung nicht der Descriptor-Reihenfolge entspricht.

bAlternateSetting Ist dieses Feld ungleich Null, so handelt es sich bei diesem Interface um eine Alternative zu einem vorangehend beschriebenen Interface. Interface Descriptoren mit gleicher Nummer (*bInterfaceNumber*) aber unterschiedlichem *bAlternateSetting* beschreiben jeweils exklusiv nutzbare Alternativinterfaces.

bNumEndpoints Hier ist angegeben, wie viele Endpunkte durch das Interface zusammengruppiert sind. Endpunkt 0 ist immer aktiv, gehört aber zu keinem Interface und wird daher bei dieser Zählung nicht berücksichtigt. Auch lässt *bNumEndpoints* keinen genauen Schluss auf die Zahl der dem Interface zugerechneten Descriptoren zu – Zusätzlich zu den zu erwartenden

Endpunkt Descriptoren können noch beliebige weitere (nicht-Endpunkt) Descriptoren auf das Interface folgen. Der Host kann jedoch das Auftreten eines solchen Descriptors leicht durch die Typangabe im Descriptor-Header feststellen und diesen dann entweder weiter verarbeiten oder aber mit Hilfe der ebenfalls vorhandenen Längenangabe überspringen.

bInterfaceClass An dieser Stelle kommt das durch die USB-Entwickler an ADB angelehnte Klassenkonzept zum Vorschein. Grundgedanke des Klassenkonzeptes ist, dass Geräte, welche dem Host eine vergleichbare Funktionalität zur Verfügung stellen, dies auch auf identische Art und Weise tun. Somit wird die Zahl der verschiedenen benötigten Gerätetreiber reduziert.

Interfaces, welche der selben Klasse angehören, sind über ein identisches *bInterfaceClass*-Feld identifiziert. Die Zuordnung zwischen Klassen und *bInterfaceClass*-Einträgen wird durch das USB-Implementers-Forum (USB-IF) beziehungsweise weiterführende Dokumente vorgenommen.

Ist ein Interface keiner definierten Klasse zuzuordnen, so kann der Hersteller es mittels eines *bInterfaceClass*-Feldes mit Wert 0xFF als "Vendor-Specific" kennzeichnen. Ein geeigneter Treiber muss dann auf andere Art und Weise gefunden werden. Der Wert Null ist zukünftiger Definition vorbehalten.

bInterfaceSubClass Dieses Feld dient einer genaueren Interfacespezifierung innerhalb der durch *bInterfaceClass* angegebenen Klasse. Ein Wert von 0xFF dient auch hier wieder als herstellerspezifische Angabe, andere Werte werden durch das USB-IF definiert.

bInterfaceProtocol Im Falle der Zugehörigkeit des Interfaces zu einer bestimmten Klasse gibt dieses Feld an, welche klassenspezifischen Kommunikationsprotokolle das Interface unterstützt. Ein Wert von Null zeigt an, dass keine klassenspezifischen Protokolle Verwendung finden. Der Wert 0xFF zeigt die Verwendung eines herstellerspezifischen Protokolls an.

iInterface Gibt die Nummer des sogenannten String Descriptors an, welcher das Interface in menschenlesbarer Form beschreibt. Details zu String Descriptoren werden später erläutert.

Configuration Descriptor

Bei USB-Geräten werden jeweils ein oder mehrere Interfaces zu einer sogenannten "Konfiguration" zusammengefasst. Jede Konfiguration beschreibt

eine Betriebsart, welche zu Beginn der Betriebszeit des Gerätes durch den Host gewählt wird und danach für den Rest der Betriebsdauer feststeht.

Als Anwendungsbeispiel beschreibt die USB-Spezifikation ein ISDN-Gerät, welches mittels einer ersten Konfiguration zwei Interfaces bereitstellt. Jedes dieser Interfaces stellt dem Host jeweils einen 64kbps-Kanal zur Datenübertragung zur Verfügung. In einer weiteren Konfiguration besitzt das Gerät nur noch ein Interface, welches geräteintern beide Kanäle belegt und dem Host dann eine 128kbps-Verbindung bietet.

Wie Tabelle 2.5 zeigt, macht ein Gerät dem Host auch die jeweils benötigte Leistungsaufnahme über den Configuration Descriptor bekannt – Je nach Konfiguration können sich die benötigten Hardwaregruppen und damit auch der Strombedarf eines Gerätes verändern.

Häufig werden verschiedene Konfigurationen gerade dazu eingesetzt, die Leistungsaufnahme von mobilen Kleingeräten wie Mobiltelefonen und MP3-Playern zu steuern. Solche Geräte nutzen die Stromversorgung des USB oft nicht nur für den Betrieb am PC, sondern laden während dieser Betriebsdauer auch ihren Geräteakku über die durch den Host zur Verfügung gestellte Stromquelle.

Wird ein solches Gerät an den Bus angeschlossen, so entscheidet der Host, ob über den Bus genügend Leistung zur Verfügung gestellt werden kann, um einen Ladebetrieb zu ermöglichen. Anderenfalls wird dieser durch Wahl einer Konfiguration, in der das Gerät den Ladebetrieb einstellt, unterbunden.

Anzumerken ist jedoch, dass die Energieverwaltung hinfällig ist, sobald ein Gerät Leistung vom Bus aufnimmt, ohne dies dem Host korrekt bekannt zu machen. Dies ist zwar durch die Spezifikation eindeutig untersagt, in der Praxis jedoch trotzdem anzutreffen. Zur Gruppe solcher Geräte zählen zum einen kleinere Lampen oder vegleichbares, die per Definition keine USB-Geräte sind, aber trotzdem Leistung über den Bus aufnehmen. Zum anderen benötigen mitunter aber auch USB-Geräte, wie externe Festplatten, einen höheren Betriebsstrom, als sie dem Host im Configuration Descriptor angeben.

Die Felder des Configuration Descriptors sind wie folgt definiert:

bLength Im Falle des Configuration Descriptors beträgt die Descriptorlänge neun Byte.

bDescriptorType Configuration Descriptoren sind durch den Wert 0x02 gekennzeichnet.

Offset	Feld	Größe	Wert
0	bLength	1	Nummer
1	bDescriptorType	1	Konstante
2	wTotalLength	2	Nummer
4	bNumInterfaces	1	Nummer
5	bConfigurationValue	1	Nummer
6	iConfiguration	1	Index
7	bmAttributes	1	Bitfeld
8	bMaxPower	1	Nummer

Tabelle 2.5: Format des Configuration Descriptors

wTotalLength Dieses Feld gibt die Gesamtlänge der Konfiguration (im vom Host abgefragten Bytestrom) an. Diese umfasst die Summe der Länge aller nachfolgenden Descriptoren (Interfaces, Endpoints usw.), welche der Konfiguration zugehörig sind. Die Größe des Configuration Descriptors selbst ist dabei in dieser Längenangabe ebenfalls enthalten.

bNumInterfaces Gibt die Anzahl der in der Konfiguration aktiven Interfaces an, alternative Interfaces werden dabei also nicht mitgerechnet.

bConfigurationValue Bezeichnet einen Wert, mit dessen Hilfe der Host das Gerät anweisen kann, die durch den Descriptor beschriebene Konfiguration zu nutzen.

iConfiguration Gibt die Nummer des String Descriptors an, welcher die Konfiguration in menschenlesbarer Form beschreibt.

bmAttributes Das hier gespeicherte Bitfeld gibt Auskunft über weitere Eigenschaften der Konfiguration. Bit4 bis Bit0 sind dabei reserviert und auf Null zu halten, Bit7 ist aus historischen Gründen immer auf Eins gesetzt.

Ein Gerät, welches das sogenannte "Remote Wakeup"-Feature unterstützt, setzt hier Bit5. Remote Wakeup beschreibt einen Mechanismus, über welchen

ein Gerät aktiv den Suspended-Zustand verlassen und den Host benachrichtigen kann. Geräte mit eigener Stromversorgung setzen das Bit6. Dies verbietet ihnen jedoch nicht alternativ zu ihrer eigenen Stromversorgung auch den Bus als Stromquelle zu nutzen.

bMaxPower Über dieses Feld geben Geräte an, welche Stromaufnahme sie zum Betrieb in einer bestimmten Konfiguration vom Bus benötigen. Die Angabe erfolgt dabei in Schritten von 2mA, ein Wert von 100 beschreibt also beispielsweise eine Stromaufnahme von 200mA.

Geräte mit externer Stromversorgung können während ihrer Betriebsdauer möglicherweise von dieser getrennt oder an sie angeschlossen werden. Sollte das Gerät auch ohne externe Stromversorgung den Betrieb aufrecht erhalten können, so kann die USB-Systemsoftware den jeweiligen Betriebszustand (Bus- oder Self-Powered) zur Laufzeit vom Gerät erfragen. Das Gerät darf jedoch zu keinem Zeitpunkt mehr Strom vom Bus aufnehmen, als über das *bMaxPower*-Feld spezifiziert ist.

Device Descriptor

Jedes USB-Gerät besitzt genau einen Device Descriptor, in welchem dem Host global für das Gerät gültige Eigenschaften des Gerätes mitgeteilt werden. Hierzu zählen Angaben zum Hersteller, Produktnummer, die unterstützte Version der USB-Spezifikation, aber auch Informationen über die Default Control Pipe des Gerätes.

Tabelle 2.6 zeigt den Aufbau des Device Descriptors.

Die Felder des Device Descriptors sind wie folgt definiert:

bLength Im Falle des Device Descriptors beträgt die Descriptorlänge 18 Byte.

bDescriptorType Device Descriptoren sind durch den Wert 0x01 gekennzeichnet.

bcdUSB Dieses Feld gibt die unterstützte Version der USB-Spezifikation in BCD-Darstellung an. Hierbei wird eine Versionsnummer JJ.M.N[20] in die Darstellung 0xJJMN überführt; eine Version 2.10 ergibt also einen Wert von 0x0210, die Version 2.0 ist entsprechend mit 0x0200 codiert.

Sollte das Gerät eine höhere Versionsnummer als der Host aufweisen, so kann der Host von einer Abwärtskompatibilität des Geräts ausgehen.

[20] JJ=MajorNumber, M=MinorNumber, N=SubMinorNumber

Offset	Feld	Bytes	Wert
0	bLength	1	Nummer
1	bDescriptorType	1	Konstante
2	bcdUSB	2	BCD
4	bDeviceClass	1	Klasse
5	bDeviceSubClass	1	SubKlasse
6	bDeviceProtocol	1	Protokoll
7	bMaxPacketSize0	1	Nummer
8	idVendor	2	ID
10	idProduct	2	ID
12	bcdDevice	2	BCD
14	iManufacturer	1	Index
15	iProduct	1	Index
16	iSerialNumber	1	Index
17	bNumConfigurations	1	Nummer

Tabelle 2.6: Format des Device Descriptors

bDeviceClass Wie schon die Interfaces sind auch Geräte zu Klassen zusammengruppiert. Für das Geräteklassenfeld gelten grundsätzlich identische Angaben wie für *bInterfaceClass*.

Eine Null bedeutet hier jedoch keine Klassenzugehörigkeit des Gerätes – Der Host orientiert sich dann an den Klassen der einzelnen Interfaces.

bDeviceSubClass Die Verwendung des Subklassenfeldes für Geräte verläuft analog zu der Verwendung des Feldes *bInterfaceSubClass* bei Interfaces.

bDeviceProtocol Die Verwendung des Protokollfeldes für Geräte verläuft analog zu der Verwendung des Feldes *bInterfaceProtocol* bei Interfaces.

bMaxPacketSize Das Feld gibt die maximale Paketgröße für die Default Control Pipe an (Paketgröße der Endpunkte 0). Die Angabe korrespondiert mit den Angaben, die auch in den Endpunkt Descriptoren getroffen werden.

idVendor Jeder Hersteller von USB-Geräten kann sich eine eindeutige Identifikationsnummer durch das USB-IF zuweisen lassen. Geräte dieses Herstellers tragen dann die entsprechende Nummer im *idVendor*-Feld.

idProduct Dieses Feld trägt die durch den Hersteller vergebene Produktnummer des Gerätes und kann durch die USB-Systemsoftware zusammen mit der Herstellernummer zur Identifikation des zum Gerätebetrieb notwendigen Treiber verwendet werden.

bcdDevice Hier bringt der Hersteller eine BCD-Codierte Versionsnummer des Gerätes unter. Geräte mit gleicher ID aber unterschiedlichen Treibern können so gehandhabt werden.

iManufacturer Index des String Descriptors (siehe folgender Abschnitt), welcher den Herstellernamen enthält.

iProduct Index des String Descriptors, welcher den Produknamen enthält.

iSerialNumber Index des String Descriptors, welcher die Seriennummer des Gerätes enthält.

bNumConfigurations Dieses Feld gibt die Anzahl der Konfigurationen an, welche das Gerät besitzt. Der Host kann diese einzeln erfragen und auswählen.

String Descriptor

Anders als die vorangehend gezeigten Descriptoren enthalten String Descriptoren Informationen in menschenlesbarer beziehungsweise natürlichsprachlicher Form. Die Verwendung von String Descriptoren ist optional. Unterstützt ein Gerät keine String Descriptoren, so sind alle auf solche verweisenden Referenzen mit Null zu belegen. Werden nur einige der vorangehend genannten String Descriptoren nicht unterstützt, so verbleiben deren Referenzen entsprechend auf Null. Der Host kann die einzelnen String Descriptoren über deren Index abfragen.

Offset	Feld	Größe	Wert
0	bLength	1	Nummer
1	bDescriptorType	1	Konstante
2	wLANGID[0]	2	LANGID
2	.....	2	LANGID
2	wLANGID[N]	2	LANGID

Tabelle 2.7: Format des String Descriptors an Index 0

String Descriptoren enthalten Verschriftlichung natürlicher Sprache in Unicode-Darstellung, wie sie durch das Unicode Consortium festgeschrieben ist. Dabei können die String Descriptoren in verschiedenen Sprachen vorliegen. USB-Geräte, welche mindestens einen String Descriptor bieten, enthalten an Descriptor-Index 0 eine Datenstruktur der in Tabelle 2.7 gezeigten Form. Diese Struktur gibt Auskunft über die unterstützten Sprachen, wobei eine Tabelle zur Zuordnung der Feldwerte von *wLANGID[N]* zur Sprache des String Descriptors über eine bei http://www.usb.org/developers/docs.html beziehbare Tabelle vollzogen werden kann[21].

Die Länge des Descriptors und damit der Eintrag im Feld *bLength* variiert mit der Zahl der unterstützten Sprachen. Der in *bDescriptorType* eingetragene Descriptortyp lautet für String Descriptoren 0x03.

Der Host erhält einen String Descriptor bestimmter Sprache durch Anfrage beim Gerät unter Angabe des Descriptor-Index sowie der gewünschten LANGID. Das Gerät liefert dann einen String Descriptor der in Tabelle 2.8 gezeigten Form zurück.

Der Descriptortyp lautet wieder 0x03, die Länge variiert mit der Länge der in *bString* gespeicherten Zeichenkette.

Interface Associaton Descriptor

Zusätzlich zu den vorangehend beschriebenen Descriptoren bietet eine Erweiterung der USB-Spezifikation ([Spec]) eine Möglichkeit, mehrere Interfaces mit einer einzigen Funktion zu assoziieren. Die ursprüngliche Spezifikation beschreibt eine Eins-zu-Eins-Beziehung zwischen einem Treiber und der dazugehörigen Gerätefunktion, welche durch ein Interface beschrieben wird.

[21] Diese URL ist ebenfalls durch [Spek] spezifiziert.

Offset	Feld	Größe	Wert
0	bLength	1	Nummer
1	bDescriptorType	1	Konstante
2	bString	N	UNICODE String

Tabelle 2.8: Format der String Descriptorn an Indizes ungleich 0

Dieses Vorgehen erschien jedoch einigen Herstellern in der Praxis nicht ausreichend, so dass für eine einzelne Gerätefunktion mitunter ein kombinierte Nutzung mehrerer Interfaces zum Einsatz kam.

In der Erweiterung [Spec] wird daher eine standardisierte Methode beschrieben, um mehrere Interfaces zu einer sogenannten Interface Association zu verbinden. Die Interfaces der Association stellen dann gemeinsam eine Gerätefunktion bereit. Der Descriptor, welcher die Association beschreibt, wird Interface Association Descriptor genannt. Für den Betrieb einer Interface Association kommt auf Hostseite ein einzelner Treiber zum Einsatz.

Angezeigt wird das Vorhandensein eines solchen Descriptors durch verschiedene Einträge im Device Descriptor. Hier sind die Felder *bDeviceClass* mit 0xEF, *bDeviceSubClass* mit 0x02 und *bDeviceProtocol* mit 0x01 zu belegen.

Tabelle 2.9 zeigt das Format eines Interface Association Descriptors.

bLength Im Falle des Interface Association Descriptors beträgt die Descriptorlänge acht Byte.

bDescriptorType Interface Association Descriptoren sind durch den Wert 0x0B gekennzeichnet.

bFirstInterface Dieses Feld spezifiziert die Nummer des ersten Interfaces, welches mit der Funktion assoziiert ist. Die Nummer gibt den Wert des *bInterfaceNumber*-Feldes des entprechenden Interface Descriptors an.

bInterfaceCount Über dieses Feld wird angegeben, wie viele Interfaces zu der Association gehören und somit zusammen eine Funktion bilden. Die zugehörigen Interfaces sind dabei in aufsteigender Reihenfolge durchnummeriert, so daß alle Interfaces zwischen *bFirstInterface* und *bFirstInterface* +

Offset	Feld	Bytes	Wert
0	bLength	1	Nummer
1	bDescriptorType	1	Konstante
2	bFirstInterface	1	Nummer
3	bInterfaceCount	1	Nummer
4	bFunctionClass	1	Klasse
5	bFunctionSubClass	1	SubKlasse
6	bFunctionProtocol	1	Protokoll
7	iFunction	1	Index

Tabelle 2.9: Format des Interface Association Descriptors

bInterfaceCount zu der durch die Association gebildeten Funktion gehören.

bFunctionClass Die Bedeutung dieses Feldes folgt der von *bInterfaceClass* eines Interface Descriptors. Ein Wert von Null ist hier verboten, 0xFF zeigt eine herstellerspezifische Klasse an. Andere Werte werden durch das USB-IF spezifiziert und sind weiterführenden Dokumenten zu entnehmen.

bFunctionSubClass Dieses Feld dient einer genaueren Spezifizierung der Association innerhalb der durch *bFunctionClass* angegebenen Klasse. Ein Wert von 0xFF dient wieder als herstellerspezifische Angabe, andere Werte werden durch das USB-IF definiert.

bFunctionProtocol Im Falle der Zugehörigkeit der Interface Association zu einer bestimmten Klasse gibt dieses Feld an, welche klassenspezifischen Kommunikationsprotokolle die Gerätefunktion unterstützt. Ein Wert von Null zeigt an, dass keine klassenspezifischen Protokolle Verwendung finden. Der Wert 0xFF zeigt die Verwendung eines herstellerspezifischen Protokolls an.

iFunction Gibt die Nummer des String Descriptors an, welcher die Interface Association in menschenlesbarer Form beschreibt.

Offset	Feld	Größe	Wert
0	bmRequestType	1	Bitmap
1	bRequest	1	Nummer
2	wValue	2	Nummer
4	wIndex	2	Index und offset
6	wLength	2	Zähler

Tabelle 2.10: Format eines Requests

2.4.5.2 Device Requests

Wie bereits mehrfach erwähnt, richtet der Host Anfragen und Kommandos an Geräte, um diese zu konfigurieren oder auch die Descriptoren zu erhalten. Hierzu nutzt der Host die Default Control Pipe eines Gerätes und sendet darüber einen sogenannten Request eines festgeschrieben Formats, woraufhin das Gerät mit den geforderten Informationen oder der Durchführung der geforderten Aktion reagiert.

Jeder Request wird dabei durch eine Sequenz von acht Byte repräsentiert, welche in der Setup-Stufe eines Control Transfers zum Gerät übertragen wird. Diese Sequenz bestimmt vollständig den weiteren Ablauf des Control Transfers. Sollten zur Erfüllung des Request auf Geräteseite weitere Daten notwendig sein, so stellt der Host diese in der folgenden Daten-Stufe bereit. Hat der Host Daten vom Gerät angefordert, so überträgt das Gerät diese in der Daten-Stufe zum Host. Der Transfer endet dann, wie durch den Ablauf von Control Transfers gefordert, mit der Status-Stufe. Benötigt ein Geräte weitere Zeit zur Bearbeitung des Requests, so kann es dem Host dies durch Senden von NAKs in der Status-Stufe mitteilen – Diese dauert entsprechend bis zur vollständigen Behandlung des Requests durch das Gerät an.

Tabelle 2.10 zeigt den Aufbau eines jeden Requests.

Die Bedeutung der einzelnen Felder der acht Byte langen Struktur ist im Folgenden erläutert.

bmRequestType Dieses Feld spezifiziert die Charakteristika des Requests. Hierzu gehören Datenrichtung, Typ und Empfänger des Requests. Bezüglich der Datenrichtung wird nur in Up- und Downstream unterschieden, so dass für diese Angabe nur ein einziges Bit benötigt wird.

Angabe	Index	Wert	Beschreibung
Datenrichtung	7	0	Host zu Gerät
		1	Gerät zu Host
Typ	6..5	0	Standard
		1	Class
		2	Vendor
		3	Reserved
Empfänger	4..0	0	Device
		1	Interface
		2	Endpoint
		3	Other
		4..31	Reserved

Tabelle 2.11: Zuordnung der Bits innhalb von *bmRequestType*

Der Requesttyp ist in zwei Bit codiert, wobei die USB-Spezifikation zwischen drei Typen unterscheidet: Die sogenannten Standard-Requests, welche im Weiteren noch detailliert erläutert werden, sind für alle USB-Geräte gemeinsam definiert. Daneben existieren noch klassen- und herstellerspezifische Requests. Die Unterstützung solcher Requests muss jeweils nur durch Geräte, welche der entsprechenden Klasse angehören gewährleistet werden beziehungsweise kann im herstellerspezifischen Fall auch vollkommen frei sein.

Als letzte Information ist der Empfänger zu beachten, wobei hiermit explizit nicht der empfangende Endpunkt (also Endpunkt 0) sondern der Typ der den Request bearbeitenden Instanz angeben wird. Dabei wird unterschieden zwischen dem Gerät selbst, einem seiner Interfaces oder einem seiner Endpunkte. Ist der Request keinem dieser Kontexte zuzuordnen, so kann dies durch Angabe von "Other" gekennzeichnet werden.

Tabelle 2.11 gibt Auskunft über die Zuordnung der einzelnen Bits innerhalb des *bmRequestType*-Bitfeldes.

bRequest Dieses Feld spezifiert den Request selbst in Abhängigkeit der Typangabe in *bmRequestType*. Jedem spezifischen Request ist ein einzigartiger Eintrag zugeordnet, wobei die USB-Spezifikation nur die Zuordnung für die Standard Requests vorschreibt[22]. Weitere Zuordnungen werden entwe-

[22]Siehe auch Tabelle 2.12.

der durch weiterführende klassenspezifische Dokumente oder den Hersteller selbst vorgenommen.

wValue Über dieses Feld werden requestspezifische Zusatzinformationen bereitgestellt. Hierzu gehören – abhängig von der durchzuführenden Aktion – beispielsweise die einem Gerät neu zuzuweisende Adresse oder auch die Nummer einer zu aktivierenden Konfiguration.

Die genaue Bedeutung der im Feld enthaltenen Werte erschließt sich daher nur bei Betrachtung eines spezifischen Requests.

wIndex Ist durch *bmRequestType* das Gerät selbst (oder "Other") als Empfänger adressiert, so variiert die Bedeutung der hier eingetragenen Werte wie auch bei *wValue*.

Ist der Empfänger ein Interface, so geben die unteren acht Bit die Nummer des adressierten Interfaces an. Diese entspricht genau dem in *bInterfaceNumber* des Interface Descriptors vorgefundenen Wert.

Wird ein Endpunkt adressiert, so tragen die unteren vier Bit dessen Nummer. Zusätzlich gibt Bit7 die Datenrichtung des Endpunktes an. Das Feld enthält dann also den Wert des *bEndpointAddress*-Feldes innerhalb des den Endpunkt beschreibenden Descriptors.

wLength Gibt die Anzahl der Bytes an, welche in der Daten-Stufe übertragen werden. Im Falle eines Control Writes entspricht diese Zahl exakt der vom Host zu sendenden Anzahl Bytes, bei einem Control Read darf ein Gerät niemals mehr als die hier geforderte Bytezahl liefern. Hat ein Gerät weniger Daten zu senden als hier angegeben, so wird die Daten-Stufe wie gewohnt durch ein Underlength-Packet beendet.

Sollte keine Daten-Stufe benötigt werden, so ist das Feld mit Null zu belegen – Bei solchen No-Data Control Transfers wird ferner der Wert des Datenrichtungsbits in *bmRequestType* ignoriert.

2.4.5.3 Standard Device Requests

Im Folgenden werden die für alle USB-Geräte geltenden Standard Device Requests betrachtet. Tabelle 2.12 gibt eine vollständige Übersicht der verfügbaren Standard Requests und deren für das *bRequest*-Feld einzutragende Werte. Im Weiteren werden zur besseren Verständlichkeit nur noch die Requestnamen und nicht deren zugehörige *bRequest*-Einträge genutzt. Die hier verwendete Betrachtungsreihenfolge orientiert sich sich nicht an der Tabelle,

bRequest	Value
GET_STATUS	0
CLEAR_FEATURE	1
Reserved for future use	2
SET_FEATURE	3
Reserved for future use	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

Tabelle 2.12: Übersicht der Standard Device Requests und deren zugeordneter *bRequest*-Einträge.

sondern ist so weit wie möglich entsprechend der in der Praxis notwendigen Nutzungsreihenfolge gehalten.

Wird ein falscher oder nicht unterstützter Request empfangen, so interpretiert ein Gerät dies als Fehler und reagiert mit einem STALL in der auf das Setup folgenden Stufe.

Get Descriptor

Über den "Get Descriptor"-Request kann der Host zu einem beliebigen Zeitpunkt einen spezifischen Descriptor vom Gerät abfragen. Das Gerät liefert daraufhin in der Daten-Stufe den erfragten Descriptor zurück, sofern er denn existiert. Tabelle 2.13 zeigt die für diesen Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_DESCRIPTOR	Descriptor Typ und Descriptor Index	0 oder LANGID	Descriptor Length	Descriptor

Tabelle 2.13: Format des Requests "Get Descriptor"

Da der Empfänger des Requests hier das Gerät selbst ist (vergleiche *bmRequestType*, Bit4..0), kommt *wValue* eine requestspezifische Bedeutung zu.

Der Typ des erfragten Descriptors (wie in *bDescriptorType*) ist im höherwertigen Byte angegeben. Das niederwertige Byte trägt den Index des Descriptors.

Handelt es sich bei dem zurückzuliefernden Descriptor um einen String Descriptor, so gibt *wIndex* die geforderte LANGID des Strings an. Bei Abfrage anderer Descriptortypen ist *wIndex* mit Null zu belegen.

Einschränkend ist zu sagen, dass nicht jeder Descriptortyp direkt in *bDescriptorType* einsetzbar ist – nur Device-, Configuration- und String Descriptoren sind direkt mittels eines "Get Descriptor"-Requests abrufbar. Die zu der abgefragten Konfiguration gehörenden Interface- und Endpoint Descriptoren werden vom Gerät automatisch zusammen mit dem Configuration Descriptor geliefert. Gleiches gilt für eventuell vorhandene klassen- oder herstellerspezifische Descriptoren, welche zur Konfiguration gehören.

Zum besseren Verständnis der beschriebenen Descriptoren und damit der durch "Get Descriptor" abgefragten Daten zeigt Abbildung 2.28 alle Descriptoren eines USB-Gerätes (in diesem Fall eines Keyboards). Vor dem Configuration Descriptor der ersten (und einzigen) Konfiguration wurde der global gülte Device Descriptor abgedruckt. Die auf der rechten Seite genutzte gegliederte Descriptoransicht lässt sich mit Hilfe des Linux-Utilities `lsusb` erzeugen. Gut zu sehen ist auch die Einbindung eines klassenspezifischen Descriptors. In diesem Fall handelt es sich um den in [Speb] definierten HID-Descriptor, welcher zur Beschreibung von Ein- und Ausgabegeräten genutzt wird. Die Abkürzung HID steht dabei für "Human Interface Device".

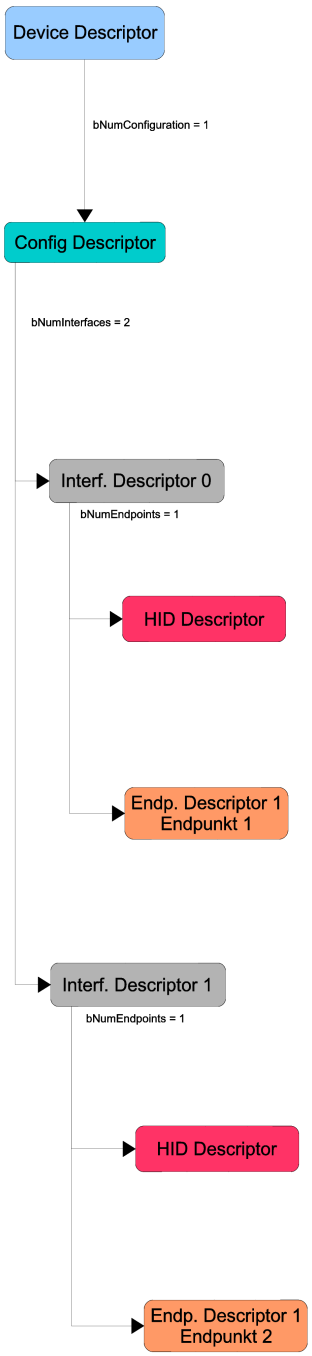
Set Address

Dieser Request dient dem Host zur Vergabe einer Adresse an ein Gerät. Tabelle 2.14 zeigt die für diesen Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_ADDRESS	Geräte Adresse	0	0	–

Tabelle 2.14: Format des Requests "Set Address"

wValue definiert hier die Adresse, welche dem Gerät zugeteilt werden soll. Sind die Werte der Felder *wIndex* und *wLength* ungleich Null oder ist die geforderte Adresse größer als 127, so gibt das Gerät einen Fehler zurück.




```
ID 0a81:0101 Chesen Electronics Corp. Keyboard
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB                1.10
  bDeviceClass            0 (Defined at Interface
  bDeviceSubClass         0                level)
  bDeviceProtocol         0
  bMaxPacketSize0         8
  idVendor           0x0a81 Chesen Electronics C.
  idProduct          0x0101 Keyboard
  bcdDevice             1.10
  iManufacturer           1 CHESEN
  iProduct                2 USB Keyboard
  iSerial                 0
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                9
    bDescriptorType        2
    wTotalLength          59
    bNumInterfaces         2
    bConfigurationValue    1
    iConfiguration         0
    bmAttributes        0xa0
      Remote Wakeup
    MaxPower             100mA
    Interface Descriptor:
      bLength                9
      bDescriptorType        4
      bInterfaceNumber       0
      bAlternateSetting      0
      bNumEndpoints          1
      bInterfaceClass        3 HID
      bInterfaceSubClass     1 Boot Interface
      bInterfaceProtocol     1 Keyboard
      iInterface             0
        HID Device Descriptor:
          bLength                9
          bDescriptorType       33
          bcdHID              1.10
          bCountryCode          33 US
          bNumDescriptors        1
          bDescriptorType       34 Report
          wDescriptorLength     65
         Report Descriptors:
           ** UNAVAILABLE **
      Endpoint Descriptor:
        bLength                7
        bDescriptorType        5
        bEndpointAddress    0x81  EP 1 IN
        bmAttributes           3
          Transfer Type            Interrupt
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0008  1x 8 bytes
        bInterval             10
    Interface Descriptor:
      bLength                9
      bDescriptorType        4
      bInterfaceNumber       1
      bAlternateSetting      0
      bNumEndpoints          1
      bInterfaceClass        3 HID
      bInterfaceSubClass     0 No Subclass
      bInterfaceProtocol     0 None
      iInterface             0
        HID Device Descriptor:
          bLength                9
          bDescriptorType       33
          bcdHID              1.10
          bCountryCode          33 US
          bNumDescriptors        1
          bDescriptorType       34 Report
          wDescriptorLength    102
         Report Descriptors:
           ** UNAVAILABLE **
      Endpoint Descriptor:
        bLength                7
        bDescriptorType        5
        bEndpointAddress    0x82  EP 2 IN
        bmAttributes           3
          Transfer Type            Interrupt
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0008  1x 8 bytes
        bInterval             10
```

Abbildung 2.28: Aufbau der Descriptoren am Beispiel eines USB-Keyboards

Wurde einem Gerät einmal eine Adresse zugewiesen, so ist diese für die restliche Betriebsdauer beziehungsweise bis zum nächsten USB-Reset festgeschrieben und kann nicht durch einen erneutes "Set Address" verändert werden.

Set Configuration

Mit diesem Request aktiviert der Host eine bestimmte Konfiguration des Gerätes. Tabelle 2.15 zeigt die für den Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_CONFIGURATION	Konfigurationswert	0	0	–

Tabelle 2.15: Format des Requests "Set Configuration"

Das niederwertige Byte des *wValue*-Feldes gibt die zu aktivierende Konfiguration (entsprechend des Feldes *bConfigurationValue* innerhalb des zugehörigen Configuration Descriptors) an. Das höherwertige Byte ist immer mit Null zu belegen.

Der Host darf diesen Request nur durchführen, nachdem er dem betroffenen Gerät bereits eine Adresse zugewiesen hat. Sollte nachträglich (also nach einem bereits erfolgten "Set Configuration") eine Änderung der Konfiguration notwendig sein, so muss der Host vor der Anforderung der neuen Konfiguration einen "Set Configuration"-Request durchführen, dessen *wValue*-Feld den Wert Null enthält[23].

Get Configuration

Dieser Request veranlasst das Gerät, dem Host die momentan aktive Konfiguration mitzuteilen. Das vom Gerät in der Daten-Stufe gelieferte Byte entspricht dem *bConfigurationValue* der aktiven Configuration. Ein Rückgabewert von Null gibt dabei an, dass das Gerät momentan keine aktive Konfiguration besitzt.

Tabelle 2.16 zeigt die für diesen Request definierten Feldeinträge.

Set Interface

Dieser Request erlaubt es dem Host, aus mehreren Alternativen für ein Interface zu wählen.

[23]Dies entspricht einem Rückfallen des Gerätes in den Address-Zustand und startet damit praktisch den Betrieb eines Gerätes neu. Die dem Gerät im Vorfeld zugewiesene Adresse bleibt dabei jedoch gültig. Siehe hierzu 2.4.6.1.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_CONFIGURATION	0	0	1	Aktive Configuration

Tabelle 2.16: Format des Requests "Get Configuration"

Sind zu einer Interfacenummer mehrere Interfaces mit unterschiedlichem *bAlternateSetting* vorhanden, so kann der Host mittels "Set Interface" eines dieser alternativen Interfaces aktivieren.

Tabelle 2.17 zeigt die für diesen Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000001B	SET_INTERFACE	Alternate Setting	Interface	0	–

Tabelle 2.17: Format des Requests "Set Interface"

wValue gibt dabei den *bAlternateSetting*-Eintrag des zu aktivierenden Alternativinterfaces an. Der Request kann nur dann behandelt werden, wenn das betroffene Gerät sich bereits in einem konfigurierten Zustand befindet.

Get Interface

Der "Get Interface"-Request veranlasst das Gerät, dem Host das momentan gewählte *bAlternateSetting* eines Interfaces mitzuteilen. Die Rückgabe des geforderten Wertes erfolgt entsprechend in der Daten-Stufe.

Tabelle 2.18 zeigt die für diesen Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000001B	GET_INTERFACE	0	Interface	1	Alternate Setting

Tabelle 2.18: Format des Requests "Get Interface"

Existiert das angegebene Interface nicht, so reagiert das Gerät mit einem Fehler. Auch dieser Request kann nur dann durchgeführt werden, wenn das Gerät bereits konfiguriert wurde.

Set Feature

Über "Set Feature" werden geräte-, interface- oder endpunktspezifische Zusatzattribute aktiviert.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_FEATURE	Feature	0	0	–
00000001B		Selector	Interface		
00000010B			Endpunkt		

Tabelle 2.19: Format des Requests "Set Feature"

Tabelle 2.19 zeigt die für "Set Feature" definierten Feldeinträge.

Für Full- und Lowspeed-Geräte ist die Wahlmöglichkeit für den Feature Selector auf REMOTE_WAKEUP (0x01) mit dem fraglichen Gerät als Empfänger sowie auf das HALT-Feature (0x00) mit adressiertem Endpunkt beschränkt. Hierzu ist zu sagen, dass das HALT-Feature jedoch vor allem für den nachfolgend beschriebenen "Clear Feature"-Request von Bedeutung ist, da so ein aufgrund eines Fehlers angehaltener Endpunkt wieder zur Funktion gebracht werden kann.

Clear Feature

Analog zum Setzen eines Features mit "Set Feature" kann dieses auch mittels "Clear Feature" gelöscht werden.

Tabelle 2.20 zeigt die für diesen Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	CLEAR_FEATURE	Feature	0	0	–
00000001B		Selector	Interface		
00000010B			Endpunkt		

Tabelle 2.20: Format des Requests "Clear Feature"

Unterstützt das Gerät das Löschen des angegebenen Features nicht, so reagiert es mit einem STALL.

Get Status

Mittels "Get Status" kann der Host zur Betriebszeit eines Gerätes dessen aktuellen Status in Erfahrung bringen. Im gezeigten Fall eines Standard Requests teilt das Gerät dem Host darauf in der Daten-Stufe mit, ob das REMOTE_WAKEUP-Feature aktiviert ist und ob es seine Stromzufuhr gerade über den Bus oder eine externe Stromversorgung erhält.

Auch Interfaces und Endpunkte können Empfänger dieses Requests sein und liefern entsprechend ihren Status.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_STATUS	0	0	2	Device
10000001B			Interface		Interface oder
10000010B			Endpunkt		Endpunkt
					Status

Tabelle 2.21: Format des Requests "Get Status"

Tabelle 2.21 zeigt die für diesen Request definierten Feldeinträge.

Der Gerätestatus ist immer in zwei Bytes codiert, wobei jedoch nur den niedrigsten beiden Bits tatsächlich Bedeutung zukommt. Ein gesetztes Bit0 gibt an, dass sich das Gerät im Selfpowered-Modus befindet. Anderenfalls bezieht es seine Energie vom Bus. Bit1 gibt ein durch den Host aktiviertes REMOTE_WAKEUP-Feature an.

Alle anderen Bits gelten als reserviert und sind vom Gerät mit Null zu belegen.

Innerhalb der Antwort eines Interfaces auf den "Get Status"-Request sind alle Bytes reserviert und mit Null belegt.

Für Endpunkte gibt ein gesetztes Bit0 an, dass sich der Endpunkt im angehaltenen Zustand befindet (HALT-Feature ist gesetzt). Auch hier sind die restlichen Bits reserviert und mit Null belegt.

Synch Frame

Bei Kommunikation mit Isochronous-Endpunkten kann es notwendig sein, dass die verwendete Paketfolge einem bestimmten (immer wiederkehrenden) Muster bezüglich Paketgröße und enthaltener Daten folgt. Solche Muster ergeben sich aus der Anwendung beziehungsweise dem Design eines Gerätes und werden daher nicht näher durch die USB-Spezifikation erläutert. Die Spezifikation stellt jedoch mit dem "Synch Frame"-Request ein für alle Geräte einheitliches Mittel zur Verfügung, um bei Notwendigkeit solcher Muster die Synchronität zwischen Host und Gerät zu gewährleisten.

Der Host kann dazu mittels des "Synch Frame"-Requests die Framenummer erfragen, welche als Startpunkt für das aktuell laufende Muster diente. Das Gerät überträgt die Framenummer codiert in zwei Byte während der Daten-Stufe.

Der Gerätetreiber kann somit die notwendige Paketfolge einhalten.

Tabelle 2.22 zeigt die für diesen Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000010B	SYNCH_FRAME	0	Endpunkt	2	Frame-Nummer

Tabelle 2.22: Format des Requests "Synch Frame"

Set Descriptor

Optional kann ein USB-Gerät die Modifikation seiner Descriptoren zur Betriebszeit durch den Host unterstützen. Hierzu nutzt der Host den "Set Descriptor"-Request.

Tabelle 2.23 zeigt die für diesen Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_DESCRIPTOR	Descriptor Typ und Descriptor Index	LANGID	Descriptor Length	Descriptor

Tabelle 2.23: Format des Requests "Set Descriptor"

Die Feldwerte besitzen hier jeweils die selbe Bedeutung, die ihnen auch bei "Get Descriptor" zukommen. Dieser Request wird nur selten unterstützt und findet daher keine breite Anwendung.

Übersicht über die Standard Device Requests

Abschließend zeigt Tabelle 2.24 noch einmal eine vollständige Übersicht aller Requests, wie sie auch in [Spek] zu finden ist.

2.4.6 Gerätezustände

Ein USB-Gerät kann sich in verschiedenen Zuständen befinden, wobei eine gewisse Teilmenge dieser Zustände unmittelbar durch den Host beziehungsweise die USB-Systemsoftware erkennbar ist.

Die USB-Spezifikation gibt sowohl diese Zustände selbst, als auch die zwischen ihnen möglichen Zustandsübergänge vor – Die Zustandsübergänge werden hierbei nicht selten überhaupt erst durch einen entsprechenden Request der USB-Systemsoftware ausgelöst.

Abbildung 2.29 zeigt den spezifikationsgemäßen Zustandsgraphen eines USB-Gerätes. Die einzelnen Zustände selbst werden nachfolgend erläutert.

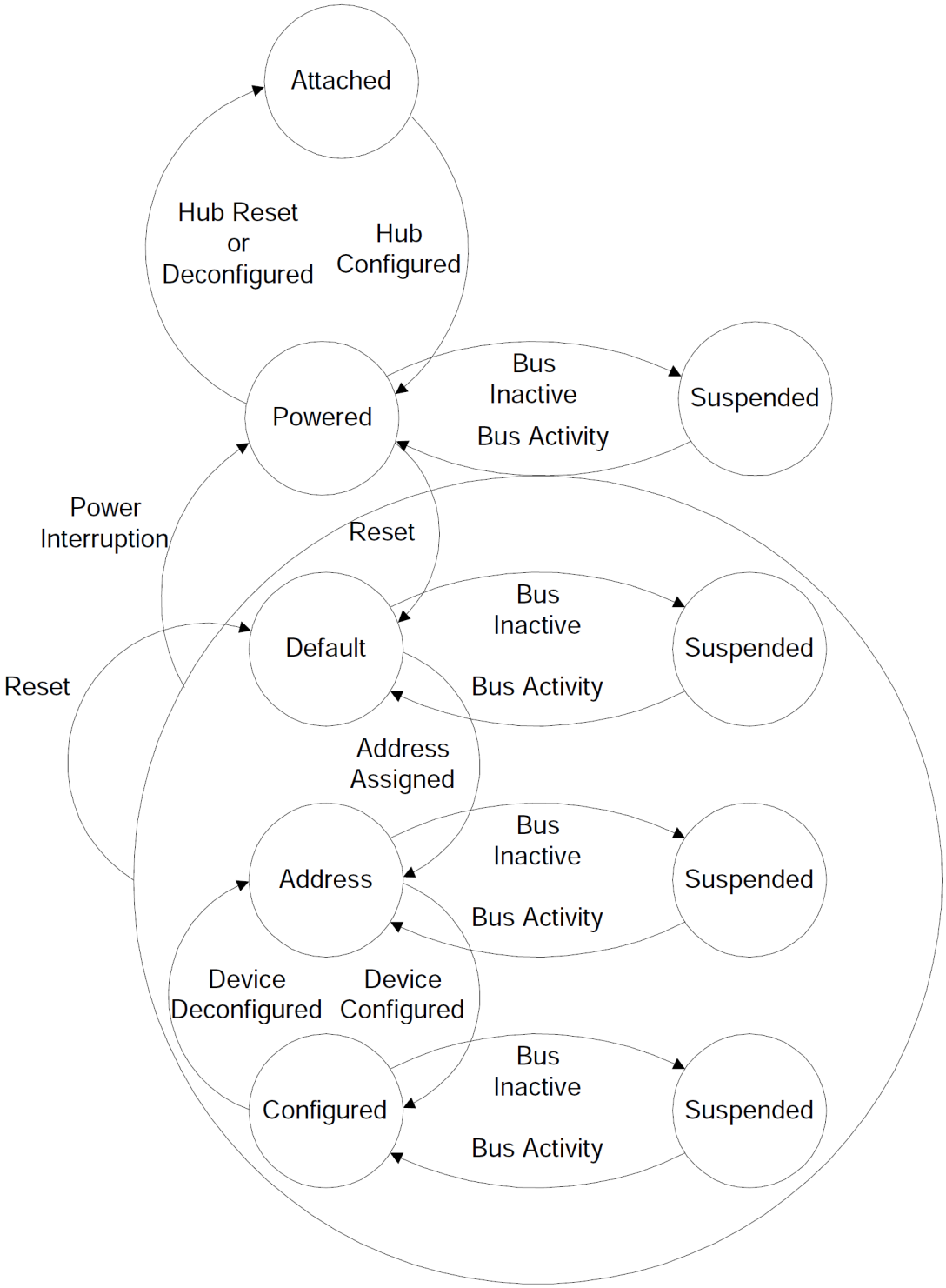


Abbildung 2.29: Zustandsmaschine für USB-Geräte nach [Spek]

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector Endpunkt	0 Interface	0	–
10000000B	GET_CONFIGURATION	0	0	1	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Typ und Descriptor Index	0 oder LANGID	Descriptor Length	Descriptor
10000000B	GET_INTERFACE	0	Interface	1	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	0	0 Interface Endpunkt	2	Device, Interface, oder Endpunkt Status
00000000B	SET_ADDRESS	Device Address	0	0	–
00000000B	SET_CONFIGURATION	Configuration Value	0	0	–
00000000B	SET_DESCRIPTOR	Descriptor Typ und Descriptor Index	LANGID	Descriptor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	0 Interface Endpunkt	0	–
00000001B	SET_INTERFACE	Alternate Setting	Interface	0	–
10000001B	SYNCH_FRAME	0	Endpunkt	2	Frame Nummer

Tabelle 2.24: Standard Device Requests nach [Spek]

Attached Ein Gerät, welches physisch mit dem Bus verbunden wird, begibt sich automatisch in diesen Zustand. Zustände, welche ein Gerät zu Zeitpunkten einnimmt, zu denen es nicht mit dem Bus verbunden ist, sind offensichtlich nicht durch die USB-Spezifikation abzudecken.

Powered Ein Gerät, dessen VBus-Leitung mit der Betriebsspannung des VBusses und dessen GND-Leitung mit Masse verbunden ist, gerät in den Powered-Zustand. Defacto begeben sich alle Geräte sofort in diesen Zustand, sobald sie mit einem korrekt konfigurierten Hub verbunden sind. Ein Host kann die Versorgungsspannung der VBus-Leitung deaktivieren, womit das Gerät zwar augenscheinlich noch (physisch) mit dem Bus verbunden ist, jedoch auf diesen keinerlei Einfluss mehr ausübt. In einem solchen Fall fällt

das Gerät in den Attached-Zustand zurück.

Insbesondere ist es für den Powered-Zustand irrelevant, ob das Gerät bus- oder selfpowered betrieben wird – Der Zustand wird allein durch die Spannungsversorgung der VBus-Leitung definiert.

Default Ein Gerät kann aus allen Zuständen mit Ausnahme von Attached durch einen USB-Reset in den Default-Zustand gebracht werden. In diesem Zustand ist das Gerät bereit, um Kommunikation über den Bus durchzuführen und reagiert auf mit der Standard-Geräteadresse 0 adressierte Pakete. Die USB-Systemsoftware achtet darauf, dass sich zu jedem Zeitpunkt immer nur genau ein Gerät in diesem Zustand befindet[24], so dass durch Adresse 0 niemals mehrere Geräte gleichzeitig angesprochen werden.

Kein Gerät, welches sich im Default-Zustand befindet, benötigt einen Strom von mehr als 100mA vom Bus. Kommunikation ist in diesem Zustand ausschließlich über die Default Control Pipe möglich.

Address Mit der Zuweisung einer Adresse an das Gerät mittels eines "Set Address"-Requests begibt sich dieses in den Address-Zustand.

Das Gerät reagiert nun nur noch auf Pakete, welche an die durch den Host zugewiesene Adresse gerichtet sind. Ein Selfpowered-Gerät, welches in einem der im Weiteren beschriebenen Zustände seine externe Stromversorgung verliert und zur Aufrechterhaltung des Betriebs in seiner aktuellen Konfiguration mehr als 100mA vom Bus benötigt, muss sich ebenfalls in den Address-Zustand begeben. Dabei muss es seine Stromaufnahme auf maximal 100mA einschränken.

Ferner kann der Host ein Gerät zum Übergang in diesen Zustand zwingen, indem er einen "Set Configuration"-Request mit Konfigurationswert Null an das Gerät sendet.

Configured Ein Übergang in den Configured-Zustand ist ausschließlich aus dem Address-Zustand möglich. Der Host aktiviert dazu eine durch das Gerät unterstützte Konfiguration mittels "Set Configuration".

Geräten in diesem Zustand ist es erlaubt, ihre Stromaufnahme vom Bus auf den im *bMaxPower*-Feld des Descriptors der aktuellen Konfiguration anzuheben. Zusätzlich sind in der Folge alle durch die Konfiguration abgedeckten Interfaces beziehungsweise deren Endpunkte initialisiert und kommuni-

[24] Die USB-Systemsoftware weist Hubs zur Signalisierung eines USB-Resets über einen einzelnen, spezifischen Port an. Ein Reset erfolgt damit niemals global, sondern immer spezifisch für ein einzelnes Gerät.

kationsbereit. Dabei werden insbesondere alle Data-Toggles der betroffenen Endpunkte mit DATA0 vorbelegt.

Suspend Geräte, welche wie in Abschnitt 2.3.4 beschrieben einen für mindestens 3ms inaktiven Bus vorfinden, begeben sich in den Suspended-Zustand. Der Suspended-Zustand ist von allen Zuständen mit Ausnahme von Attached aus zugänglich.

Geräte, welche sich in den Suspended-Zustand begeben, behalten eine ihnen eventuell bereits zugewiesene Adresse und Konfiguration bei und kehren bei erneuter Aktivität auf dem Bus in den zuvor herrschenden Zustand zurück.

2.4.6.1 Bus Enumeration

Auf Basis der beschriebenen Descriptoren, Requests und Gerätezustände kann der Host die sogenannte Bus Enumeration durchführen, welche jedes USB-Gerät zur Inbetriebnahme durchlaufen muss.

Die Enumeration beschreibt alle durch den Host notwendigen Maßnahmen, um ein Gerät nach dessen Anschluss an den Bus zu identifizieren, initialisieren und konfigurieren. Das Gerät durchläuft dabei eine Zustandsfolge, bis es sich im Configured-Zustand befindet. Bezüglich des USB ist das Gerät damit voll betriebs- und kommunikationsbereit und kann dann im Zusammenspiel mit einem übergeordneten Treiber seinen eigentlichen Verwendungszweck erfüllen.

Die Beschreibung der Bus Enumeration ist abhängig vom Vorhandensein und der Funktionsweise eines Hubs, wobei zusätzliche Hubs selbst den Prozess der Enumeration durchlaufen müssen, um ihren Betrieb aufnehmen zu können. Die Inbetriebnahme des Root-Hubs (als erster Hub im System) wird dabei durch die Treibersoftware des Betriebssystems vorgenommen. Hubs und deren Arbeitsweise werden ab Abschnitt 2.4.7 genauer betrachtet.

Die bei der Bus Enumeration durchlaufenen Aktionen lauten wie folgt:

1. Der Host wird durch einen Hub darüber informiert, dass eine Zustandsänderung an einem der Ports des Hubs stattgefunden hat. Das an diesen Port neu angeschlossene Gerät befindet sich dabei bereits im Zustand “Powered”.

2. Der Host erfragt die genaue Natur der Statusänderung vom Hub und erlangt somit Kenntnis vom Anschluss des Gerätes.

3. Der Host wartet nun mindestens 100ms, bis sich die Stromversorgung des Gerätes stabilisiert hat. Dann erlaubt die USB-Systemsoftware dem

Hub mit Hilfe eines hubspezifischen Requests, Kommunikation über den fraglichen Port weiterzuleiten. Anschließend weist sie den Hub an, ein Resetsignal auf dem Port zu erzeugen.

4. Nach Ablauf des USB-Resets befindet sich das Gerät im Default-Zustand.

5. Der Host nutzt die Default Control Pipe zur Abfrage des Device Descriptors. Dieser enthält die maximale Payload für alle weiteren Transfers über die Default Control Pipe.

6. Der Host teilt dem Gerät nun eine eindeutige Adresse zu. Das Gerät wechselt seinen Zustand daraufhin zu "Address".

7. Die USB-Systemsoftware nutzt "Get Descriptor" zur Abfrage der Configuration Descriptoren.

8. Basierend auf den Konfigurationsinformationen weist die USB-Systemsoftware dem Gerät mittels "Set Configuration" eine Konfiguration zu. Das Gerät befindet sich daraufhin im Configured-Zustand und ist betriebsbereit.

2.4.7 USB-Hubs

2.4.7.1 Übersicht

Wie bereits angedeutet, handelt es sich bei Hubs um spezielle USB-Geräte, welche weitere Geräte mit dem Bus verbinden können. Physisch entsteht durch die Verwendung von Hubs eine kombinierte Stern-Strang-Struktur. Durch die Eigenschaften des USB sind auf logischer Ebene jedoch alle Geräte direkt mit dem Host verbunden. Elektrisch gesehen besteht jeweils nur eine Punkt-zu-Punkt-Verbindung zwischen zwei am USB angeschlossenen Gerätschaften (Host, Hubs sowie Endgeräte). Dies hat insbesondere den Vorteil, dass Hubs einen Einfluss von Störungen in einem Ast des Busses auf weitere Äste unterbinden können.

Die folgende Liste zeigt einige der wichtigsten Fähigkeiten von Hubs nach Version 2 der USB-Spezifikation auf:

- Verbindung der Geräte miteinander
- Stromverteilung
- Verbindungs- und Trennungserkennung

- Fehlererkennung und Behandlung
- Unterstützung für High-, Full- und Lowspeed-Geräte

Für die weitere Entwicklung von Experimentierplatine und Software sind weder die Highspeed-Fähigkeiten noch der genaue interne Aufbau von Hubs von Interesse. Vielmehr ist es notwendig, die softwareseitig zu erfassende und zu manipulierende Arbeitsweise dieser Geräte darzustellen, damit die Bibliothek im Falle der Möglichkeit eines Mehrgerätebetriebs entsprechend konstruiert und implementiert werden kann.

Hubs bilden bezüglich des USB eine eigene Geräteklasse. Jedes dieser Klasse angehörende Gerät besitzt genau einen Upstream-, jedoch mehrere Downstream-Ports. Signale können nach den Vorgaben des Hosts zwischen dem Upstream- und beliebigen Downstream-Ports weitergeleitet werden. Eine Verbindungsherstellung zwischen zwei Downstream-Ports ist jedoch unzulässig.

Anders als beispielsweise bei Ethernet sind USB-Hubs durch ihre Arbeitsweise nicht protokolltransparent, sondern stellen Geräte mit spezialisierter Aufgabe dar, deren Funktion nur in striktem Zusammenspiel mit dem Host gewährleistet ist.

Hubs bemerken eigenständig Zustandsänderungen an ihren Downstream-Ports und halten Informationen darüber für eine Abfrage durch den Host bereit. Jeder Hub stellt zu diesem Zweck neben der Default Control Pipe genau ein Interface mit einem zugehörigen Interrupt-Endpunkt zur Verfügung.

An dieser Stelle sei nochmals darauf hingewiesen, das der USB aufgrund seiner Funktionsweise keinen echten Interruptbetrieb bietet. Das Vorhandensein neuer Daten auf Geräteseite muss immer durch explizite Anfrage des Hosts beim Gerät (Polling) festgestellt werden.

Der Host fragt das Vorliegen neuer Informationen regelmäßig über den Interrupt-Endpunkt ab und erlangt so Kenntnis von Zustandsänderungen wie dem Anschließen und Abziehen von Geräten. Die durch die Interrupt-Pipe an den Host gelieferten Daten stellen dabei eine Bitmaske dar, in welcher ein gesetztes Bit jeweils eine Zustandsänderung am entsprechenden Port des Hubs signalisiert. Die Zählung der Ports beginnt dabei bei Eins. Bit0 der Änderungsbitmaske ist der Anzeige einer Änderung des Hubzustandes selbst vorbehalten.

Die genaue Art der Zustandsänderung muss über die Default Control Pipe mittels hubklassenspezifischer Requests festgestellt werden. Über solche klassenspezifische Requests kann ferner auch der Zustand des Hubs und dessen Ports manipuliert werden.

Diese später in Software zu implementierenden Requests sowie klassenspezifischen Descriptoren und Feldeinträge werden im Folgenden beschrieben.

Offset	Feld	Bytes	Wert
0	bDescLength	1	Nummer
1	bDescriptorType	1	Konstante
2	bNbrPorts	1	Nummer
3	wHubCharacteristics	2	Bitfeld
5	bPwrOn2PwrGood	1	Nummer
6	bHubContrCurrent	1	Nummer
7	DeviceRemovable	Variabel	Bitfeld
Variabel	PortPwrCtrlMask	Variabel	Bitfeld

Tabelle 2.25: Format des Hub-Klassendescriptors

2.4.7.2 Descriptoren und klassenspezifische Einträge

Für den Full- und Lowspeed-Betrieb sind nur die vorangehend beschriebenen Standard Descriptoren von Bedeutung. Die Klasse der Hubs in Device- und Interface Descriptoren wird dabei durch einen Klassencode von 0x09 identifiziert. Etwaige Angaben zu Subklasse und Protokoll sind mit 0 zu füllen. Die restlichen Angaben repräsentieren das oben beschriebene Vorhandensein von einem Interface mit einem zugehörigen Interrupt-Endpunkt.

Geräte der Hubklasse besitzen zusätzlich einen klassenspezifischen Descriptor, der mittels eines speziellen Requests abgefragt werden kann. Hierin sind Informationen über die Anzahl der vorhandenen Ports, Möglichkeiten zur Stromversorgung der einzelnen Ports sowie weitere Charakteristika des Hubs enthalten.

Tabelle 2.25 stellt den Aufbau des Hubdescriptors dar.

Die Bedeutung der einzelnen Felder ist nachstehend beschrieben.

bDescLength Gibt wie gewohnt die Länge des Descriptors an. Diese ist wegen der letzten beiden Felder variabel, wodurch dem Längenfeld hier besondere Bedeutung zukommt.

bDescriptorType Die Typangabe für Hub-Klassendescriptoren ist durch das USB-Implementers-Forum auf 0x29 festgeschrieben.

bNbrPorts Hier ist die Anzahl der vorhandenen Downstream-Ports angegeben.

wHubCharacteristics Die niederwertigsten beiden Bits geben hier die Möglichkeiten eines Hubs an, Ports von der Stromversorgung des USB zu trennen. Ist Bit0 mit einer Eins beschrieben, so kann der Hub die Spannungsversorgung für jeden Port einzeln kontrollieren. Eine Null gibt entsprechend an, dass immer alle Ports zusammen einen Status einnehmen. Bit1 wird in Hubs gesetzt, welche keine schaltbare Stromversorgung besitzen – Dies ist nur nach Version 1.0 der USB-Spezifikation möglich.

Ist Bit2 in *wHubCharacteristics* gesetzt, so zeigt dies an, dass der Hub Teil eines Compound-Devices ist. Er wurde also zusammen mit anderen Geräten in einem Gehäuse verbaut.

Bit4..3 beschreiben die Sicherung des Hubs vor zu hoher Leistungsaufnahme durch an ihn angeschlossene Endgeräte (Over-Current Protection). Ist Bit4 gesetzt, so besitzt der Hub keine solche Sicherung. Ist eine Sicherung vorhanden, so kann diese portbasiert (Bit3 gesetzt) oder aber lediglich global für alle an den Hub angeschlossenen Geräte arbeiten (Bit3 nicht gesetzt).

Ein gesetztes Bit7 zeigt an, dass der Hub Indikatoren für den Status der einzelnen Downstreamports (LEDs) besitzt und diese durch den Host steuerbar sind.

Die restlichen Bits sind bei Arbeit mit Geräten im Full- und Lowspeed-Modus zu ignorieren.

bPwrOn2PwrGood Beschreibt die Wartezeit in Intervallen von 2ms, welche ein Port nach Einschalten seiner Spannungsversorgung benötigt, um eine stabile Versorgungsspannung bereitzustellen.

bHubContrCurrent Gibt den maximalen Betriebsstrom der Hub-Controller-Elektronik in mA an.

DeviceRemovable Dieses Feld entspricht einer Bitmaske, welche die Ports des Hubs repräsentiert. Ein gesetztes Bit zeigt an, dass sich an dem jeweiligen Port ein fest mit dem Hub verkabeltes Gerät befindet. Das Gerät ist also untrennbar mit dem Port verbunden.

Die Zählung der Ports beginnt dabei bei Eins. Bit0 des *DeviceRemovable*-Feldes ist dementsprechend ungenutzt. Ergibt sich – bedingt durch die Anzahl der vorhandenen Ports – eine Bitzahl, bei welcher das Feld nicht genau auf einer Bytegrenze endet, so wird das Feld bis zur nächsten Bytegrenze mit ignorierbaren Bits aufgefüllt.

PortPwrCtrlMask Dieses Feld existiert nur noch aus Gründen der Abwärtskompatibilität zu USB-1.0. Es enthält für jeden Port des Hubs ein Bit, wobei jedes Bit fest mit Eins zu belegen ist. Auch hier ist das Feld bei Bedarf wieder auf die nächste Bytegrenze aufzufüllen.

2.4.7.3 Hubklassenspezifische Requests

Alle der hier beschriebenen Requests dürfen erst durchgeführt werden, wenn der Host die Bus Enumeration für den Hub vollständig durchgeführt hat. Anderenfalls ist die Gerätereaktion undefiniert.

Wie auffallen wird, orientieren sich die klassenspezifischen Requests der Hubs stark an den Standard Device Requests. Oftmals ändert sich, was den Aufbau des Requests selbst angeht, lediglich das Feld *bmRequestType*, da hier der klassenspezifische Request angezeigt werden muss.

Get Hubdescriptor

Über diesen Request kann der Host den Hubdescriptor erfragen. Wie bei den Standard Descriptoren auch, liefert der Hub diesen in der Daten-Stufe zurück.

Tabelle 2.26 zeigt die für den Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100000B	GET_DESCRIPTOR	Descriptor Typ und Descriptor Index	0	Descriptor Länge	Descriptor

Tabelle 2.26: Format des Requests "Get Hub Descriptor"

Get Hub Status

Auf diesen Request hin liefert ein Hub seinen Status an den Host zurück. Dies geschieht über eine vier Byte lange Datenstruktur, deren Aufbau in Tabelle 2.27 gezeigt ist.

Über das niederwertigste Bit in *wHubStatus* wird angegeben, ob der Hub momentan mit einer externen Spannungsquelle verbunden ist. Eine Null zeigt eine funktionierende externe Spannungsquelle an, bei einer Eins ist diese entsprechend inaktiv.

Ein gesetztes Bit1 zeigt an, dass der Hub überlastet wurde. Die an ihn angeschlossenen Endgeräte versuchen also, mehr elektrische Leistung vom Hub aufzunehmen, als dieser liefern kann. Dieses Bit bezieht sich gemeinsam auf

Offset	Feld	Bytes	Wert
0	wHubStatus	2	Bitfeld
2	wHubChange	2	Bitfeld

Tabelle 2.27: Format der Antwort auf "Get Hub Status"

alle vorhandenen Ports – Sollte der Hub den Zustand zu hoher Leistungsaufnahme auch portbasiert zurückgeben können oder aber keine diesbezügliche Sicherung besitzen, so hat Bit1 immer den Wert Null.

Die restlichen Bits innerhalb *wHubStatus* gelten als reserviert und sind immer Null.

Das Bitfeld *wHubChange* dient als einfache Bitmaske, welche anzeigt, welche Bits in *wHubStatus* einer Änderung unterlegen waren. Der Host quittiert diese Anzeige, indem er die Bits mittels eines "Clear Hub Feature"-Requests zurücksetzt.

Tabelle 2.28 zeigt die für "Get Hub Status" definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100000B	GET_STATUS	0	0	4	Hub Status und Änderungs Anzeige

Tabelle 2.28: Format des Requests "Get Hub Status"

Get Port Status

Analog zu "Get Hub Status" liefert dieser Request den Status eines speziellen Ports.

Tabelle 2.29 zeigt die für den Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100011B	GET_STATUS	0	Port	4	Port Status und Änderungs Anzeige

Tabelle 2.29: Format des Requests "GetPortStatus"

Das erste Word (*wPortStatus*) der insgesamt aus vier Byte bestehenden Hub-Antwort enthält den Status des Ports. Das zweite Word (*wPortChange*)

dient auch hier als Änderungsbitmaske. Dabei werden nur die unteren fünf Bit der Maske ausgewertet, die restlichen lauten immer Null.

Die Bedeutung der einzelnen Bits innerhalb *wPortStatus* lautet wie folgt:

Bit0 (Current Connect Status) Ist gesetzt, wenn gerade ein Gerät mit dem fraglichen Port verbunden ist.

Bit1 (Port Enable/Disable) Ist gesetzt, wenn der Port aktiv ist. Ein Port kann entweder durch einen Fehler oder durch explizite Anweisung der USB-Systemsoftware deaktiviert werden. Das Setzen dieses Bits kann ausschließlich durch einen erfolgreichen USB-Reset über diesen Port verursacht werden.

Bit2 (Suspend) Ist dieses Bit gesetzt, so befindet sich der Port in einem Suspend-Status, in welchem keinerlei Datenverkehr in Downstreamrichtung weitergeleitet wird. Die USB-Systemsoftware kann dieses Bit nach Belieben setzen und Rücksetzen[25].

Bit3 (Over-current) Unterstützt der Hub eine portbasierte Angabe bei Aufnahme von zu hoher Leistung durch das angeschlossene Gerät, so gibt ein gesetztes Bit hier den Zustand zu hoher Leistungsaufnahme an an.

Bit4 (Reset) Setzt die USB-Systemsoftware dieses Bit, so führt der Hub ein USB-Reset für den Port aus. Dieses Bit wird automatisch nach Ablauf des Resets zurückgesetzt.

Bit5..7 (Reserved) Diese Bits gelten als reserviert und sind immer mit Null belegt.

Bit8 (Port Power) Unterstützt der Hub das portbasierte Aktivieren und Deaktivieren der Stromversorgung, so kann die USB-Systemsoftware durch Setzen dieses Bits die Stromversorgung für den Port einschalten. Dieses Bit repräsentiert immer den aktuellen Zustand der Port-Stromversorgung, kann also im Fehlerfall auch durch den Hub zurückgesetzt werden.

[25]Ein Rücksetzen kann gegebenenfalls auch durch ein Gerät, welches das Remote-Wakeup-Feature unterstützt, ausgelöst werden.

Bit9..10 (Low/Highspeed-Device attached) Diese beiden Bits geben die Geschwindigkeit des Gerätes an, falls denn ein solches mit dem Port verbunden ist. Ein gesetztes Bit9 zeigt ein Lowspeed-Gerät an; ein gesetztes Bit10 ein Highspeed-Gerät. Sind beide Bits Null, so beherrscht das Gerät den Fullspeed-Modus.

Bit11 (Port Test Mode) Über ein Setzen dieses Bits kann der Port in den sogenannten Testmodus gebracht werden. Dies ist für den normalen Betrieb nicht von Bedeutung.

Bit12 (Port Indicator Control) Durch ein Setzen dieses Bits kann die USB-Systemsoftware die Kontrolle über eventuell vorhandene Portstatus-Indikatoren (LEDs) des Hubs übernehmen.

Bit13..15 (Reserved) Diese Bits gelten ebenfalls als reserviert und sind immer Null.

Set Hub Feature

Mit Hilfe des "Set Hub Feature"-Requests kann ein bestimmtes Feature des Hubs aktiviert werden. Dieser Request ist zwar definiert, erfüllt jedoch keine Aufgabe, da die wählbaren Hubfeatures sich ausschließlich auf die Änderungen im Hubstatus anzeigenden Bits innerhalb der Rückgabe von "Get Hub Status" beziehen. Diese sind jedoch lediglich mit einem "Clear Hub Feature" zurückzusetzen.

Tabelle 2.28 zeigt dennoch die für "Set Hub Feature" definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100000B	SET_FEATURE	Feature Selector	Null Interface	Null	Keine

Tabelle 2.30: Format des Requests "Set Hub Feature"

Clear Hub Feature

Dieser Request setzt, wie bereits erwähnt, ein Hubfeature – also die Änderungsbits des Hubstatus – zurück. Damit wird dieser Request durch die USB-Systemsoftware dazu genutzt, die Wahrnehmung einer Änderung im Hubstatus zu quittieren.

Tabelle 2.28 zeigt die für "Clear Hub Feature" definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100000B	CLEAR_FEATURE	Feature Selector	0	0	–

Tabelle 2.31: Format des Requests "Clear Hub Feature"

Die validen Werte für *wValue* sind in diesem Falle 0x00 und 0x01. 0x00 dient zum Quittieren einer Änderung des Zustandes der externen Stromversorgung. 0x01 quittiert eine Überlastung des Hubs.

Set Port Feature

Dieser Request aktiviert ein spezielles Feature eines Ports. Alle Portfeatures finden sich hierbei auch in der bereits beschriebenen Statusrückgabe wieder.

Tabelle 2.32 zeigt die für diesen Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100011B	SET_FEATURE	Feature Selector	Port / Selector	0	–

Tabelle 2.32: Format des Requests "Set Port Feature"

Als Eintrag des Feldes *wValue* wird wieder ein "Feature Selector" verwendet, der das zu aktivierende Feature angibt. Das Aktivieren eines Features führt letztlich zum Setzen eines der im Feld *wPortStatus* der Rückgabe von "Get Port Status" enthaltenen Bits. Zu beachten ist, dass der für den Feature Selector verwendete Wert dabei nicht zwangsläufig gleich der entsprechenden Bitnummer innerhalb von *wPortStatus* ist.

Sinnvolle Werte für den Feature Selector lauten dabei wie folgt:

4 (PORT_RESET) Durch diesen Selector wird Bit4 (Reset) für den Portstatus gesetzt.

2 (PORT_SUSPEND) Durch diesen Selector wird Bit2 (Suspend) für den Portstatus gesetzt.

8 (PORT_POWER) Durch diesen Selector wird Bit8 (Port Power) für den Portstatus gesetzt.

21 (PORT_TEST) Durch diesen Selector wird Bit11 (Port Test Mode) für den Portstatus gesetzt. Bei Wahl dieses Features dient das obere Byte des *wIndex*-Feldes ferner als Zusatzangabe für die Art des zu nutzenden Test-Modus. Details sind Tabelle 11-24 in [Spek] zu entnehmen.

22 (PORT_INDICATOR) Durch diesen Selector wird Bit12 (Port Indicator Control) für den Portstatus gesetzt. Bei Wahl dieses Features dient das obere Byte von *wIndex* zur Farbauswahl für die mit dem Port verbundene LED. Hierbei sind folgende Werte definiert:

Wert	Farbwahl
0	Automatisch
1	bernsteinfarben
2	grün
3	LED aus

Clear Port Feature

Über diesen Request lassen sich Features für einen Port deaktivieren. Auch dies spiegelt sich direkt in den entsprechenden Bits des Portstatus wieder. Zusätzlich werden wieder Änderungsanzeigen des Status mit diesem Request quittiert.

Tabelle 2.33 zeigt die für diesen Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100011B	CLEAR_FEATURE	Feature Selector	Port	0	–

Tabelle 2.33: Format des Requests "Clear Port Feature"

Die für den Feature Selector wählbaren Werte lauten wie folgt:

Statusmodifikation PORT_SUSPEND, PORT_POWER und PORT_-INDICATOR entsprechen den bereits bei "Set Port Feature" beschriebenen Werten und deaktivieren das fragliche Feature. Zusätzlich kann der Port über PORT_ENABLE, welches mit einem Wert von Eins ansprechbar ist, deaktiviert werden.

Änderungsquittierung Über die in der folgenden Tabelle angegeben Werte lassen sich Bits, welche der Hub innerhalb des *wPortChange*-Feldes liefert, zurücksetzen.

Wert	Mnemonic	Bitstelle in wPortChange
16	C_PORT_CONNECTION	0
17	C_PORT_ENABLE	1
18	C_PORT_SUSPEND	2
19	C_PORT_OVER_CURRENT	3
20	C_PORT_RESET	4

Set Hub Descriptor

Durch diesen Request kann ein Hub das Überschreiben seines Klassendescriptors zur Betriebszeit zulassen. Die Unterstützung dieses Requests ist jedoch optional und bei der Mehrheit der auf dem Markt befindlichen Geräte nicht vorhanden.

Tabelle 2.34 zeigt die für diesen Request definierten Feldeinträge.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100000B	SET_DESCRIPTOR	Descriptor Typ und Descriptor Index	0	Descriptor Länge	Descriptor

Tabelle 2.34: Format des Requests "Set Hub Descriptor"

Der neu zu setzende Descriptor wird dem Hub in der Daten-Stufe des Requests mitgeteilt.

2.5 Abschließende Anmerkungen

Die in diesem Kapitel enthalten Informationen reichen aus, um den weiteren Verlauf der Arbeit zu beschreiben und die geforderte Software zu entwickeln. Es ist jedoch klar, dass es nicht sinnvoll war, sämtliche Definitionen und Details aus der USB-Spezifikation in dieses Kapitel zu übernehmen. Insbesondere die Beachtung der Einschränkungen der elektrischen Komponenten obliegen, wie schon erwähnt, eher dem Produzenten der für die Buskommunikation verwendeten ICs und spiegeln sich nicht unbedingt in der darauf basierenden Software wieder.

Wichtig ist es jedoch, den Verlauf der Buskommunikation verstehen und deuten zu können. Für eine korrekte Zusammenstellung der Hardware auf Basis der beteiligten ICs ist ein Verständnis der elektrischen Seite des USB daher unabdingbar.

Die in diesem Kapitel gezeigten Descriptoren, Request-Typen und der Ablauf der einzelnen Transfers finden sich dann nach Vervollständigung der

Hardware als Strukturen und Kontrollfluss der zu programmierenden Software wieder. Daher war eine mitunter detaillierte Angabe der hierfür verwendeten, genau spezifizierten Werte notwendig.

Auf Basis der in diesem Kapitel gegebenen Definitionen und Ablauferläuterungen kann im Folgenden sowohl die Hardware als auch die Software erstellt werden.

Kapitel 3

Hardware

Wie eingangs bereits erwähnt, beschäftigt sich dieses Kapitel mit der Beschreibung der verwendeten Hardwarekomponenten. Bei Beginn der Arbeit waren Mikrocontroller der AT90USB-Serie nur relativ schwierig zu beschaffen. Dieser Zustand hat sich erfreulicherweise gebessert, so dass nun Controller in nennenswerten Stückzahlen verfügbar sind.

Als Herzstück der Hardware kommt ein Gerät des Typs AT90USB1287 zum Einsatz. Es gilt also, eine Schaltung zu entwickeln, mit welcher dieser Controller in Betrieb genommen, programmiert und letztlich auch physisch mit anderen USB-Geräten verbunden werden kann. Außerdem wird ein genauerer Blick auf den Controller, seine relevanten Register und die Familie der AVRs im Allgemeinen geworfen. Doch im Moment bedarf es noch einer Klärung der Begrifflichkeiten – Vor der weiteren Beschäftigung mit speziellen Chips gilt es zunächst, eine elementare Frage zu beantworten: Was ist eigentlich ein Mikrocontroller?

3.1 Mikrocontroller

Als Mikrocontroller bezeichnet man solche Geräte, bei denen nicht nur die CPU, sondern auch noch weitere zum Betrieb nötige Komponenten auf einem einzigen Chip untergebracht sind. Zu diesen Zusatzkomponenten zählen typischerweise vor allem Speicher, Ein- und Ausgabeschnittstellen sowie zusätzliche Peripherie wie Timer, Taktgenerator oder auch Watchdogschaltungen. Damit integrieren solche Controller alle Einheiten eines "üblichen" Computers in einen IC, woher auch die bereits in der Einführung genannte Bezeichnung "Ein-Chip-Computersystem" stammt. Ziel der Verwendung solcher Controller ist in der Mehrheit der Fälle der Aufbau von Schaltungen zu Mess-, Steuer- und Regelungsaufgaben mit einer möglichst gering gehal-

tenen Anzahl von externen Komponenten. Idealerweise besteht die gesamte Schaltung später nur noch aus dem Controller selbst und wenigen externen Schaltelementen, wie beispielsweise Treibern zum Betrieb von Aktoren beziehungsweise Sensoren.

Bezogen auf die Rechenleistung rangieren Mikrocontroller typischerweise am unteren Ende des aktuell üblichen Leistungsspektrums. Das Augenmerk liegt bei ihrer Herstellung vornehmlich auf geringen Kosten und niedrigem Energieverbrauch. Neben speziell für Mikrocontroller neu designten Prozessoren findet man daher häufig auch modifizierte Kerne älterer CPUs, welche auf die benötigten Einheiten "zusammengestrichen" wurden.

Oft folgt der Lebenszyklus eines Mikrocontrollers der Vorgehensweise: "Einmal programmiert, danach angewendet". Der Controller wird dabei zu Beginn seines Einsatzes mit einem Programm versehen, welches er dann für den Rest seiner Lebensdauer ausführt. Der dafür nötige Befehlscode wird im Allgemeinen in einem günstigen Flashspeicher vorgehalten, während für den Arbeitsspeicher schneller, aber auch kostenintensiver SRAM genutzt wird.

Als Ein- und Ausgabeschnittstellen sind neben einfachen Digitalports je nach Modell eine ganze Palette an weiteren Interfaces vorhanden: UART, USART, I^2C, CAN und – besonders im Rahmen dieser Arbeit von Interesse – auch USB sind nur einige Beispiele für die breitgefächerte Auswahl an I/O-Komponenten, die sich bei Mikrocontrollern finden lassen. Oftmals ist auch noch ein spezielles Bussystem zur Erweiterung der Möglichkeiten des Controllers vorgesehen, beispielsweise zum Anschluss von externem RAM und ähnlichem. Abbildung 3.1 zeigt den schematischen Aufbau eines Mikrocontrollers.

3.2 Die Familie der Atmel AVR-Controller

Eine bekannte und recht umfangreiche Familie von Mikrocontrollern ist die Baureihe AVR der Firma Atmel. Wegen ihres guten Preis-Leistungs-Verhältnisses und vor allem des breiten Angebots an Entwicklungstools aller Art haben sich diese Controller eine große Anhängerschaft erschlossen.

Bei allen AVRs handelt es sich um 8Bit-Mikrocontroller[1] in Harvard-Architektur, es liegen also getrennte Speicherbereiche für Programm- und Datenspeicher vor. Die gesamte Baureihe nutzt einen RISC-Prozessorkern, der ursprünglich auf die Studenten Alf-Egil Bogen und Vegard Wollan zurückgeht. Diese Entwickelten das Core zu Anfang der 90er Jahre im Rahmen ihrer

[1]Neben den AVRs existiert inzwischen noch eine namentlich ähnliche Reihe in 32Bit-Ausführung: AVR32. Diese ist jedoch technisch gesehen nicht weiter mit der AVR-Familie verwandt.

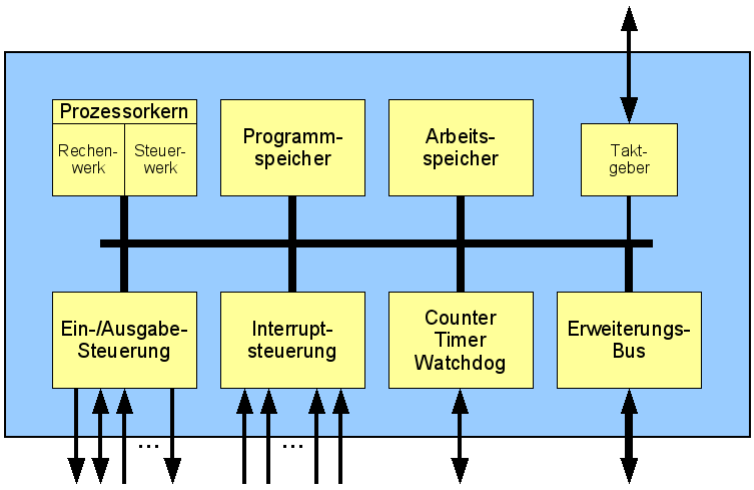


Abbildung 3.1: Schemadarstellung des Aufbaus eines Mikrocontrollers. Vergl. auch [UB02].

Diplomarbeit an der Universität Trondheim/Norwegen. Wollan und Bogens Design wurde später von Atmel aufgekauft.

Bereits bevor die Entwicklung des Kerns abgeschlossen war, liefen Arbeiten an einem Compiler zur Erzeugung von ausführbarem Code aus Hochsprachen. Diese parallele Entwicklung ermöglichte weitreichende Optimierungen des Kerns und führte somit zu einigen Veränderungen im endgültigen Design. Somit kann der AVR-Kern nun besonders effizient mit aus Hochsprachen wie C erzeugtem Code umgehen. Neben dem Austausch einiger Befehle und dem Hinzufügen eines weiteren Adressregisters haben die AVRs diesem Optimierungsprozess auch die Möglichkeit der direkten Adressierung von bis zu 64kB RAM zu verdanken. Eine Kachelung des Speichers, wie sie ursprünglich mit einem sogenannten "paged direct accessing mode" angedacht war, entfiel, da sie für Compiler nur sehr umständlich zu handhaben ist. Nähere Informationen zu dieser Optimierungsphase können [Myk] entnommen werden.

Das AVR-Core besitzt eine zweistufige Pipeline mit den Phasen "Fetch" und "Execute", wodurch die jeweils nächste Instruktion schon geladen werden kann, während die vorherige sich noch in Ausführung befindet. Dies erlaubt es, die Mehrzahl der vorhandenen Maschinenbefehle in nur einem Takt auszuführen, was Controllern der AVR-Serie einen oft nicht unerheblichen Geschwindigkeitsvorteil vor Konkurrenzprodukten verschafft.

Jeder AVR verfügt über 32 sogenannte "General Purpose"-Register[2]. Je-

[2]Tatsächlich ist diese Angabe Atmels nur mit der Einschränkung richtig, dass einige

des dieser Register ist direkt mit der ALU verbunden und kann als Quell- oder Zielregister für Operationen dienen. Insbesondere bedeutet dies, dass es kein Akkumulatorregister gibt. Ein umständliches "Umherschieben" von Daten zwischen den Registern entfällt damit weitestgehend. Die obersten sechs Register sind ferner zu drei 16Bit-Registern zusammenfassbar und können dann zur indirekten Adressierung genutzt werden[3].

Alle AVRs besitzen einen Flashspeicher für den Programmcode, SRAM als Arbeitsspeicher und zusätzlich ein EEPROM zur persistenten Speicherung von Daten aus dem ausgeführten Programm heraus. Die Kapazität der verschiedenen Speicher variiert je nach Modell. Die verfügbare Kapazität beim Programmspeicher erstreckt sich bei Erstellung dieser Arbeit zwischen einem und 256kB. Alle Controller sind "in system programmable", was bedeutet, dass sie zum Programmieren nicht aus der Schaltung, in der sie eingesetzt sind, herausgenommen werden müssen. Das Beschreiben des Flash sowie das Setzen diverser weiterer Einstellungen erfolgt dabei im Reset-Modus des Chips. Verschiedene Sleep-States zum Energiesparen, eine Brownout-Detection zur Reaktion auf einen Einbruch der Versorgungsspannung, ein interner Oszillator, Hardwareunterstützung für einen Stack sowie eine Watchdogschaltung verlängern die Liste der Features, die jeder Controller bietet, weiter.

Innerhalb der AVR-Familie existieren einige Untergruppen, welche den Controller-Kern jeweils um einige besondere Befehle und Funktionalitäten erweitern. Den unteren Rand des Spektrums bildet aktuell die Reihe der tinyAVRs. Diese unterstützen einen kleineren Befehlssatz und weniger Features als ihre leistungsstärkeren Verwandten aus der ATmega-Serie. Modellabhängig sind bei den ATmegas nahezu alle Schnittstellen verfügbar, die sich überhaupt in der Welt der Mikrocontroller finden lassen. Neben Pins zur digitalen Ein- und Ausgabe findet man PWM-Ausgänge, Analog-Komparatoren und A/D-Wandler, außerdem auch Hardwareunterstützung für diverse Arten der Kommunikation mit anderen Komponenten: SPI, USART, I^2C/TWI, CAN sowie ein Debugging-Interface nach dem JTAG-Standard. Am oberen Ende des Leistungsspektrums schließlich rangieren spezielle Modelle zum Betrieb von LCDs, Licht- oder Motorsteuerungen und ähnlichem. Bei diesen handelt es sich üblicherweise um erweiterte ATmega-Versionen, die entweder unter einer ATmega-Kennung oder als AT90-Modelle verkauft werden.

Abhängig von Komplexität und Anzahl der benötigten Ein- und Ausgänge, sind die meisten Modelle in verschiedenen Bauformen verfügbar. Der ATmega8515 ist beispielsweise gleich in vier verschiedenen Package-Typen zu

wenige Befehle nicht auf die unteren 16 Register anwendbar sind.

[3]Diese Adressregister werden auch X, Y und Z genannt.

bekommen: als PDIP mit 40, sowie TQFP, MLF und PLCC mit jeweils 44 Pins. Außerdem lassen sich die meisten Typen in verschiedenen Ausfertigungen erwerben, die mit unterschiedlichen Spannungen, meist im Bereich von 1,8V bis 5,5V, betrieben werden. Dies erhöht die Flexibilität beim Einsatz solcher Controller weiter.

Wie schon anfänglich beschrieben, existiert nun auch eine Baureihe mit Unterstützung für USB-Interfaces. Namentlich ist dies die Reihe AT90USB, auf welche im Folgenden eingegangen wird.

3.3 Die Baureihe AT90USB

Bei den Controllern der Serie AT90USB handelt es sich um ATmega-Modelle, welche um ein USB-Interface erweitert wurden. Alle diese Controller besitzen Hardwareunterstützung zum Betrieb als USB-Device. Die Familie untergliedert sich jedoch weiter in zwei Gruppen: Es existieren sowohl Geräte mit als auch ohne zusätzliche Unterstützung des Hostmodus[4]. Zum Zeitpunkt der Erstellung dieser Arbeit führt Atmel Controller ohne Hostmodus unter der Bezeichnung AT90USBx6 und solche mit Hostmodus als AT90USBx7. Anhand des x ist jeweils die ATmega-Version ersichtlich, auf der das jeweilige Modell basiert. Interessant im Rahmen dieser Arbeit sind selbstverständlich nur Modelle letztgenannter Serie, denn ohne Hardwareunterstützung ist ein Betrieb im Hostmodus nicht realisierbar.

Für diese Arbeit nutzen die Autoren den zum Erstellungszeitpunkt leistungsstärksten verfügbaren Controller mit Host-Unterstützung. Der folgende Abschnitt beschreibt diesen etwas näher.

3.4 Der Mikrocontroller AT90USB1287

3.4.1 Allgemeines

Der AT90USB1287 ist ein als Modifikation des ATmega128 entworfener AVR-Controller und verfügt damit über 128kB Flashspeicher, ein 4kB EEPROM und 8kB SRAM.

Die Liste der vorhandenen Features ist lang: Neben der im nächsten Abschnitt genauer beschriebenen USB-Funktionalität verfügt der Controller über eine USART zur seriellen Datenübertragung, ein SPI-Interface sowie I2C/TWI-Unterstützung. Er bringt einen Real Time Counter und zwei weitere 8- sowie ebenfalls zwei 16Bit-Counter/Timer und damit verbunden

[4]Genauer gesagt handelt es sich um Unterstützung für den OTG-Modus.

auch die Möglichkeit hardwaregestützter Erzeugung von PWM-Signalen mit. Zusätzlich besitzt der AT90USB1287 einen acht-Kanal-Analog-Digital-Konverter mit 10Bit-Auflösung sowie einen integrierten Analog-Komparator. Alle Standardfeatures der AVR-Reihe, wie beispielsweise der interne Oszillator, sind selbstverständlich auch vorhanden. Die Möglichkeit zum Anschluss von externem SRAM und externen Interruptquellen sorgen weiter für mehr Variabilität im Umgang mit diesem Modell. Ebenfalls integriert sind JTAG-Unterstützung, Brownout-Detection und Unterstützung von sechs verschiedenen Sleep-Modi.

Nicht eben kurz stellt sich jedoch leider auch die Liste der Ungenauigkeiten und Fehler im Datenblatt ([Dat]) des Controllers dar. Dieses wurde von Atmel ausdrücklich als "vorläufig" markiert und ist wohl ein im Anpassungsprozess befindliches ATmega128-Datenblatt. So enthält beispielsweise das Blockschaltbild des Mikrocontrollers einen Port mit der Bezeichnung "G", der bei Geräten der AT90USB-Reihe gar nicht existiert – wohl aber beim ATmega128. Das USB-Interface hingehen sucht man in besagtem Blockbild vergeblich.

Einen guten Überblick über die Funktionseinheiten des AT90USB1287 bietet Abbildung 3.2. Sie zeigt ein durch die Autoren korrigiertes Blockbild, welches sogleich auch um die Anschlüsse für USB erweitert wurde.

Der AT90USB1287 kann mit Taktraten von bis zu 16MHz und Spannungen zwischen 2,7V und 5,5V betrieben werden. Verfügbar ist er sowohl in QFN- als auch in TQFP-Bauform mit jeweils 64 Pins, wobei im Rahmen dieser Arbeit ein Controller im TQFP-Package zum Einsatz kommt. Mit einer Pinbreite von etwa 0,4mm und einem mittleren Abstand der Pins von 0,8mm läßt sich dieses Package trotz SMD-Bauform noch mit vertretbarem Aufwand von Hand verarbeiten.

Von den zur Verfügung gestellten Einheiten ist, mit Augenmerk auf die bevorstehende Softwareentwicklung, vor allem die Verwendung der USART für Debug- und Statusausgaben interessant. Die zu deren Nutzung benötigten zusätzlichen Bauelemente und Schaltungen sind im Rahmen vorangegangener Projekte bereits erprobt. Ähnlich verhält es sich mit der für die Programmierung des Controllers nötige Beschaltung. Die hierfür benötigten Verdrahtungen werden später in Abschnitt 3.5.2 gezeigt.

Zunächst ist es jedoch notwendig, sich mit der Anbindung des AT90USB1287 an den USB auseinander zu setzen. Elemente, welche eventuell zusätzlich für die elektrische Verbindung zum USB benötigt werden, können beim folgenden Entwurf dann direkt in die Schaltung integriert werden.

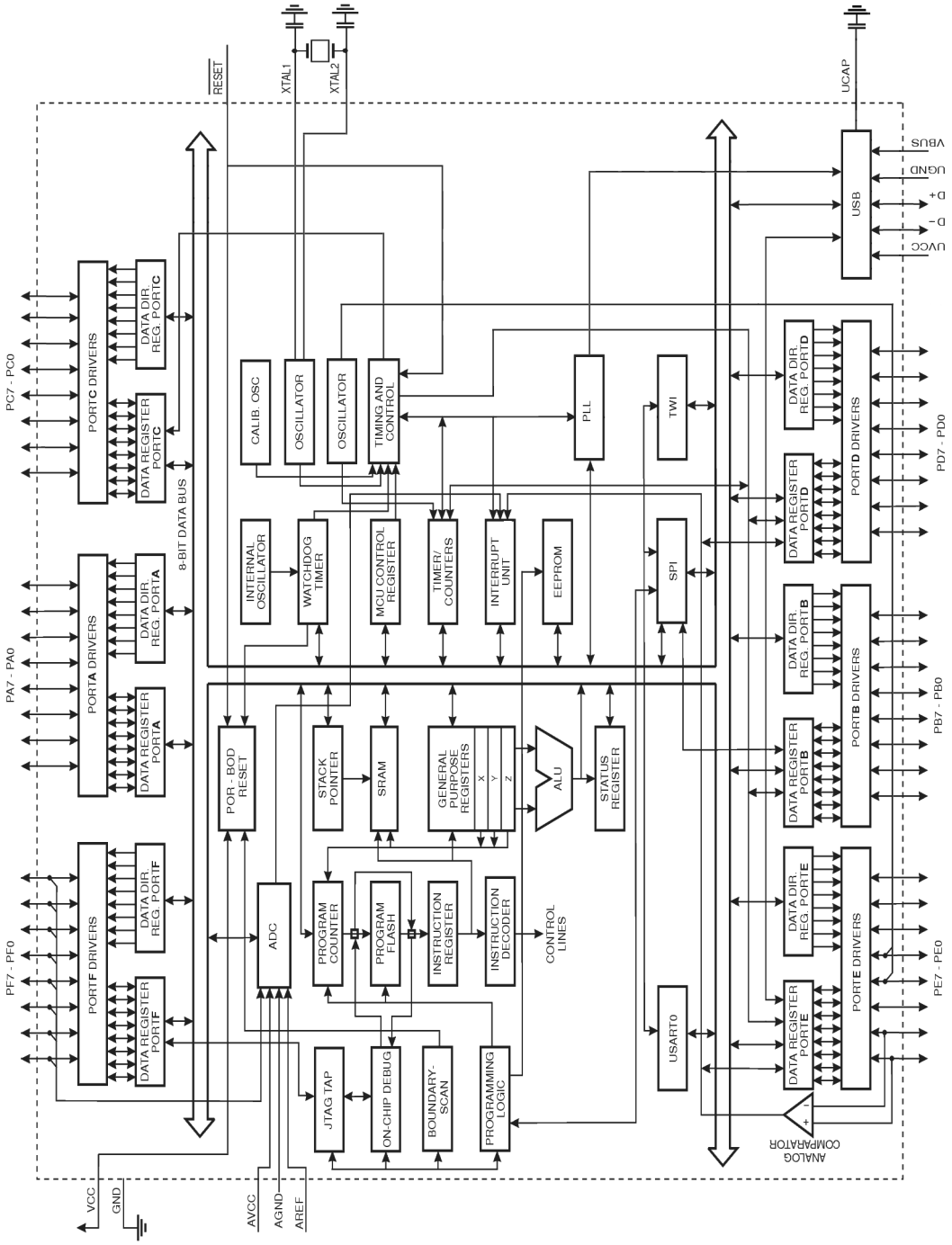


Abbildung 3.2: Korrigiertes Blockbild des AT90USB1287

3.4.2 Übersicht zur USB-Funktionalität

3.4.2.1 Grundlegende Eigenschaften

Bezüglich der USB-Funktionalität bringt der AT90USB1287 Hardwareunterstützung für den Einsatz als USB-Function-, (reduced) Host- und OTG-Controller mit. Er erfüllt alle Anforderungen, die durch die USB-2.0-Spezifikation ([Spek]) sowie auch deren Zusatz "On-The-Go Supplement to the USB 2.0 Specification" ([Speh]) an USB-Geräte gestellt werden. Einschränkend ist zu beachten, dass der als wesentliches Feature gegenüber USB-1.1 in 2.0 hinzugefügte High-Speed-Modus jedoch nicht unterstützt wird. Damit ist der Controller auf den Betrieb mit Übertragungsgeschwindigkeiten des Low- beziehungsweise Fullspeed-Modus beschränkt, womit also Datenraten von 1,5 und 12Mbit/s erzielt werden können.

Die über die USB-Schnittstelle zu übertragenden beziehungsweise empfangenen Daten legt der Chip in einem eigens dafür zur Verfügung gestellten DPRAM ab. Die 832 Byte dieses Speichers lassen sich variabel als Pufferspeicher auf bis zu sieben Pipes oder Endpoints verteilen. Im Hostmodus können dabei eine maximal 64 Byte fassende Default Control Pipe, eine bis zu 256 Byte und fünf weitere bis zu 64 Byte große IN- oder OUT-Pipes verwaltet werden. Diese können jeweils im Ein- oder Zweibank-Interleaved-Modus betrieben werden, je nachdem, welche Anforderungen die jeweilige Anwendung an Geschwindigkeit und/oder Speicherplatz stellt[5]. Im Device-Modus gelten analoge Angaben für die zur Verfügung stehenden Endpoints. Wichtig für den noch folgenden Schaltungsaufbau ist bezüglich des Speichers vor allem die Erkenntnis, dass zur Aufnahme der Daten keine externen Baugruppen wie Schieberegister oder ähnliches benötigt werden.

Die für den Betrieb der USB-Einheit notwendige Taktrate von 48MHz kann über eine interne PLL (Phase-Locked Loop) erzeugt werden. Hierzu ist lediglich eine Beschaltung des AT90USB1287 mit einem 16 oder 8Mhz Quarz beziehungsweise Oszillator nötig[6]. Auch diesbezüglich werden also keine besonderen Anforderungen an die Schaltung gestellt.

3.4.2.2 Pin-Übersicht

Sowohl das Blockschaltbild als auch das Pinout in Abbildung 3.3 zeigen jedoch natürlich einige Anschlüsse des Mikrocontrollers, die zur Funktion der USB-Einheit beschaltet werden müssen. Diese Pins sind im Einzelnen:

[5] Natürlich darf der maximal zur Verfügung stehende Speicherplatz von 832 Byte nicht überschritten werden.

[6] Der Betrieb des USB-Interfaces bei anderen Taktraten ist jedoch nicht möglich.

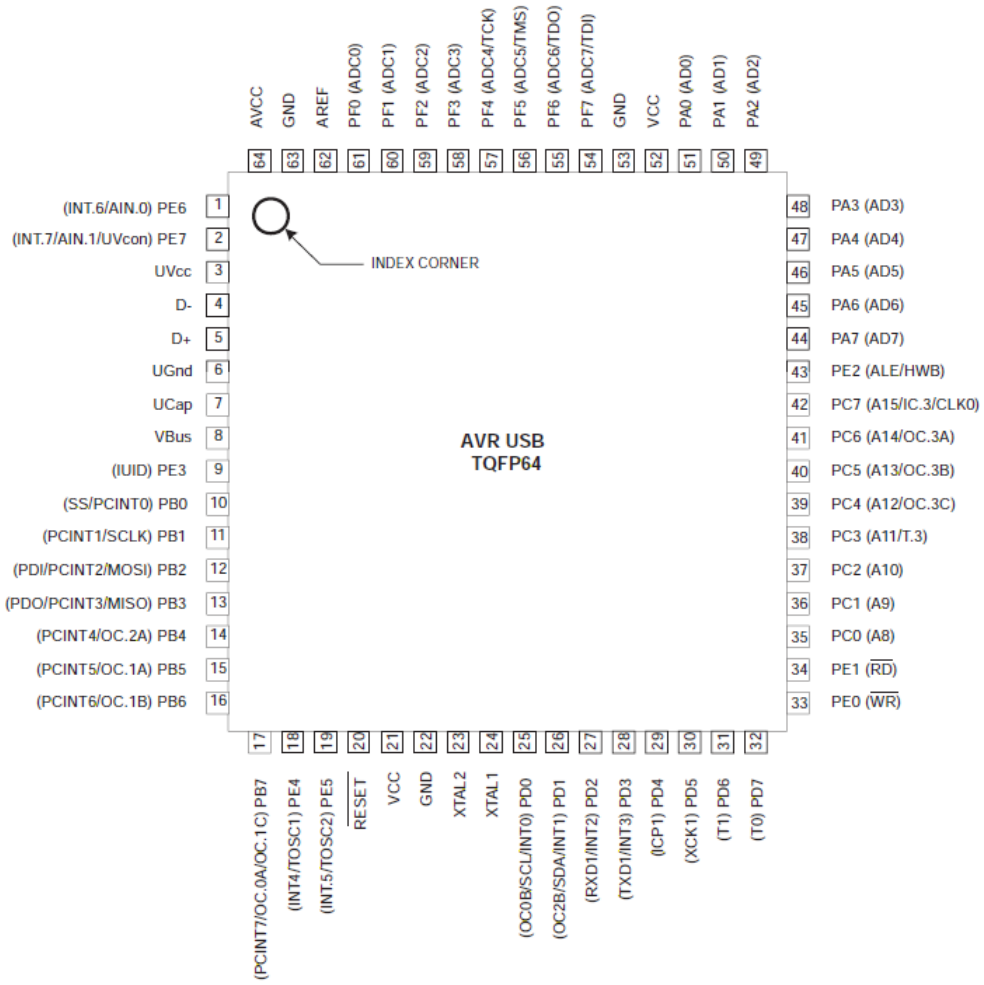


Abbildung 3.3: Pinout des AT90USB1287 im TQFP64-Package

PIN4 / D− sowie PIN5 / D+ Anschlüsse für die D+ und D− Leitungen des USB. Diese sind jeweils mit einem in Reihe geschalteten 22Ω-Widerstand an die entsprechenden Pins der USB-Buchse zu führen.

PIN3 / UVCC sowie PIN6 / UGND Anschlüsse für die Spannungsversorgung des internen Spannungsreglers (Pad-Regulator) des USB-Interfaces. Der Pad-Regulator erzeugt den für die Kommunikation mittels USB nötigen Pegel von 3,3V für die Signale auf D+ und D−.

PIN7 / UCAP Anschluss für einen externen Kondensator, der für die Spannungsstabilisierung des 3,3V Pegels am Ausgang des Pad-Regulators

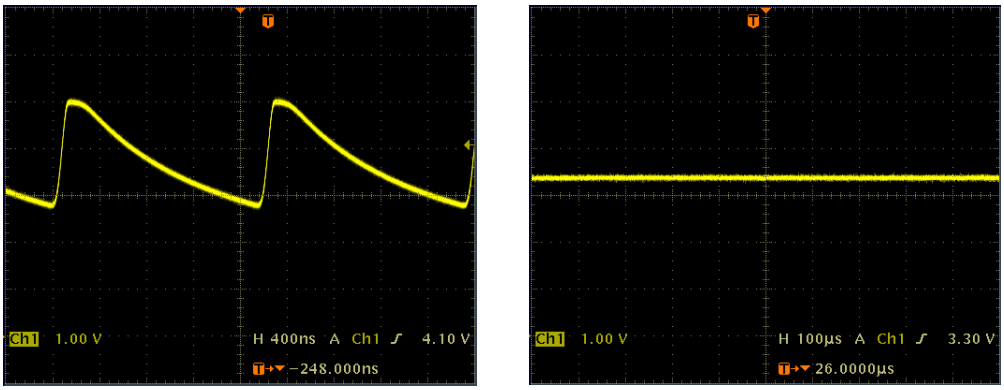


Abbildung 3.4: Oszilloskopbild der Signale an *UCAP* – Links: ohne externe Kapazität, rechts: mit $1\mu F$ extern beschaltet.

sorgt. Der *UCAP*-Pin muss mit einem Kondensator der Kapazität $1\mu F$ gegen Masse verbunden werden. Ein Durchmessen des Pins mittels Oszilloskop offenbart die vom Pad-Regulator genutzte Methode zur Erzeugung der 3,3V. Dabei wird eine kleine interne Kapazität in schnellem Wechsel ge- und entladen. Ein Abgreifen von *UCAP* ohne Anschluss des externen Kondensators zeigt dementsprechend die in Abbildung 3.4 links dargestellte Spannungskurve. Nach Anschluss der externen Kapazität von $1\mu F$ ergibt sich die rechts abgebildete Spannung von konstant 3,3V.

PIN8 / VBUS Der *VBUS*-Pin wird direkt an den Pluspol der Spannungsversorgung des USB angeschlossen und dient zum Monitoring des Status dieser Leitung. Diese Spannungsversorgung wird im Folgenden mit "VBus" bezeichnet.

PIN2 / PE7(UVCON) Dieser Pin kann im Hostmodus dazu genutzt werden, eine externe Versorgungspannung für den VBus ein- beziehungsweise aus zu schalten.

PIN9 / PE3(UID) *UID* ist weiterer Eingangspin, der dazu genutzt werden kann, die Wahl zwischen Host- und Device-Modus von der Hardware treffen zu lassen. Liegt Massepotential an dem Pin an, wenn die USB-Funktionalität des Controllers zugeschaltet wird, so kommt der Host-Controller zum Einsatz. Bleibt der Pin offen oder liegt auf High-Pegel, so aktiviert die Hardware den Device-Controller. Diese Funktionalität kann auch per Software übergangen werden.

Diese Übersicht soll zunächst genügen, da sie bezüglich des USB-Anschlusses genügend Hintergrundwissen vermittelt, um den Entwurf des Ex-

perimentierboards durchzuführen. Weitere Einzelheiten, die sich auf die Programmierung der USB-Funktionen beziehen, folgen später in Abschnitt 3.6.

3.5 Entwicklung der Experimentierschaltung

Beim Entwurf einer Platine wird üblicherweise in drei Teilschritten vorgegangen. Zunächst gilt es, ein Schema der Schaltung zu entwickeln. Im folgenden Arbeitsschritt können dann die im Schema verwendeten Bauteile im Modell der Platine geeignet angeordnet werden. Den letzten Arbeitsschritt des Entwurfs stellt das sogenannte "Routing" dar, welches in diesem Kontext für die Festlegung des Verlaufs der einzelnen Leiterbahnen auf der Platine steht.

Für die vorliegende Arbeit wurden die benötigten Elemente auf zwei Platinen verteilt. Elemente, die in einem späteren Einsatz dauerhaft vorhanden sein müssen, wurden auf der eigentlichen Experimentierplatine platziert. Hierzu zählen neben dem Controller selbst beispielsweise Elemente der Stromversorgung oder auch die Baugruppe zum Anschluss an den USB. Andere Baugruppen – namentlich die Schaltungen zum Anschluss an die serielle und parallele Schnittstelle des PCs – wurden auf eine kleinere Adapterplatine ausgelagert. Diese Anschlussmöglichkeiten sind hauptsächlich für Entwicklungszwecke und Programmierung interessant und werden später aller Wahrscheinlichkeit nach nicht mehr benötigt.

Eine solche Adapterplatine wurde bereits im Vorfeld der Arbeit im Rahmen anderer Projekte von den Autoren entworfen und eingesetzt. Die Nutzung der Platine ist also im Sinne der Verwendung normierter Hardware auch für den in dieser Arbeit erstellten Aufbau sinnvoll.

3.5.1 Experimentierboard

Um die Erläuterungen zur Schaltung möglichst überschaubar zu gestalten, sei im Vorfeld bereits auf Abbildung 3.5 hingewiesen. Diese zeigt das komplette Schema der Schaltung, wobei der AT90USB1287 erwartungsgemäß im Mittelpunkt des Aufbaus steht. Die einzelnen Baugruppen, beziehungsweise die Funktion der vorhandenen Elemente um den AVR-Controller herum, werden im Folgenden beschrieben.

3.5.1.1 Spannungsversorgung

Für den Anschluss der Versorgungsspannung ist die Buchse *X1* in Verbindung mit einem Brückengleichrichter (*B1*) vorgesehen. Dies ermöglicht den

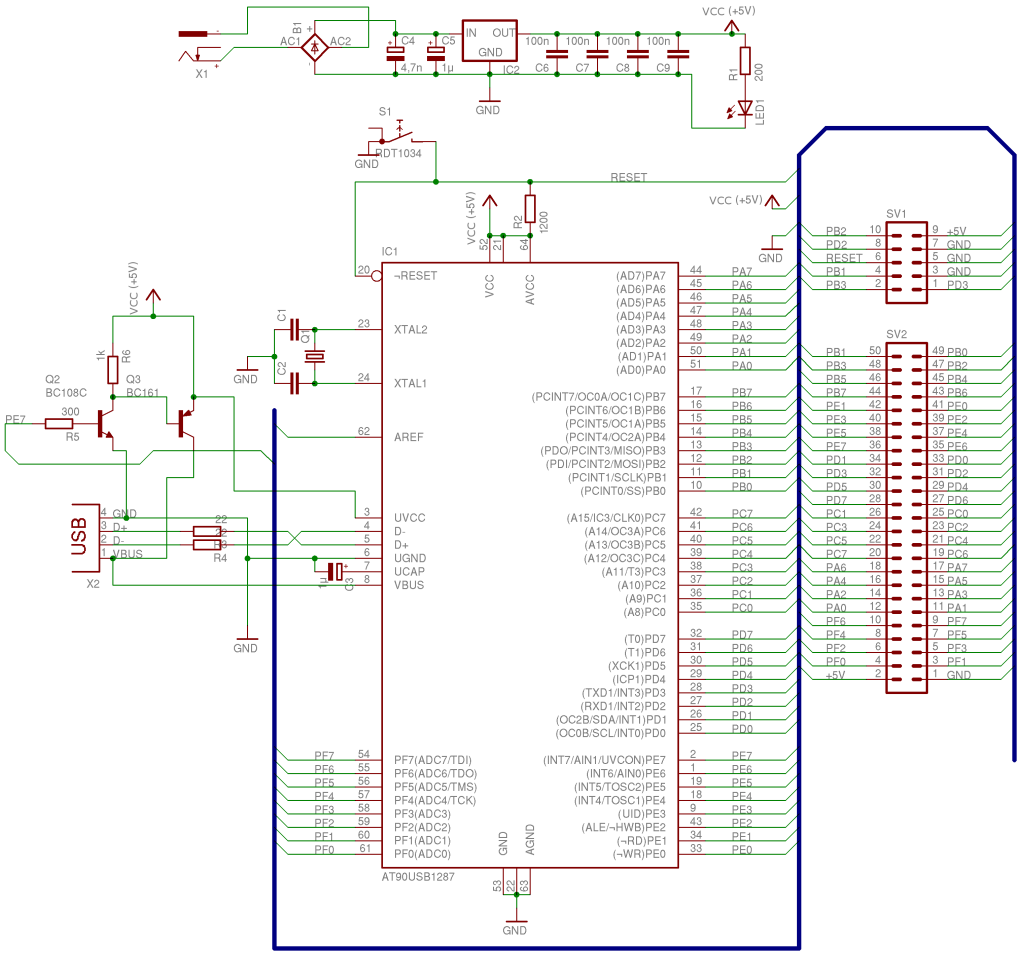


Abbildung 3.5: Schematic der Experimentierplatine – Mitte: AT90USB1287, Oben: Elemente der Spannungsversorgung und Reset-Taster, Links: Bauteile für USB und Takterzeugung, Rechts: Schnittstelle zur Adapterplatine sowie Abgriffe für freie Pins des Controllers.

Anschluss sowohl von Gleich- als auch von Wechselstrom liefernden Netzteilen und trägt damit erheblich zur Robustheit des Aufbaus im experimentellen Einsatz bei.

Die gleichgerichtete Spannung am Ausgang der Graezbrücke wird durch die beiden Kondensatoren *C4* und *C5* stabilisiert, wobei deren unterschiedliche Kapazität für eine Filterung von sowohl hohen als auch tiefen Störfrequenzen sorgt. Die so geglättete Spannung wird dann einem Festspannungsregler des Typs 78LS05 zugeführt. Dieser garantiert an seinem Ausgang eine konstante Spannung von 5V, solange er eingangsseitig mit mindestens 6V

versorgt wird. Die vom 78LS05 erzeugte Ausgangsspannung wird im Folgenden als VCC bezeichnet.

Zur rudimentären Kontrolle der Versorgungsspannung dient eine LED, die im laufenden Betrieb permanent leuchtet. Sie wurde in Reihe mit einem geeigneten Vorwiderstand direkt zwischen Masse und VCC geschaltet. Ferner sind noch die vier Kondensatoren *C6–C9* vorgesehen. Diese werden im späteren Layout physisch nahe an den für die Spannungsversorgung vorgesehen Pins des AT90USB1287 platziert und dienen mit ihren $100nF$ als Abblockkondensatoren. So glätten sie eventuell eingestreute Störungen in der Spannungsversorgung und stabilisieren nochmals die Stromzufuhr zum Mikrocontroller.

3.5.1.2 Elemente zur Takterzeugung

Zur Takterzeugung wird ein einfacher Schwingquarz (*Q1*) genutzt, aus dessen Oszillation der AT90USB1287 intern einen stabilen Takt generieren kann. Für den Anschluss werden außer dem Quarz selbst nur noch die beiden $22pF$-Kondensatoren *C1* und *C2* benötigt. Der Quarz wird zwischen die beiden Pins *XTAL1* und *XTAL2* des Controllers geschaltet. Die Pins müssen nun nur noch mit je einem der besagten Kondensatoren gegen Masse verbunden werden.

3.5.1.3 Reset-Taster

Der Schließer *S1* bietet die Möglichkeit, einen Reset des Mikrocontrollers herbeizuführen. Dafür wird er direkt zwischen den invertierenden Reset-Eingang des AVR und Masse geschaltet. Der Pin wird zusätzlich mit einem PullUp-Widerstand (*R2*) gegen VCC verbunden. Die Hardware des Controllers führt einen Reset immer dann aus, wenn für die Dauer von mindestens zwei Takten Low-Pegel am Resetpin anliegt.

3.5.1.4 Schnittstelle zur Adapterplatine

Auf die Schnittstelle zur Adapterplatine sind sowohl die Pins der USART als auch die für die Programmierung benötigten Pins des Controllers zu führen. Für die physische Verbindung wird eine zehnpolige Wannenbuchse (*SV1*) eingesetzt.

Im Falle der USART lauten die benötigten Pins *TxD* und *RxD*. Sie werden einfach auf die Anschlüsse der Wannenbuchse geklemmt. Weitere zur Funktion der seriellen Kommunikation notwendige Komponenten befinden sich auf der Adapterplatine. Nähere Informationen dazu finden sich in Abschnitt 3.5.2.

Für das ISP[7]-Interface sind die Pins *PDO/MISO*, *PDI/MOSI* und *SCK* sowie auch der Reset-Pin des AVRs wichtig. Auch diese werden an die Wannenbuchse herangeführt.

Zusätzlich ist jeweils noch ein Masseausgleich der beteiligten Geräte nötig, weshalb auch GND über die Buchse verteilt werden muss. Außerdem bietet es sich an, die Elemente der Adapterplatine über die Hauptplatine mit Strom zu versorgen. Daher wurde auch VCC auf die Wannenbuchse geschaltet.

3.5.1.5 Anschluss an den USB

Da der Controller im USB-Hostmodus betrieben werden soll, wird zum Anschluss von Peripherie eine USB-Buchse des Typs A verbaut. Außerdem soll die USB-Einheit des AT90USB1287 ständig mit Strom versorgt werden. Dafür werden die Pins *UVCC* und *UGND* jeweils mit VCC beziehungsweise GND beschaltet.

D+ und *D−* werden – wie schon in Abschnitt 3.4.2.2 angekündigt – jeweils über einen Widerstand von 22Ω mit den entsprechenden Anschlüssen der Buchse verbunden. Im Schema sind dies die beiden Widerstände *R3* und *R4*.

Der Pin *VBUS* wird ohne weitere Elemente direkt auf den VBUS-Anschluss der Buchse gelegt. Die Masseleitung der USB-Buchse ist mit *UGND* und daher hier auch mit GND zu verbinden.

Nun gilt es noch, die Spannungsversorgung für über den Bus mit Strom versorgte USB-Geräte herzustellen. Die Möglichkeit, die Stromversorgung solcher Geräte über den dafür angebotenen Pin *UVCON* zu steuern, soll dabei nicht ungenutzt bleiben.

Um VCC auf den (anfänglich auf keinem definierten Potential liegenden) VBUS durchzuschalten, wird ein PNP-Transistor benötigt. Hierfür wurde ein BC161 mit Emitter gegen VCC und Kollektor gegen VBUS verbunden. Da *UVCON* eine highaktive Schaltung des VBus vorsieht, der PNP-Transistor aber bekanntermaßen eine gegenüber dem Emitter negative Basisspannung zum Durchschalten benötigt, wird mittels der beiden Widerstände *R5* und *R6* sowie dem NPN-Transistor *Q2* eine Inverterschaltung aufgebaut. Liefert *UVCON* High-Pegel, so liegt damit VCC am VBUS-Anschluss der USB-Buchse an.

Abschließend wird noch *UCAP* mit Kondensator *C3* ($1\mu F$) gegen *UGND* beziehungsweise GND verbunden, womit die Schaltung zum Anschluss an den Bus komplett ist.

[7] In System Programming

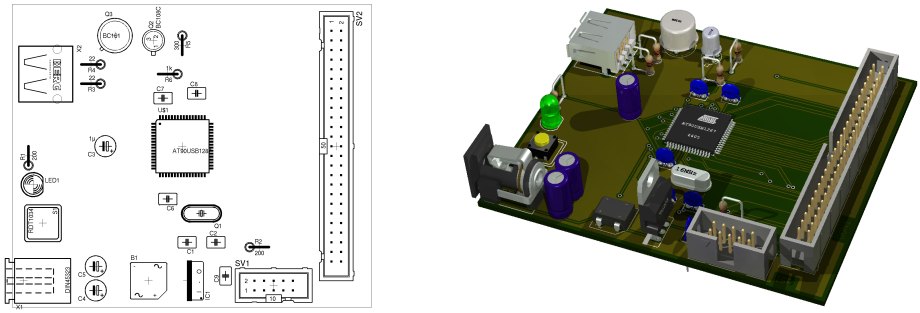


Abbildung 3.6: Layout und Bild der Experimentierplatine

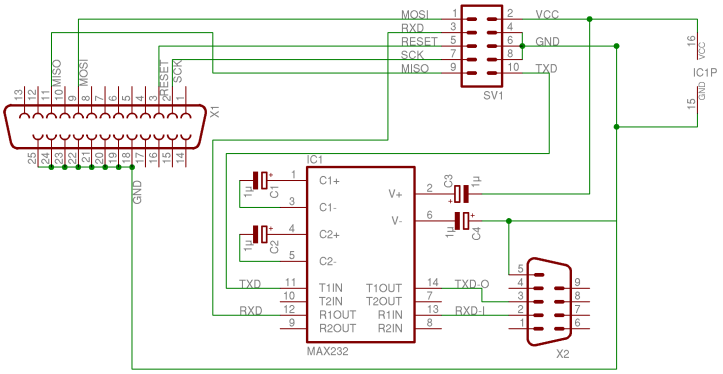


Abbildung 3.7: Schematic der Adapterplatine

3.5.1.6 Abgriffe für freie Pins des AVR

Um die nun noch freien Pins des AVR nach außen hin zugänglich zu machen, wurden sie jeweils auf einen Anschluss der fünfzigpoligen Wannenbuchse *S2* geführt.

Damit ist die Schaltung der Experimentierplatine komplett. Nach geeigneter Bauteilanordung und Routing ergibt sich die in Abbildung 3.6 gezeigte Platine. Eine Bestückungsliste sowie Ätzvorlagen sind unter 7.1 im Anhang dieser Arbeit zu finden.

3.5.2 Adapterplatine

Für die Beschreibung der Adapterplatine wird analog zur Erläuterung der Hauptplatine vorgegangen. Abbildung 3.7 zeigt dabei das Schema der für die Adapterplatine erstellten Schaltung.

Der Abgriff der von der Hauptplatine an den Adapter geführten Signale

erfolgt mit einer Wannenbuchse (*SV1*), so dass beide Platinen einfach durch ein Flachbandkabel mit geeigneten Steckern verbunden werden können.

3.5.2.1 Anschluss an die parallele Schnittstelle

Die Pins der Programmierschnittstelle des AVR werden entsprechend der sogenannten *sp12*-Belegung[8] mit der SubD25-Buchse (*X1*) verbunden. Die Buchse dient dann als Verbindungsmöglichkeit zur parallelen Schnittstelle eines PCs ("Druckerport"). Die *sp12*-Belegung ist gängigen Flashtools für den AVR wie zum Beispiel `avrdude` bekannt, so dass ein Programmieren des Controllers damit auf einfache Weise möglich ist.

3.5.2.2 Anschluss an die serielle Schnittstelle

Die USART des AT90USB1287 bietet die Möglichkeit zur seriellen Kommunikation mit anderen Geräten nach dem RS232-Protokoll. Hierbei ist eine Verbindung der Geräte untereinander mit jeweils zwei Signalleitungen vorgesehen, wobei eine Leitung zum bitseriellen Senden (TxD) und die andere zum ebenfalls bitseriellen Empfang von Daten (RxD) dient. Die Codierung erfolgt denkbar einfach im NRZ-Format (Non Return to Zero).

Es handelt sich um eine asynchrone Art der Datenübertragung, eine zusätzliche Leitung zur Taktübertragung ist also nicht vorgesehen. Im Vorfeld der Übertragung muss daher beiden Kommunikationspartner die Baudrate, mit der übertragen wird, bekannt sein. Eine Synchronisation erfolgt anhand in den Datenstrom eingefügter Start- und Stoppbits.

Die zur Datenübertragung benötigten Pins des AVR – also *TxD* und *RxD* – werden über die Wannenbuchse auf die Adapterplatine geführt. Für die Einhaltung der RS232-Spezifikation ist der von diesen Pins gelieferte TTL-Pegel jedoch nicht ausreichend. Die 0V beziehungsweise 5V müssen noch auf die von RS232 vorgesehen Pegel von 12V für logisch Null und -12V für logisch Eins umgesetzt werden.

Für diese Pegelwandlung kommt ein IC vom Typ MAX232 (*IC1*) zum Einsatz. Er ist explizit für diese Aufgabe vorgesehen und erzeugt mit Hilfe der Elektrolytkondensatoren *C1*–*C4* eine Spannung von -10 beziehungsweise 10V, was für eine stabile Kommunikation ausreicht. Abbildung 3.8 zeigt Auszüge aus dem Datenblatt des MAX232, welchen der Schaltungsaufbau folgt.

Die Ausgänge des MAX232 sind entsprechend der Belegung, wie sie auch am COM-Port des PC zu finden ist, an den SubD9-Stecker *X2* angeschlossen.

[8] "Steve Bolt's Programmer", benannt nach dem Entwickler.

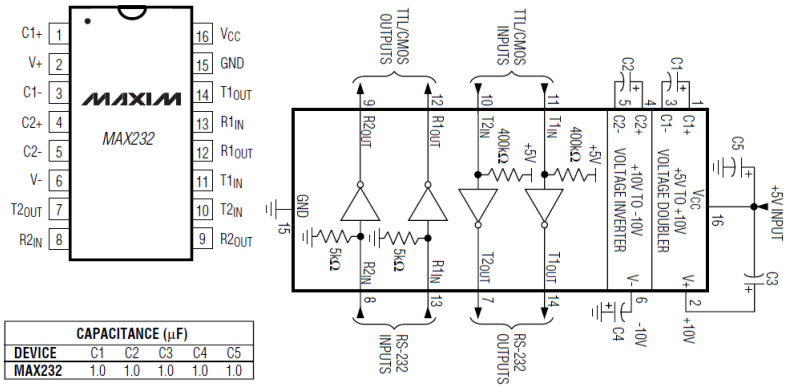


Abbildung 3.8: Auszug aus dem MAX232-Datenblatt: Pinout, Funktionsschema und Kapazität der Kondensatoren.

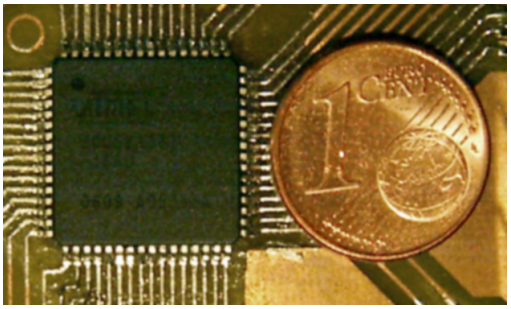


Abbildung 3.9: Größenvergleich des AT90USB1287 zu einem Centstück.

Damit ist das Design der Hardwarekomponenten abgeschlossen. Nach erfolgreicher Herstellung der Platine sowie Ein- beziehungsweise Auflöten der benötigten Bauelemente, ist das Experimentierboard bereit für den Einsatz. An dieser Stelle sei jedoch auf die Schwierigkeit des Auflötens des AT90USB1287 hingewiesen. Abbildung 3.9 verdeutlicht die Größenverhältnisse, mit denen es zu arbeiten gilt. Ungeübten Lötern ist daher von der manuellen Verarbeitung abzuraten.

Mit vorhandenem Arbeitsmaterial fehlt schließlich noch das Wissen um die Schnittstelle zur Software, die der Controller für die USB-Funktionalität bereitstellt. Diese wird im Folgenden betrachtet.

All USB Registers				
USB	USB Device		USB Host	
General	General	Endpoint	Host	Pipe
UHWCON	UDCON	UENUM	UHCON	UPNUM
USBCON	UDINT	UERST	UHINT	UPRST
USBSTA	UDIEN	UECONX	UHIEN	UPCONX
USBINT	UDADDR	UECFG0X	UHADDR	UPCFG0X
OTGCON	UDFNUMH	UECFG1X	UHFNUMH	UPCFG1X
OTGTCON	UDFNUML	UESTA0X	UHFNUML	UPCFG2X
OTGIEN	UDMFN	UESTA1X	UHFLEN	UPSTAX
OTGINT		UEINTX		UPINRQX
		UEIENX		UPERRX
		UEDATX		UPINTX
		UEBCHX		UPIENX
		UEBCLX		UPDATX
		UEINT		UPBCHX
				UPBCLX
				UPINT

Tabelle 3.1: Grobe Gliederung der USB-Register. Nicht aufgeführt: PLLCSR zur Taktkonfiguration.

3.6 Schnittstelle zur Software

3.6.1 Allgemeines

Die Programmierung der USB-Einheit des AT90USB1287 erfolgt über den Zugriff auf insgesamt 51 Register. Diese lassen sich – sieht man einmal von dem Register zur Taktkonfiguration ab – grob in drei Gruppen unterteilen: "USB General Registers", "USB Device Registers" sowie "USB Host Registers".

Die Device-Register gliedern sich dann weiter in "Device General" und "Device Endpoint" Register. Bei den Host-Registern verhält es sich analog: Sie zerfallen in "Host General" sowie "Host Pipe" Register. Tabelle 3.1 gibt eine Übersicht über die Register und ihre Gruppenzugehörigkeit.

Schon die schiere Anzahl der Register lässt vermuten, dass es sich bei der Programmierung des USB-Controllers um eine deutlich komplexere Aufgabe handelt, als dies beispielsweise bei den Einheiten für die Kommunikation über I^2C oder gar mittels der einfachen USART der Fall ist.

Nachfolgend sollen die für die Programmierung des Experimentierboards nötigen Register beziehungsweise deren Bits beschrieben werden. Im Rah-

men dieser Arbeit uninteressante Konfigurations- und Statusinformationen bleiben hier außen vor – eine vollständige Registerliste ist dem Datenblatt [Dat] zu entnehmen.

Wegen der teilweise recht unübersichtlichen Beziehungen der Bits untereinander und deren nicht immer offensichtlicher Zugriffsreihenfolge geht es hier darum, einen Überblick über die Funktionalität der Register zu schaffen, wie er zusammenfassend im Datenblatt fehlt. Dies soll insbesondere helfen, die mitunter fragwürdigen Funktionsweisen und Seiteneffekte einiger Bits hervorzuheben, die das Datenblatt entweder als implizit klar voraussetzt oder einfach vollkommen außer Acht läßt.

Da die Beschreibung der Register unweigerlich auch Konfigurationsmöglichkeiten für die Unterbrechungsbehandlung des AT90USB1287 aufzeigt, ist es jedoch sinnvoll, vorher noch eine kurze Einführung in das Interruptsystem zu geben.

3.6.2 Interrupt-System

3.6.2.1 Allgemeines

Der AT90USB1287 besitzt die Möglichkeit, auf bestimmte Ereignisse hin die Abarbeitung des laufenden Programmes zu unterbrechen und eine, je nach Ereignis spezifische, andere Codesequenz auszuführen. Die Hardware kann dafür für die jeweiligen Ereignisse einen Interrupt generieren, der wiederum die Ausführung eines bestimmten Codestücks zur Folge haben kann. Mögliche Unterbrechungsquellen sind zum einen externe Vorgänge, wie Zustandsänderungen an bestimmten Pins oder Datenübertragung über verschiedene Kommunikationseinheiten. Zum anderen können aber auch interne Einheiten, wie Timer, Interrupts auslösen.

Generell kann das Auftreten von Interrupts durch Manipulation des "Global Interrupt Enable"-Bits im Statusregister SREG des AVR-Controllers einbeziehungsweise ausgeschaltet werden. Alternativ zum direkten Setzen oder Löschen dieses Bits können auch die Befehle `SEI` und `CLI` genutzt werden. Ist die Interrupt-Behandlung generell aktiviert, so wird anhand einer feiner untergliederten Konfiguration festgemacht, für welche Ereignisse tatsächlich Unterbrechungen ausgelöst werden und für welche nicht. Diese Konfiguration erfolgt über das Setzen und Löschen von speziell für diesen Zweck vorgesehenen Registerbits.

Stellt die Hardware (über den Zustand der entsprechenden Bits) schließlich fest, dass ein Interrupt auszulösen ist, so beginnt sie mit der Ausführung einer Unterroutine (genannt Interrupt Service Routine, ISR), deren Startadresse (auch Interruptvektor genannt) durch die jeweilige Interruptquelle

fest vorgegeben ist[9]. Die Startadressen der Service-Routinen befinden sich dabei allesamt nebeneinander am Anfang des Programmspeichers. Ihre genauen Adressen sind dem Datenblatt zu entnehmen. Üblicherweise findet sich als erster Befehl einer ISR ein Befehl zum unbedingten Sprung in einen größeren verfügbaren Bereich des Programmspeichers, in dem dann das eigentlich vom Programmierer zur Ausführung gedachte Codesegment zu finden ist.

Verschiedene Interruptquellen lösen dabei unter Umständen die Ausführung von ein und derselben ISR aus. Diese Quellen werden dann gemeinsam als dem entprechenden Interruptvektor zugehörend bezeichnet. Eine tiefergehende Erläuterung des AVR-Interruptkonzeptes geht über den Umfang dieser Arbeit heraus, zumal der C-Compiler die Details des Systems weitestgehend vor dem Programmierer verbirgt. Der Leser sollte aber grundsätzlich mit dem Begriff des Interrupts im Kontext des AVR-Controllers vertraut sein.

3.6.2.2 USB-Interrupts

Ereignissen am beziehungsweise auf dem USB sind beim AT90USB1287 zwei verschiedene Interruptvektoren zugeordnet. Bei diesen handelt es sich zum einen um den "USB-General-Interrupt-Vektor" und zum anderen um den "USB-Endpoint/Pipe-Interrupt-Vektor".

Wie die Namen schon vermuten lassen, handelt es sich bei ersterem um einen Interrupt, der bei allgemeinen Statusänderungen des USB ausgelöst werden kann. Hierzu zählen zum Beispiel Änderungen der VBus-Spannung. Ebenfalls ausgelöst werden kann dieser Interrupt auch als Reaktion auf das Anschließen eines Gerätes beziehungsweise auch bei dessen Abziehen vom Bus.

Der zweite Interruptvektor fängt Ereignisse auf, die sich konkreter auf die Kommunikation über den Bus oder besser gesagt auf den Zustand der Kommunikationseinheit beziehen. Hierzu zählen Ereignisse wie das Melden einer fehlerhaften Übertragung oder auch das Bekanntgeben der Sendebereitschaft der Hardware.

Abbildung 3.10 zeigt die für den Hostmodus relevanten Interruptquellen des USB-General-Vektors in Analogie zu den im Datenblatt gegebenen Abbildungen. Wichtig ist die Feststellung, dass bestimmte Interrupts auch dann ausgelöst werden können, wenn die USB-Einheit gerade vom Taktsignal abgeschnitten ist, um beispielsweise die Leistungsaufnahme des AT90USB1287 zu reduzieren[10]. Umgekehrt heißt dies aber auch, dass bestimmte Signale ohne eine Taktversorgung des USB-Controllers eben nicht auftreten können, da sie

[9] Dies ist ein Unterschied zu gängigen Systemen, bei denen der Vektor selbst an einer durch die Hardware fest vorgegebenen Adresse liegt, die ISR also indirekt adressiert wird.

[10] siehe hierzu FRZCLK in USBCON, Abschnitt 3.6.3.2.

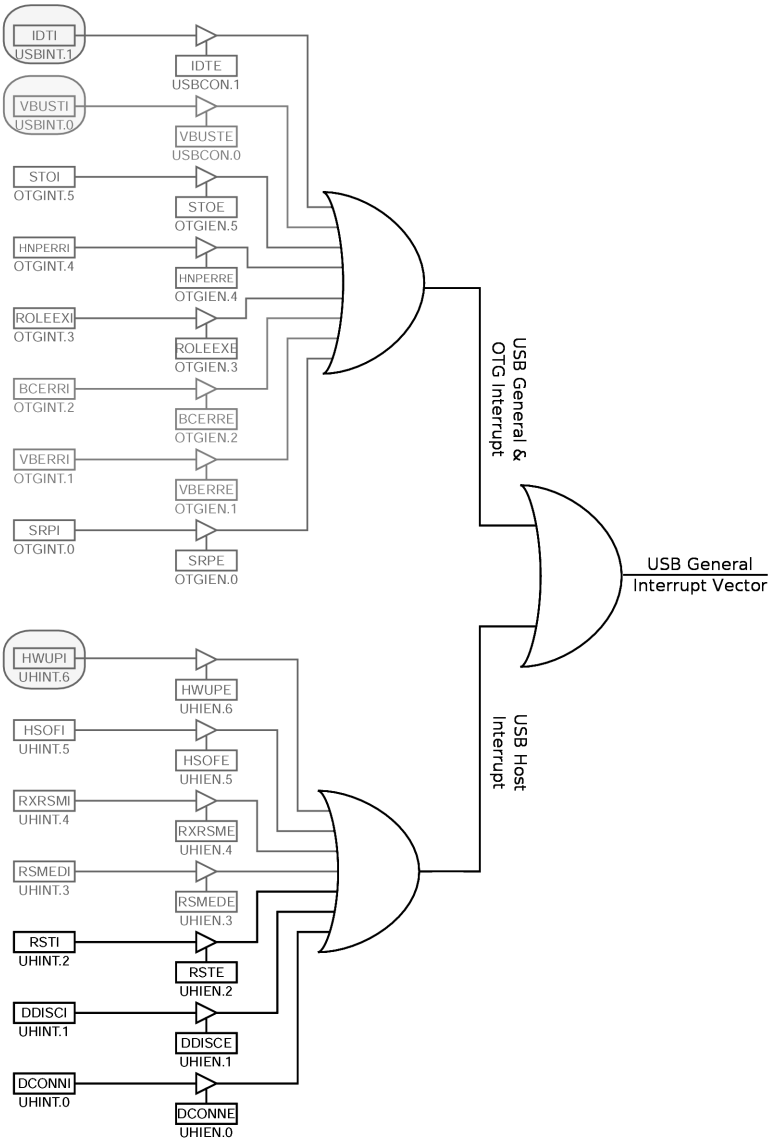


Abbildung 3.10: Für den Hostmodus relevante Interruptquellen des USB-General-Vektors. Schwach gezeichnet: Später nicht als Interruptauslöser genutzte Flags. Umrahmt: Asynchron zum Takt der USB-Einheit auslösbare Interrupts.

aktiv von diesem selbst erzeugt werden. Damit können solche Signale auch nicht genutzt werden, um die USB-Einheit aus einem Zustand niedrigerer Leistungsaufnahme “aufzuwecken”. Dies ist insbesondere bei der Erkennung des Anschlusses von Geräten zu beachten.

3.6.3 Beschreibung der USB-Register

Nach dem kurzen Einblick in das Interruptsystem des AT90USB1287 folgt nun, wie angekündigt, eine Beschreibung der relevanten Register. Dabei werden die Register, soweit wie möglich, in ihrer für die Funktion des Controllers relevanten Reihenfolge vorgestellt. In diesem ersten Teil wird nur auf die Register eingegangen, die bis zum Erkennen eines angeschlossenen Gerätes notwendig sind. Die zum Datenaustausch nötigen Register werden später in Abschnitt 3.6.5 erläutert.

Zur Aktivierung der USB-Einheit sind zunächst die beiden Register UHWCON und USBCON von Bedeutung.

3.6.3.1 UHWCON

Über UHWCON wird die generelle Konfiguration der Hardware vorgenommen, soweit sie die Anschlüsse der internen Komponenten betrifft.

Bit	7	6	5	4	3	2	1	0	
	UIMOD	UIDE		UVCONE				UVREGE	**UHWCON**
Read/Write	R/W	R/W	R	R/W	R	R	R	R/W	
Initial Value	1	0	0	0	0	0	0	0	

UVREGE – USB-Pad-Regulator Enable Ein Setzen dieses Bits schaltet den USB-Pad-Regulator, also die 3,3V Spannungsversorgung für die Signalleitungen des USB, ein. Ist das Bit Null, so ist der Regulator entsprechend abgeschaltet.

UIMOD – USB Mode Bit Wird dieses Bit gesetzt, so wird die Kontrolle der USB-Funktionalität dem USB-Device-Controller übergeben. Ist das Bit Null, so ist der Host-Controller aktiviert. Dieses Bit hat nur eine Aktivierung beziehungsweise Deaktivierung der jeweiligen Baugruppe zur Folge – um später tatsächlich mit der Software im Host- oder Device-Modus operieren zu können, ist noch die korrekte Wahl des Bits HOST in USBCON (3.6.3.2) notwendig. Ferner ist darauf zu achten, dass die Funktionalität von UIMOD durch das nachfolgend beschriebene UIDE nichtig gemacht werden kann.

UIDE – UID Pin Enable Bei gesetztem UIDE-Bit trifft die Hardware die Wahl zwischen der Aktivierung des Host- oder des Device-Controllers auf Basis der Beschaltung von Pin9 (*PE3/UID*) (siehe hierzu 3.4.2.2). Entsprechend entscheidet der Zustand dieses Bits auch darüber, ob sich Pin9 als I/O-Pin für PortE nutzen lässt oder durch den USB belegt ist. Sobald die Kontrolle über die Baugruppenwahl dem *UID*-Pin übergeben wurde, ist der

Zustand des vorangehend beschriebenen Bits UIMOD irrelevant. Die Hardware hält sich dann auf jeden Fall an die Vorgabe von *UID*. Die Wahl für UIDE ist zu treffen, solange sich die USB-Einheit im deaktivierten Zustand befindet (siehe hierzu USBE in Abschnitt 3.6.3.2).

UVCONE – UVCON Pin Enable Über dieses Bit ist die Funktion von Pin2 des AT90USB1287 zwischen *PE7* und *UVCON* wählbar. Solange das Bit mit Null belegt ist, dient auch dieser Pin als normaler I/O-Pin für PortE. Es ist darauf zu achten, dass das Bit nur bei eingeschalteter USB-Einheit gesetzt wird. Ferner sei hier zur Klarstellung erwähnt, dass dieses Bit nur das grundsätzliche Umschalten zwischen den Funktionalitäten von Pin2 übernimmt – die Steuerung der nach Setzen des Bits auf dem Pin anliegenden Signale ist erst durch weitere Bits möglich. Namentlich sind dies: VBUSHWC, VBUSREQ und VBUSRQC in Register OTGCON (siehe 3.6.3.5).

3.6.3.2 USBCON

Mittels des Registers USBCON werden weitere Einstellungen vorgenommen, die den generellen Betrieb des USB-Controllers betreffen. Unter anderem kann so die Funktionalität der durch die Angaben in Register UHWCON selektierten Einheiten beeinflusst werden. Die wichtigste Möglichkeit ist hierbei sicherlich das Ein- und Ausschalten der USB-Einheit (im Datenblatt "USB macro" genannt). Zusätzlich bietet das Register über die Bits VBUSTE und IDTE auch die Möglichkeit, auf den USB betreffenden Teil des Interruptsystems Einfluss zu nehmen.

Bit	7	6	5	4	3	2	1	0	
	USBE	HOST	FRZCLK	OTGPADE			IDTE	VBUSTE	**USBCON**
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

USBE – USB macro Enable Die durch UHWCON ausgewählte USB-Einheit (also Host- oder Device-Controller) beginnt zu arbeiten, sobald dieses Bit gesetzt wird. Bei bereits aktiviertem USB-Interface ist ein Rücksetzen der gesamten Einheit jederzeit möglich, indem USBE zunächst auf Null und dann wieder auf Eins gesetzt wird. Wird die Einheit ausgeschaltet, so wird auch ein eventuell vorher auf High- Pegel gebrachter *UVCON*-Pin zurückgesetzt. Dieses Verhalten ist bei einer schaltbaren Spannungsversorgung für den VBus zu berücksichtigen.

HOST – Host Bit Während über das vorangehend beschriebene UHWCON zwischen dem Betrieb der Host- oder der Device-Hardware gewählt

werden kann, regelt das HOST-Bit den Zugriff auf die zugehörigen Register. Der AT90USB1287 besitzt laut Datenblatt gemultiplexte Register für Host- und Devicemodus. Ist das HOST-Bit gesetzt, so greift die Software auf die Register der Host-Einheit zu, bei gelöschtem Bit sind entsprechend die Device-Register verfügbar.

Tatsache ist jedoch, dass Device- und Hostregistern verschiedene Adressen zugeordnet sind, wobei einige dieser Adressen (beispielsweise bei UPDATX und UEDATX) auch noch – unabhängig vom Zustand des HOST-Bits – auf dasselbe (physische) Register verweisen.

FRZCLK – Freeze USB Clock Bit Bei gesetztem FRZCLK-Bit wird die USB-Einheit von der Taktzufuhr abgeschnitten. Dies kann den Energieverbrauch senken. Zu beachten ist, dass dem USB-Controller für den ordnungsgemäßen Betrieb überhaupt erst einmal ein geeigneter Takt zur Verfügung gestellt werden muss. Dies geschieht über korrekte Konfiguration von PLLCSR (siehe 3.6.3.3).

FRZCLK ist ein Bit, welches die Hardware im Falle eines Fehlers häufig eigenständig setzt. Bei der Reaktion auf etwaige aufgetretene Fehler sollte darauf geachtet werden, die USB-Einheit danach wieder in den laufenden Zustand zu versetzen, indem FRZCLK gelöscht wird.

OTGPAD – OTG Pad Enable Dieses Bit aktiviert das auf dem AT90USB1287 befindliche OTG-Pad. Dies ist in nahezu allen denkbaren Szenarien nötig, da zum einen ein schaltbarer VBus nur so kontrollierbar und zum anderen auch das Erkennen von neu angeschlossenen Geräten nur mit gesetztem Bit möglich ist. Sowohl die Funktion des Bits VBUS in USBSTA (siehe 3.6.3.6) als auch die von VBUSTI in USBINT (siehe 3.6.3.7) hängen von OTGPAD ab.

Auch wenn auf die Funktionalität von VBUSTI aus später noch erläuterten Gründen ohnehin besser verzichtet wird, erweist sich ein deaktiviertes OTG-Pad dennoch als fatal: Durch die fehlende Statuserkennung für VBus kann nicht auf Fehler in der Versorgungsspannung der angeschlossenen Geräte reagiert werden. Zusätzlich funktioniert auch das Feststellen eines korrekten oder fehlgeschlagenen Verbindungsversuchs zum Gerät nicht[11]. Das Datenblatt vermittelt insbesondere letzteres nicht, da es irreführenderweise das Bit OTGPAD ausschließlich mit der Statuserfassung für VBus in Verbindung bringt.

[11] Die Bits BCERRI und VBERRI in OTGINT bleiben genauso funktionslos wie auch DCONNI und DDISCI in UHINT.

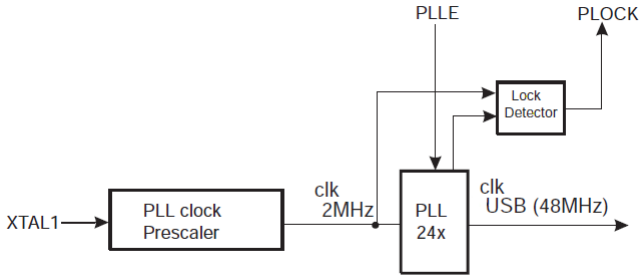


Abbildung 3.11: Blockbild der PLL-Schaltung für USB.

IDTE – ID Transition Interrupt Enable Bit Wird Pin9 als *UID*-Pin genutzt (siehe 3.6.3.1: UIDE in UHWCON), so löst ein Pegelwechsel an diesem Pin einen Interrupt aus, sofern IDTE gesetzt wurde.

VBUSTE – VBUS Transition Interrupt Enable Bit Ist dieses Bit gesetzt, so löst ein Pegelwechsel an *VBUS* einen Interrupt aus (vorausgesetzt OTGPAD ist gesetzt). Vorgreifend sei gesagt, dass dies immer bei gesetztem VBUSTI-Bit in USBSTA (siehe 3.6.3.6) der Fall ist.

Vorsicht ist bei der Nutzung des VBUSTE-Bits angesagt, denn auch wenn das Datenblatt dieses Bit lediglich mit dem Auftreten eines Interrupts bei VBUSTI assoziiert, so sind bei gelöschtem VBUSTE gleich auch noch eine Reihe anderer Interrupts abgeschaltet. Im Versuch auffällig waren vor allem das fehlende Auftreten eines Interrupts bei fehlerhafter VBus-Spannung wie auch bei Fehlschlagen eines Geräteanschlusses. Dies ist insbesondere bei Verwendung der Bits BCERRI und VBERRI in OTGINT (3.6.3.8) zu beachten.

3.6.3.3 PLLCSR

Auch wenn dieses Register nicht direkt zu den USB-Registern gehört, so kommt ihm doch eine große Bedeutung bezüglich der Funktionalität zu. Wie weiter oben schon angedeutet, erzeugt der AT90USB1287 intern die für den USB benötigte Taktfrequenz von 48MHz mittels einer digitalen PLL. Abbildung 3.11 zeigt das Blockbild der dafür verwendeten Logik, wie es ähnlich auch im Datenblatt zu finden ist. Die verbaute PLL erzeugt an ihrem Ausgang eine Frequenz, die um den Faktor 24 höher ist als ihre Eingangsfrequenz. Um die geforderten 48MHz zu generieren, ist die PLL daher mit einer Eingangsfrequenz von 2MHz zu versorgen. Diese kann aus dem XTAL-Takt über einen konfigurierbaren Vorteiler gewonnen werden. Hierfür ist allerdings der

Betrieb mit einem 8MHz oder einem 16MHz Quarz beziehungsweise Oszillator notwendig. Die benötigten Einstellungen werden wie folgt vorgenommen.

Bit	7	6	5	4	3	2	1	0	
				PLLP2	PLLP1	PLLP0	PLLE	PLOCK	**PLLCSR**
Read/Write	R	R	R	R/W	R/W	R/W	R/W	R	
Initial Value	0	0	0	0	0	0	0	0	

PLLP2:0 Mittels dieser Bits wird der Vorteiler konfiguriert. Beschaltet man die PLLP-Bits bei Betrieb mit 16MHz mit der im Datenblatt angegebenen Kombination von 110, so läßt sich am Oszilloskop eine signifikante Abweichung von der geforderten Taktfrequenz feststellen. Die Dauer eines Bits beträgt dann – gemessen im Fullspeed-Modus – etwa 104ns statt der geforderten $83,\overline{3}$ns.

Als Randbemerkung sei hier angegeben, dass zwei auf diese Weise gleichermaßen fehlkonfigurierte Mikrocontroller, wie sie den Autoren für diese Arbeit zur Verfügung standen, natürlich problemlos untereinander kommunizieren konnten. Nur eine Verbindung mit der restlichen USB-Welt ist dabei selbstverständlich ausgeschlossen. Im Versuch ließ sich der im 16MHz-Betrieb benötigte Wert für PLLP ermitteln.

Tabelle 3.2 zeigt die korrekten Belegungen für PLLP2:0, welche eine Kommunikation nach dem USB-Standard ermöglichen.

external XTAL (MHz)	**Divisor**	**PLLP2**	**PLLP1**	**PLLP0**
8	4	0	1	1
16	8	1	0	1

Tabelle 3.2: Valide Werte für PLLP2:0 zur Erzeugung von 48MHz.

PLLE Ein Setzen von PLLE aktiviert die PLL. Entsprechend führt ein Löschen des Bits zum Anhalten der Takterzeugung.

PLOCK Hat sich die Ausgangsfrequenz der PLL stabilisiert, so setzt die Hardware dieses Bit. Die Stabilisierung nimmt typischerweise etwa 100ms in Anspruch. Soll dieses Bit zurück gesetzt werden, so müssen sowohl das Bit PLLE als auch die Bits PLLP2:0 gelöscht werden. Ein Zurücksetzen ist jedoch im normalen Betrieb nicht notwendig.

3.6.3.4 UDCON

Um den Host-Controller tatsächlich in einen Status zu versetzen, in dem auf den Bus zugegriffen werden kann, ist noch ein weiteres Bit notwendig. Dies ist das DETACH Bit, welches sich jedoch, wenig intuitiv, in einem der Device-Register – namentlich UDCON – befindet. Die anderen Bits dieses Registers werden für den Hostmodus nicht genutzt.

Bit	7	6	5	4	3	2	1	0	
						LSM	RMWKUP	DETACH	**UDCON**
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	1	

DETACH – Detach Bit Ein Löschen dieses Bits schaltet die durch den USB-Pad-Regulator generierte Spannung auf die D+ beziehungsweise D− Leitung durch. Die Beschaltung ist dabei abhängig von der Frage, ob sich der Controller derzeit im Full- oder im Lowspeed-Modus befindet.

Im Hostmodus schaltet die Hardware die internen PullDown-Widerstände der D+ und D− Pins zu und stellt den jeweils für das angeschlossene USB-Gerät benötigten Modus eigenständig fest. (Zur Kontrolle der momentanen Einstellung siehe auch Bit SPEED in Abschnitt 3.6.3.6.)

3.6.3.5 OTGCON

Unter der Annahme, dass der Controller mittels der bis hierhin beschriebenen Register in einen einsatzfähigen Zustand gebracht wurde, kann nun mit der Kontrolle des Bus-Zustandes begonnen werden. Für den Fall, dass die Spannungsversorgung des VBus mittels des Pins *UVCON* gesteuert werden soll, muss dieser vor dem Anschluss von USB-Geräten in einen Zustand versetzt werden, in dem der Bus mit Spannung versorgt ist. Da besagter Pin in dem vorgestellten Experimentierboard in der Tat die Kontrolle über die Versorgungsspannung des Busses übernimmt, soll nun betrachtet werden, wie der Zustand des Pins kontrolliert werden kann. Hierzu sind die drei im Folgenden beschriebenen Bits notwendig. Die hier nicht erläuterten Bits des Registers OTGCON beziehen sich explizit auf im Weiteren ungenutzte Fähigkeiten des On-the-Go-Supplements ([Speh]), ihre Funktion kann [Dat], Seite 265, entnommen werden.

Bit	7	6	5	4	3	2	1	0	
			HNPREQ	SRPREQ	SRPSEL	VBUSHWC	VBUSREQ	VBUSRQC	**OTGCON**
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

VBUSREQ – VBus Request Bit Wird dieses Bit auf Eins geschrieben und ist gleichzeitig UVCONE (3.6.3.1) gesetzt, so wird der *UVCON*-Pin auf High-Pegel geschaltet. Der Erfolg der Aktion kann über das VBUS-Bit (3.6.3.6) kontrolliert werden. Ein eingeschaltetes OTG-Pad ist hierfür Voraussetzung (3.6.3.2). Interessant ist, dass dieses Bit auch gesetzt sein muss, wenn der UVCON-Pin gar nicht zum Einsatz kommen soll, also das Bit UVCONE nicht gesetzt ist. Anderenfalls war im Versuch weder ein Feststellen des VBus-Status, noch ein Erkennen des Ansteckens von Geräten mittels DCONNI (siehe 3.6.3.9) möglich.

Das VBUSREQ-Bit wird von der Hardware automatisch gelöscht, sobald VBUSRQC gesetzt wird.

VBUSRQC – VBus Request Clear Bit Dieses Bit verhält sich analog zu VBUSREQ, allerdings führt ein Setzen hier zu einer Beschaltung des Pins *UVCON* mit Low-Pegel. Dieses Bit wird sofort nach dem Setzen von der Hardware wieder gelöscht.

VBUSHWC – VBus Hardware Control Bit Ist das VBUSHWC Bit gesetzt, so hat die Hardware laut Datenblatt keine Möglichkeit mehr, den Zustand des *UVCON*-Pins aus eigener Initiative zu verändern. Standardmäßig ist ihr dies erlaubt, so dass der VBus beispielsweise im Fehlerfall automatisch von der Spannungsversorgung abgeschnitten werden kann. Tatsächlich passiert dies aber auch bei gesetztem Bit, so dass der Verwendungszweck des Bits unklar ist. Jedenfalls lässt sich der Pin *UVCONE* sowohl bei gesetztem als auch bei gelöschtem VBUSHWC von der Software kontrollieren.

Mit zugeschalteter VBus-Spannung ist es nun sinnvoll, die Möglichkeiten zur Feststellung und Manipulation des Zustandes des USB-Controllers näher zu betrachten. Hierzu zählen das Erkennen der grundsätzlichen elektrischen Gegebenheiten wie das Überprüfen der VBus-Spannung oder das Unterscheiden zwischen Full- und Lowspeed-Modus eines angeschlossenen Gerätes[12]. In diese Kategorie können auch das Feststellen des Ansschließens und Abziehens von Geräten eingeordnet werden, da dies ebenfalls über den Zustand von D+ und D− erkannt wird.

[12] Zur Erinnerung: Die Wahl findet ausschließlich auf Basis der an D+ und D− festgestellten Spannung statt.

3.6.3.6 USBSTA

Das Register USBSTA enthält allgemeine Informationen über den Status des USB-Anschlusses.

Bit	7	6	5	4	3	2	1	0	
					SPEED		ID	VBUS	**USBSTA**
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	1	0	1	0	

VBUS – VBus Status Flag Dieses Bit gibt den momentanen Zustand am *VBUS*-Pin und damit auch den Status der Spannungsversorgung des VBus wieder. Hierbei bedeutet ein gesetztes Bit einen ordnungsgemäß mit Spannung versorgten VBus. Ein gelöschtes Bit gibt einen abgeschalteten beziehungsweise in der Spannung eingebrochenen VBus an. Zur Erinnerung: Diese Bit besitzt nur dann Aussagekraft, wenn das OTG-Pad eingeschaltet wurde (siehe 3.6.3.1). Verknüpft ist VBUS ebenfalls mit dem Bit VBUSTI-Bit, welches in 3.6.3.7 beschrieben ist.

ID – *UID*-Pin Flag Das ID-Bit spiegelt den Zustand des *UID*-Pins wieder, sofern UIDE in UHWCON gesetzt ist. Das Bit bleibt in dieser Arbeit ungenutzt.

SPEED – Speed Status Flag Über dieses Bit kann festgestellt werden, in welchem Geschwindigkeitsmodus sich der Controller befindet. Eine Abfrage sollte nur im Hostmodus erfolgen, im Devicemodus ist der Zustand des Bits indeterminiert. Die Hardware setzt dieses Bit, sobald die USB-Einheit sich in den Fullspeed-Modus begibt. Entsprechend bedeutet ein gelöschtes Bit, dass diese momentan im Lowspeed-Modus operiert.

3.6.3.7 USBINT

In gewisser Weise direkt mit dem vorangehend beschrieben USBSTA-Register verwandt ist das Register USBINT. In seinen Bits wird festgehalten, ob sich einmal eine Veränderung an VBUS oder ID vollzogen hat.

Bit	7	6	5	4	3	2	1	0	
							IDTI	VBUSTI	**USBINT**
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

IDTI – ID Transition Interrupt Flag Das IDTI-Bit wird von der Hardware gesetzt, sobald ein Kippen des Bits ID festgestellt wird. In Verbindung mit IDTE (siehe USBCON, Abschnitt 3.6.3.2) kann durch IDTI auch ein Interrupt ausgelöst werden. Dann ist darauf zu achten, dass das Bit entsprechend durch die Software zurückgesetzt wird, um den Interrupt zu bestätigen. Ohne diesen "Handshake" wird der Interrupt fortlaufend ausgelöst werden.

VBUSTI – VBUS Transition Interrupt Flag Stellt die Hardware eine Veränderung des VBUS-Bits und damit eine Veränderung der VBus-Spannung fest, so wird VBUSTI gesetzt. Auch durch VBUSTI kann – diesmal gesteuert durch VBUSTE in USBCON – ein Interrupt ausgelöst werden. Auch VBUSTI sollte durch die Software zurückgesetzt werden, nachdem diese auf den Interrupt reagiert hat.

3.6.3.8 OTGINT / OTGIEN

Weiter im Rahmen dieser Arbeit von Interesse sind noch einige Bits in den Registern OTGINT und ihre entsprechenden Interrupt-Enable-Bits in OTGIEN. Hiermit kann die Software auf Verbindungsfehler (auf physikalischer Ebene) sowie Fehler der VBus-Spannungsversorgung reagieren.

Bit	7	6	5	4	3	2	1	0	
			STOI	HNPERRI	ROLEEXI	BCERRI	VBERRI	SRPI	**OTGINT**
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
			STOE	HNPERRE	ROLEEXE	BCERRE	VBERRE	SRPE	**OTGIEN**
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

VBERRI – VBus Error Interrupt Flag Ein gesetztes VBERRI deutet auf einen Zusammenbruch der VBus-Spannung während des laufenden Betriebs von Geräten beziehungsweise während der Verbindungsphase zu diesen hin. In diesem Fall schaltet die Hardware den VBus aus (so ihr dies durch die Schaltung möglich ist). Die USB-Einheit sollte nach einem solchen Fehler reinitialisiert werden, um deren Funktionsfähigkeit wieder herzustellen.

VBERRE – VBus Error Interrupt Enable Flag Mit VBERRE wird das Auslösen eines Interrupts bei Setzen von VBERRI eingeschaltet. VBERRI sollte in diesem Fall nach Behandlung des Interrupts gelöscht werden.

BCERRI – B-Connection Error Interrupt Flag Die Hardware setzt das BCERRI Flag, falls sie beim Versuch, eine Verbindung zu einem USB-Gerät festzustellen, kein Gerät vorfindet. Denkt man an die vorangehend beschriebenen Sachverhalte, so ist dies dann der Fall, wenn weder D+ noch D− eine von GND verschiedene Beschaltung haben.

Der Hostmodus sorgt ja dafür, dass die internen PullDown-Widerstände dieser Pins zugeschaltet werden und die Erkennung der Geschwindigkeit (Full/Lowspeed) von angeschlossenen Geräten auf Basis der High-Pegel-Beschaltung von D+ beziehungsweise D− verläuft.

Nähere Informationen über die Verbindungsfestellung zu Geräten finden sich unter 2.3.3.

BCERRE – B-Connection Error Interrupt Enable Flag BCEERE aktiviert das Auftreten eines Interrupts bei Änderung des BCERRI-Bits. BCERRI ist dann als Bestätigung für die Behandlung des Interrupts zu löschen.

Wichtig: Bei den hier aufgeführten Bits ist darauf zu achten, dass ihre grundsätzliche Funktionalität vom Aktivieren des OTG-Pads abhängt (OTGPADE, siehe 3.6.3.1). Bei den Interrupt-Enable-Bits kommt ferner hinzu, dass ein Setzen dieser Bits allein nicht ausreicht, um die Interruptverarbeitung einzuschalten. Um tatsächlich einen Interrupt zu erzeugen, musste bei den im Versuch genutzen Geräten zusätzlich noch VBUSTE gesetzt sein, was im Datenblatt nicht erwähnt wird.

Bis jetzt hat sich die Registerbeschreibung ausschließlich im Rahmen der USB-General-Register bewegt. Die in dieser Hinsicht wichtigen Register sind damit insoweit erläutert, wie dies für das weitere Verständnis des Vorgehens notwendig ist. Um sinnvoll im Hostmodus operieren zu können sind – wie man sich leicht denken kann – noch einige Einstellungen in der Gruppe der USB-Host-Register nötig. Mit den drei nachfolgend beschriebenen Registern UHCON, UHINT und UHIEN lässt sich ein Zustand herstellen, in dem ein Gerät beim Anschließen erkannt und zur Datenübertragung bereit gemacht werden kann. Auch das Abziehen von Geräten wird sich dann erkennen lassen.

3.6.3.9 UHINT/UHIEN

Diese Register enthalten Bits, die über allgemeine für den Host relevante Ereignisse Aufschluss geben. Hierzu zählen grundlegende Dinge wie das Erfassen von An- und Absteckvorgängen von Geräten, aber auch speziellere USB-

Events, wie Resume- und Wake-Up-Ereignisse. Solche speziellen Möglichkeiten von USB werden später nicht genutzt, daher bleiben die entsprechenden Bits an dieser Stelle in der Beschreibung unberücksichtigt. Ihre genaue Funktionsweise kann [Dat], Seite 301f. entnommen werden.

Bit	7	6	5	4	3	2	1	0	
		HWUPI	HSOFI	RXRSMI	RSMEDI	RSTI	DDISCI	DCONNI	**UHINT**
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
		HWUPE	HSOFE	RXRSME	RSMEDE	RSTE	DDISCE	DCONNE	**UHIEN**
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

DCONNI – Device Connection Interrupt Die Hardware setzt das DCONNI-Bit, sobald sie eine valide Verbindung zu einem Gerät entdeckt. Diese wird als gegeben gesehen, wenn D+ und D− eine für den Low- beziehungsweise Fullspeed-Modus gültige Beschaltung eingenommen haben. Dies ist das Zeichen, dass mit der Initialisierung des Geräts begonnen werden kann. DCONNI zählt, wie schon mehrfach erwähnt, zu den von OTGPADE abhängigen Bits und erfordert einen aktiven, mit Takt versorgten USB-Host-Controller.

DCONNE – Device Connection Interrupt Enable Ist DCONNE gesetzt, so löst ein gesetztes DCONNI-Bit einen Interrupt aus. DCONNI sollte in diesem Fall nach dem Behandeln des Interrupts gelöscht werden.

DDISCI – Device Disconnection Interrupt Als Gegenstück zu DCONNI wird DDISCI von der Hardware gesetzt, sobald das Abziehen eines Geräts festgestellt wird. Es gelten die gleichen Randbedingungen wie bei DCONNI.

DDISCE – Device Disconnection Interrupt Enable Ein gesetztes Bit sorgt für das Auslösen eines Interrupts bei gesetztem DDISCI. Auch DDISCI sollte nach Behandlung der Unterbrechung gelöscht werden, um einen Wiederaufruf zu vermeiden.

HSOFI – Host Start Of Frame Interrupt Hat der Controller ein Start Of Frame Packet (SOF) auf den Bus gebracht, so setzt die Hardware dieses Bit. Für das Generieren von SOFs ist das später beschriebene Bit SOFEN aus UHCON erforderlich.

HSOFE – Host Start Of Frame Interrupt Enable Ein Setzen dieses Bits aktiviert die Interruptbehandlung für HSOFI. Auch dieser Interrupt sollte durch die Software mittels Löschen bestätigt werden.

RSTI – Device Reset Interrupt RSTI wird gesetzt, sobald die Hardware den Reset des angeschlossenen Gerätes vollzogen hat. Eine Aufforderung zum Reset muss vorher über RESET aus UHCON geschehen.

RSTE – Device Reset Interrupt Enable Ein gesetztes RSTE lässt bei Setzen von RSTI einen Interrupt auftreten. RSTI ist, wie die entsprechenden Bits der anderen vorgestellten Interrupts auch, nach Behandlung zu löschen.

3.6.3.10 UHCON

Das Register UHCON bietet, als einziges der bisher genannten, die Möglichkeit, den Bus auf logischer Ebene zu beeinflussen.

Bit	7	6	5	4	3	2	1	0	
						RESUME	RESET	SOFEN	**UHCON**
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

SOFEN – Start Of Frame Generation Enable Sobald dieses Bit gesetzt wird, beginnt der Host-Controller, SOF-Pakete oder Keep-Alive-Messages auf den Bus zu bringen. Die Wahl zwischen SOF und Keep-Alive hängt dabei davon ab, ob vorangehend die Verbindung zu einem Full- oder Low-speed-Gerät festgestellt wurde. Sind SOF-Pakete zu senden, so bezieht der Controller deren Sequenznummer aus den beiden Registern UHFNUMH und UHFNUML. Diese bilden zusammen ein 16Bit-Register (allerdings mit maximal elfbittigem Inhalt), dessen Wert die jeweils folgende Sequenznummer angibt. Der Host-Controller stoppt das Senden von SOFs oder Keep-Alives, sobald SOFEN wieder gelöscht wird.

RESET – Send USB Reset RESET veranlasst den Controller, ein Reset-Signal auf den Bus zu legen. Das Bit wird automatisch zurückgesetzt, wenn das Signal abgesetzt wurde.

3.6.4 Übersicht über das Verhalten des Host-Controllers

Nach dieser Menge von Konfigurations-Bits soll nun erst einmal das Verhalten der Hardware – soweit es die Konfiguration der USB-Einheit sowie

das Erkennen von An- und Abstecken eines Gerätes betrifft – erläutert werden. Dieser Abschnitt bringt etwas mehr Klarheit über die Arbeitsweise des Controllers.

3.6.4.1 Zustand nach dem Einschalten

Unmittelbar nach dem Einschalten befindet sich der gesamte Host-Controller im deaktivierten Zustand. Auch die Taktgenerierung ist unkonfiguriert.

3.6.4.2 Empfohlener Betriebszyklus

PLL konfigurieren Bevor etwas weiteres getan wird, sollte die Taktversorgung der USB-Einheit konfiguriert werden. Hierfür wird der korrekte Wert aus Tabelle 3.2 nach PLLCSR gebracht, anschließend PLLE gesetzt und auf das Einschwingen der PLL gewartet. Sobald die Hardware das PLOCK-Bit gesetzt hat, ist die Taktversorgung einsatzbereit.

Bereitschaft der USB-Einheit für den Hostmodus herstellen Der USB-Pad-Regulator ist in Betrieb zu nehmen. Dann wird die Wahl über den Betrieb des *UID*-Pins beziehungsweise über Host- oder Devicemodus getroffen. Hierfür sind UIMOD und UIDE zu nutzen. Danach ist die USB-Einheit zu aktivieren und anschließend die Spannungsversorgung für D+ und D− sicherzustellen. Hierfür können nacheinander die Bits UVREGE, USBE und DETACH genutzt werden. Anschließend kann der Zugriff auf die Hostregister durch Setzen des Bits HOST ermöglicht werden.

OTG-PAD und VBus-Konfiguration Aus den bei der Registerbeschreibung genannten Gründen muss das OTG-Pad zugeschaltet werden. Hierzu muss OTGPADE gesetzt werden. Soll der VBus gesteuert werden, so ist entsprechend UVCONE zu nutzen. VBUSREQ ist auf jeden Fall auf Eins zu schreiben, um später das Auslösen von DCONNI und die Kontrolle des VBus zu ermöglichen.

Starten des Takts und warten auf Verbindung Mittels FRZCLK kann die Arbeit des Controllers nun gestartet werden. Sobald der Takt läuft, wird der Host versuchen, eine Verbindung zu einem Gerät festzustellen. Gelingt ihm dies nicht innerhalb eines gewissen Zeitintervalls, dann setzt die Hardware BCERRI. Wird jedoch eine gültige Geräteverbindung gefunden, so setzt die Hardware DCONNI.

Senden von SOFs und RESET Wurde ein Gerät erfolgreich erkannt, so kann mit dem Senden von SOFs oder Keep-Alives begonnen werden. Außerdem ist nun das Gerät mit RESET zurückzusetzen, damit mit der Kommunikation begonnen werden kann.

Kommunikation und Reaktion auf Abstecken Über die noch zu beschreibenden Register kann dann mit dem Gerät kommuniziert werden. Wird das Gerät abgezogen, so wird DDISCI gesetzt. Daraufhin wird das Senden von SOFs beziehungsweise Keep-Alives eingestellt.

Auf Probleme mit der Versorgungsspannung ist zu jeder Zeit über VBERRI zu reagieren.

3.6.5 Beschreibung der USB-Register (Fortsetzung)

In der Registerliste fehlen noch diejenigen Register, welche zur Kommunikation mit einem USB-Gerät notwendig sind. Darunter fallen zum einen Register für die Speicherverwaltung der zu den verschiedenen Pipes gehörenden Daten. Zum anderen sind natürlich auch solche Register zu betrachten, die grundsätzlich das Absenden und Empfangen von Daten sowie sonstige das USB-Protokoll betreffende Einstellungen ermöglichen.

3.6.5.1 Allgemeines zur Pipe- und Speicherverwaltung

Bevor mit einem Gerät kommuniziert werden kann, muss ein geeignet großer Speicherbereich für die Daten der jeweiligen Pipe reserviert werden. Dieser Bereich steht der fraglichen Pipe dann im DPRAM zur Verfügung. Wichtig dabei ist, den Speicher für die bis zu sieben im AT90USB1287 möglichen Pipes in aufsteigender Reihenfolge zu reservieren. Beachtet man dies nicht oder konfiguriert im Nachhinein die Größe eines bereits reservierten Bereichs um, so können durch die interne Speicherverwaltung des AT90USB1287 leicht Konflikte entstehen. Verwirrend hierbei ist vor allem das Verhalten des Controllers, wenn Speicher für eine Pipe dealloziert wird. Bei der Deallokation des Speichers für Pipe P_i wird der für P_{i+1} vorgesehene Speicherbereich auf den nun freigewordenen (niedriger liegenden) Bereich verschoben. Dabei gehen eventuell für P_{i+1} im Speicher vorhandene Daten verloren. Interessanterweise bleibt allerdings der Speicher für P_{i+2} von diesem Vorgang vollkommen unberührt – In der Folge ergibt sich ein nicht zugewiesener Bereich des DPRAMs nun zwischen den Bereichen für P_{i+1} und P_{i+2}.

Wird eine Pipe P_i hingegen nachträglich – also nach der Speicherreservierung für P_{i+1} – dahingehend umkonfiguriert, dass ein größerer Speicherbedarf

PipeNummer	max. Größe	Default-Control-Pipe	Zweibank-Modus
0	64Byte	√	–
1	256Byte	–	√
2	64Byte	–	√
3	64Byte	–	√
4	64Byte	–	√
5	64Byte	–	√
6	64Byte	–	√

Tabelle 3.3: Übersicht über die Fähigkeiten der einzelnen Pipes. Die maximal verfügbare Größe des DPRAMs von 832 Byte darf bei der Allokation nicht überschritten werden.

für sie entsteht, so kommt es zu einer Bereichsüberdeckung der Speicherblöcke für P_{i+1} und P_{i+2}. Dabei gehen die Daten von P_{i+1} erneut verloren. Die Hardware ist nicht in der Lage, solche Fehlkonfigurationen zu erkennen und meldet trotz allem eine korrekte Speicherallokation.

Aufgrund des hier beschrieben Verhaltens muss die Speicherverwaltung mit großer Vorsicht gehandhabt werden. Ein gutes Vorgehen ist es, bereits bei der Initialisierung feste Speicherbereiche für alle Pipes vorzusehen.

Ferner wird der Speicherbedarf durch die Frage beeinflusst, ob die Pipe im Ein- oder Zweibankmodus betrieben wird. Der Zweibankmodus erlaubt es der Software, ein Datenpaket in die Pipe zu schreiben, während die Hardware noch mit dem Senden des vorangehenden Paketes befasst ist. Analoges gilt für den Empfang von Daten: Die Hardware nimmt schon ein neues Paket vom Bus entgegen, während die Software noch mit der Auswertung des vorangehenden beschäftigt ist.

Die zur Verfügung stehenden Pipes unterscheiden sich außerdem in ihren Fähigkeiten bezüglich des verwaltbaren Speichers und der einstellbaren Pipe-Typen[13]. So ist Pipe0 die einzige Pipe, die als Default Control Pipe dienen kann. Verbindungen zu Default Control Endpoints von Geräten können also ausschließlich über Pipe0 vorgenommen werden. Die Eigenschaften der verschiedenen Pipes sind in Tabelle 3.3 aufgeführt. Die hier beschriebenen Eigenschaften gelten analog im Devicemodus für die durch den AT90USB1287 zur Verfügung gestellten Endpoints.

Die folgenden Register sind für die grundsätzliche Konfiguration der Kommunikation und der Pipes von Bedeutung.

[13] siehe 2.1.3

3.6.5.2 UPNUM

Über das UPNUM-Register wird die aktuelle (Hardware-)Pipe gewählt. Die im Weiteren beschriebenen auf X endenden Register beziehen sich dann jeweils auf die mit UPNUM selektierte Pipe.

Bit	7	6	5	4	3	2	1	0	
						PNUM2	PNUM1	PNUM0	**UPNUM**
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PNUM2:0 Diese dreibittige Zahl entspricht der zu wählenden Pipenummer.

3.6.5.3 UPCFG1X

Das UPCFG1X-Register stellt die Möglichkeit der Speicherkonfiguration für die jeweils aktuelle Pipe zur Verfügung. Pipegröße, Ein-/Zweibankmodus und die Allokation und Deallokation von Speicher werden hiermit gesteuert.

Bit	7	6	5	4	3	2	1	0	
		PSIZE2	PSIZE1	PSIZE0	PBK1	PBK0	ALLOC		**UPCFG1X**
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R	
Initial Value	0	0	0	0	0	0	0	0	

PSIZE2:0 – Pipe Size Configuration Mit Hilfe dieser Bits wird das Fassungsvermögen der Pipe eingestellt. Die Pipegröße ergibt sich dabei nach der Formel

$$Size_{P_i} = 2^{PSIZE+3}$$

Theoretisch sind also Pipegrößen von bis zu 1024 Byte einstellbar. Es ist jedoch darauf zu achten, dass die jeweiligen Fähigkeiten der Pipe, wie sie Tabelle 3.3 bereits zu entnehmen waren, nicht überschritten werden. Die maximale Wahl ist dementsprechend für Pipe0 beispielsweise $PSIZE = 3_{10} = 011_2$, was zu einem Fassungsvermögen von $Size_{P_0} = 2^6 = 64$ Byte pro Paket führen würde.

PBK1:0 – Bank Number Selection Diese Zahl schreibt die Anzahl der zu verwendenden Bänke für die aktuelle Pipe vor. Auch dabei ist auf Einschränkungen aus Tabelle 3.3 zu achten, was hier nur bedeutet, dass Pipe0 stehts im Einbankmodus zu betreiben ist. Der folgenden Tabelle sind die Werte für PBK zu entnehmen:

PBK1:0	00	01	10	11
Bankzahl	1	2	invalid	invalid

Tabelle 3.4: Konfiguration der Bankzahl über PBK1:0.

ALLOC – Allocate Pipe Memory Sobald das ALLOC-Bit gesetzt wird, reserviert – oder besser gesagt: bestimmt – die Hardware den Speicherplatz für mit dieser Pipe zu verarbeitende Daten. Ein Löschen des ALLOC-Bits gibt den Speicher entsprechend wieder frei. Dabei gelten alle unter 3.6.5.1 gemachten Aussagen.

Welcher Pipe wieviel Speicher und welche Bankzahl zugedacht wird, ist abhängig vom jeweiligen Anwendungsfall. Sollten keine speziellen Anforderungen vorliegen, so bietet es sich an, eine Konfiguration aller Pipes auf ihre in Tabelle 3.3 maximale Größe bei gleichzeitiger Wahl des Einbankmodus vorzunehmen. Somit sind alle vorhandenen Pipes gleichzeitig in gleicher Art und Weise (keine Unterscheidung zwischen Ein- und Zweibank-Betrieb) nutzbar.

Nachdem nun klar ist, wie der Speicher einer Pipe festzulegen ist, kann sich mit weiteren Fragestellungen beschäftigt werden. Wie funktioniert der Datenaustausch zwischen Pipe beziehungsweise DPRAM und Software? Wie wird festgelegt, mit welchem Gerät und mit welchem auf diesem befindlichen Endpunkt kommuniziert wird? Und schließlich auch: Wie wird der Typ der Pipe – also beispielsweise Bulk, Interrupt oder Control – und das einfache Ein- und Ausschalten der Pipe geregelt?

Die folgenden Register enthalten Bits, mit denen diese Aufgaben zu erledigen sind. Dabei wird mit den Adresseinstellungen für die Pipe begonnen.

3.6.5.4 UHADDR

Dieses Register übernimmt die Adressierung des USB-Gerätes, mit dem kommuniziert werden soll. Auffällig ist, dass es sich um ein Register aus der Gruppe der USB-Host-Register handelt und dem Registernamen auch kein "X" nachgestellt ist. Dies bedeutet, dass sich die hier befindliche Adresse auf alle Pipes bezieht. Die Entwickler des AT90USB1287 hatten also wohl nicht im Sinn, einmal Hubs und mehrere Endgeräte gleichzeitig am Controller zu betreiben. Die Software muss später versuchen, den Mehrgerätebetrieb auf andere Weise zu ermöglichen.

Bit	7	6	5	4	3	2	1	0	
		HADDR6	HADDR5	HADDR4	HADDR3	HADDR2	HADDR1	HADDR0	**UHADDR**
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R	
Initial Value	0	0	0	0	0	0	0	0	

HADDR6:0 – Host Adress Configuration Diese Bits bilden die siebenbittige Adresse des USB-Gerätes, mit dem kommuniziert werden soll.

3.6.5.5 UPCFG0X

Mit dem UPCFG0X-Register kann ein spezieller Endpunkt auf dem durch UHADDR vorgegebenen Gerät adressiert werden.

Bit	7	6	5	4	3	2	1	0	
	PTYPE1	PTYPE0	PTOKEN1	PTOKEN0	PEPNUM3	PEPNUM2	PEPNUM1	PEPNUM0	**UPCFG0X**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PEPNUM3:0 Über den Wert dieser vier Bits wird die Nummer des entsprechenden Endpunkts des Zielgeräts ausgewählt. Um beispielsweise einen Endpunkt mit der Nummer 13 zu adressieren, wird hier also $13_{10} = 1101_2$ eingetragen.

PTYPE1:0 – Pipe Type Selection Die PTYPE-Bits selektieren die Gattung, welcher die Pipe angehört. Die Belegung ist dabei der folgenden Tabelle zu entnehmen.

PTYPE1:0	00	01	10	11
Gattung	Control	Isochronous	Bulk	Interrupt

Tabelle 3.5: Pipe-Type Konfiguration in UPCFG0X.

PTOKEN1:0 – Pipe Token Selection Diese Bits wählen den Tokentyp, den die Pipe bei Aktivität generiert. Die möglichen Tokens sind in Tabelle 3.6 aufgelistet.

3.6.5.6 UPCFG2X

Der AT90USB1287 erlaubt – wie schon in der Konfiguration für die Pipe-Gattung gesehen – die Erstellung von Pipes zur Kommunikation mit Interrupt-Endpunkten. Dabei kann die Hardware die von USB vorgesehenen

PTOKEN1:0	00	01	10	11
Token-Typ	SETUP	IN	OUT	Reserved

Tabelle 3.6: Pipe-Token-Typ Konfiguration in UPCFG0X.

Abfragen autonom vornehmen, wobei die Frequenz der Abfragen durch das UPCFG2X-Register gegeben ist.

Bit	7	6	5	4	3	2	1	0	
	INTFRQ7	INTFRQ6	INTFRQ5	INTFRQ4	INTFRQ3	INTFRQ2	INTFRQ1	INTFRQ0	**UPCFG2X**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

INTFRQ7:0 – Interrupt Pipe Request Frequency Die in diesen Bits gespeicherte Zahl gibt an, wie viele Millisekunden zwischen zwei (durch die Hardware automatisch vorgenommenen) Abfragen eines Interrupt-Endpoints liegen sollen. Für Pipes, die einen von "Interrupt" verschiedenen Typ haben, hat der Wert der INTFRQ-Bits keine Bedeutung.

3.6.5.7 UPCONX

Mit Hilfe der Bits im Register UPCONX lässt sich die Pipe aktivieren und auch die Requestgenerierung starten. Außerdem lässt sich noch auf einige weitere Statuseinstellungen der Pipe Einfluss nehmen.

Bit	7	6	5	4	3	2	1	0	
		PFREEZE	INMODE		RSTDT			PEN	**UPCONX**
Read/Write	R	R/W	R/W	R	R/W	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PEN – Pipe Enable Das Setzen dieses Bits führt zur Aktivierung der Pipe. Hierbei wird die Konfiguration geprüft und bei erfolgreichem Test CFGOK in UPSTAX (3.6.5.9) gesetzt. Nach dem Einschalten befindet sich die Pipe im eingefrorenen Zustand.

PFREEZE – Pipe Freeze Solange dieses Bit gesetzt ist, wird die Pipe keine Tokenpakete auf den Bus bringen. Das Bit wird automatisch für nicht konfigurierte Pipes oder im Fehlerfall gesetzt. Sobald PFREEZE gelöscht wird, beginnt die Pipe mit der Erzeugung von Tokenpaketen ihres jeweiligen Typs (eigenstellt über UPCFG0X).

INMODE – Infinite In-Request Mode Mittels dieses Bits kann der Controller dazu veranlasst werden, im Falle einer IN-Pipe eine unendliche Anzahl von Requests durchzuführen, statt die Pipe nach einer durch UPINRQX gegebenen Anzahl von (erfolgreichen) Request-Versuchen einzufrieren. Die Software ist dann dafür verantwortlich, bei Bedarf das Bit PFREEZE für die Pipe zu setzen.

Sobald INMODE gesetzt ist, verliert das Register UPINRQX seine Bedeutung[14].

RSTDT – Reset Data Toggle Sobald das RSTDT-Bit gesetzt ist, setzt die Hardware das Data-Toggle für die entsprechende Pipe zurück (Siehe hierzu auch Abschnitt 3.6.5.9, DTSEQ1:0). Leider gibt es keine sonstige Möglichkeit, das Data-Toggle einer Pipe direkt zu beeinflussen. Für den Mehrgerätebetrieb wäre ein direktes Setzen und Löschen dieser Bits wünschenswert.

RSTDT wird von der Hardware gelöscht, sobald das Rücksetzen des Data-Toggles erfolgreich verlaufen ist.

3.6.5.8 UPRST

Über die Bits dieses Registers können die Bankverwaltung sowie die Statusflags einer Pipe zurückgesetzt werden. Wichtig ist, dass die eigentliche Konfiguration der Pipe davon unberührt bleibt. Hierzu zählen zum Beispiel Typ- und Größeneinstellung. Auch das Data-Toggle ist von einem Rücksetzen der Pipe nicht betroffen.

Bit	7	6	5	4	3	2	1	0	
		P6RST	P5RST	P4RST	P3RST	P2RST	P1RST	P0RST	**UPRST**
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

P6:0RST – Pipe X Reset Ein Setzen und anschließendes Löschen eines dieser Bits setzt die Konfiguration von P_X zurück.

3.6.5.9 UPSTAX

Das UPSTAX-Register dient zur Statuskontrolle der Pipe P_X.

Bit	7	6	5	4	3	2	1	0	
	CFGOK	OVERFI	UNDERFI		DTSEQ:1	DTSEQ:0	NBUSYBK:1	NBUSYBK:0	**UPSTAX**
Read/Write	R	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

[14] UPINRQX enthält nur die Anzahl der Requests bis zum Pipe-Freeze als 8Bit-Zahl.

CFGOK – Configure Pipe Memory OK Dieses Bit wird gesetzt, wenn die Hardware beim Einschalten einer Pipe (siehe PEN in UPCONX, Abschnitt 3.6.5.7) eine valide Konfiguration vorfindet. "Valide" bezieht sich dabei nur auch die für die jeweilige Pipe gewählten Werte und betrachtet keine möglichen Überschneidungen mit anderen Pipes, was beispielsweise die Speicherverwaltung betrifft. Das CFGOK-Bit wird durch die Hardware automatisch gelöscht, sobald die Pipe wieder ausgeschaltet wird.

OVERFI – Overflow Interrupt Flag Sollten einmal mehr Daten auf der jeweiligen Pipe zu empfangen sein, als die Pipe aufnehmen kann, so setzt die Hardware das OVERFI-Flag. Für den Fall, das auch das FLERRE-Bit (siehe 3.6.5.13) auf Eins gebracht wurde, wird dann auch ein Interrupt ausgelöst. Das OVERFI-Bit sollte von der Software zurückgesetzt werden.

UNDERFI – Underflow Interrupt Flag Das UNDERFI-Bit wird gesetzt, wenn einer Pipe weniger Daten zum Versand zur Verfügung stehen, als dem Endpunkt, mit dem die Pipe kommuniziert, zugesichert wurden. Bezüglich der Interruptfähigkeiten hat dieses Bit die gleichen Eigenschaften wie OVERFI.

DTSEQ1:0 – Data Toggle Sequencing Flag Für OUT-Pipes zeigt dieses Feld das Data-Toggle, mit welchem das nächste zu sendende Paket markiert wird. Bei IN-Pipes ist das Data-Toggle des zuletzt auf der aktuellen Bank empfangenen Pakets sichtbar. Die Zuordnung erfolgt dabei laut Tabelle 3.7.

DTSEQ1:0	00	01	10	11
Toggle	DATA0	DATA1	Reserved	Reserved

Tabelle 3.7: Werte für das Data Toggle Sequencing Flag in UPSTAX.

NBUSYBK1:0 – Busy Bank Flag Durch diese Bits kann die Zahl der momentan (durch Daten) belegten Bänke einer Pipe bestimmt werden. Bei OUT-Pipes ergibt sich daraus die Kenntnis über noch nicht versandte Daten. Bei IN-Pipes ist feststellbar, wie viele Daten in der Pipe noch auf die Verarbeitung durch die Software warten. Tabelle 3.8 zeigt die in diesem Feld genutzte Belegung.

NBUSYBK1:0	00	01	10	11
Zahl der belegten Bänke	0	1	2	Reserved

Tabelle 3.8: Anzahl der in der Pipe belegten Bänke laut Busy Bank Flag in UPSTAX.

3.6.5.10 UPDATX / UPBCHX / UPBCLX

Über das Register UPDATX findet der eigentliche Datenaustausch statt. Es dient als Schnittstelle zum DPRAM, mit der byteweise Daten nach dem FIFO-Prinzip geschrieben beziehungsweise gelesen werden können. Das aus UPBCLX und UPBCHX gebildete Registerpaar dient dabei als Zähler für die im FIFO vorhandenen Daten.

Bit	7	6	5	4	3	2	1	0	
	PDAT7	PDAT6	PDAT5	PDAT4	PDAT3	PDAT2	PDAT1	PDAT0	**UPDATX**
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
						PBYCT10	PBYCT9	PBYCT8	**UPBCHX**
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	PBYCT7	PBYCT6	PBYCT5	PBYCT4	PBYCT3	PBYCT2	PBYCT1	PBYCT0	**UPBCLX**
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

PDAT7:0 Schreibende Zugriffe auf diese Bits führen zu einer Übernahme des geschriebenen Bytes in den FIFO-Speicher der jeweiligen Pipe im DPRAM. Lesende Zugriffe liefern entsprechend das nächste im FIFO befindliche Byte der Pipe. Lesende Zugriffe sind dabei nur bei IN-Pipes, schreibende nur bei OUT-Pipes sinnvoll.

PBYCT10:0 Diese Bits geben die Zahl der im Pipe-FIFO befindlichen Bytes wieder. Bei einer OUT-Pipe erhöht die Hardware den Wert für jedes von der Software in den FIFO gebrachte Byte um eins (schreibender Zugriff auf UPDATX). Die Hardware dekrementiert den Zählwert dementsprechend für jedes durch die USB-Einheit versendete Byte wieder. Das Verhalten für IN-Pipes ist analog: Die Hardware erhöht den Zählwert für jedes vom Bus entgegengenommene Byte und verringert ihn entsprechend für jedes von der Software gelesene Byte (lesender Zugriff auf UPDATX).

3.6.5.11 USBINTX

Über dieses Register sind verschiedene Zustandsflags für die jeweilige Pipe abrufbar. Abgelesen werden können beispielsweise die Bereitschaft zum Senden von Daten oder auch, ob ein neues Datenpaket empfangen wurden. Auch das Auftreten von Kommunikationsfehlern wird hier vermerkt und diese dann gegebenenfalls mittels des Registers UPERRX näher spezifiziert. Die meisten der hier vorhandenen Flags können als Interruptauslöser genutzt werden. Hierzu befinden sich in UPIENX die entsprechenden Enable-Bits.

Bit	7	6	5	4	3	2	1	0	
	FIFOCON	NAKEDI	RWAL	PERRI	TXSPI	TXOUTI	RXSTALLI/CRCERRI	RXINI	**UPINTX**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PERRI – Pipe Error Dieses Bit zeigt einen Fehler in der Kommunikation über die momentan gewählte Pipe an. Die Fehlerquelle wird durch die Hardware näher in UPERRX (siehe 3.6.5.12) spezifiziert. Wird das entsprechende Fehlerbit dort gelöscht, so sieht die Hardware den Fehler als behandelt an und setzt PERRI automatisch zurück.

RXSTALLI/CRCERR – Stall Received/Isochronous CRC-Error detected Dieses Bit signalisiert den Empfang eines STALL-Pakets über P_X. Hierbei hält die Hardware automatisch die Requestgenerierung für die Pipe an, es wird also PFREEZE gesetzt. Das Bit ist nach Behandlung des STALLs durch die Software zurückzusetzen.

Handelt es sich bei P_X um eine Isochronous-Pipe, so dient dieses Bit als CRC-Error-Flag[15].

RWAL – Read/Write Allow Über dieses Flag kann festgestellt werden, ob der Pipe-FIFO momentan zugreifbar ist. Bei OUT-Pipes ist dies nicht der Fall, wenn der FIFO voll ist. Bei IN-Pipes kann entsprechend kein Datum aus dem FIFO gelesen werden, wenn dieser leer ist. Auch im Fehlerfall löscht die Hardware dieses Bit, wobei dann gleichzeitig PERRI beziehungsweise RXSTALLI gesetzt wird.

NAKEDI – NAK Handshake Received Die Hardware setzt dieses Bit, sobald sie ein NAK für die aktuelle Pipe feststellt. Nach der Behandlung des NAK muss NAKEDI durch die Software zurückgesetzt werden.

[15]Man beachte, dass im Falle des CRC-Fehlers trotzdem noch ein neues Paket durch RXINI signalisiert wird.

TXSTPI – Setup Bank Ready TXSTPI wird gesetzt, wenn die Hardware feststellt, dass bei einer Control-Pipe des Tokentyps "SETUP" eine Bank zum Versenden von Daten verfügbar ist. Die Software kann dann Daten in den FIFO schreiben und diese versenden. Das Bit sollte von der Software zurückgesetzt werden.

TXOUTI – Out Bank Ready Dieses Bit wird automatisch gesetzt, wenn bei einer Pipe mit Tokentyp "OUT" eine Bank zum Versenden von Daten frei ist. Wie bei TXSTPI auch, kann die Software dann Daten in den FIFO schreiben und diese zum Versand bringen. Auch TXOUTI muss von der Software zurückgesetzt werden.

RXINI – In Data Received Bei Empfang eines Paketes über USB setzt die Hardware RXINI automatisch auf Eins. Die Software kann die empfangenen Daten dann aus dem FIFO lesen, wobei sie die Anzahl der zu lesenden Bytes dem Registerpaar UPBCX (siehe 3.6.5.10) entnehmen kann. Genau wie TXSTPI und TXOUTI muss RXINI durch die Software zurückgesetzt werden.

FIFOCON – FIFO Control Für Pipes des Typs "OUT" und "SETUP" führt ein Löschen des FIFOCON-Bits zum Senden der im FIFO befindlichen Daten. Im Zweibankmodus wird gleichzeitig auch die nächste Bank selektiert, so dass parallel zum Senden mit dem weiteren Füllen des FIFO fortgefahren werden kann. Die Hardware setzt FIFOCON jeweils gleichzeitig mit TXOUTI beziehungsweise TXSTPI.

Im Falle einer IN-Pipe setzt die Hardware FIFOCON gleichzeitig mit RXINI. Hier führt ein Löschen von FIFOCON zu einer Markierung der aktuellen Bank als "frei", also als "Bereit zur Aufnahme neuer Daten vom Bus". im Zweibankbetrieb wird auch hier auf die jeweils andere Bank umgeschaltet.

3.6.5.12 UPERRX

Das UPERRX-Register dient zur näheren Identifikation von Fehlern, die allgemein über ein gesetztes PERRI-Bit signalisiert wurden. Die Software muss die entstandenen Fehler jeweils durch Rücksetzen der beschriebenen Bits quittieren, dann wird gleichzeitig auch PERRI von der Hardware gelöscht.

Bit	7	6	5	4	3	2	1	0	
		COUNTER1	COUNTER0	CRC16	TIMEOUT	PID	DATAPID	DATAGL	**UPERRX**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

COUNTER1:0 Dieser Zähler wird bei jedem Auftreten eines Fehlers auf der Pipe inkrementiert. Sobald der Zählstand drei erreicht, wird die Requestgenerierung der Pipe mittels PFREEZE angehalten.

CRC16 – CRC16 Error Stellt die Hardware einen CRC16-Fehler beim Prüfsummentest der übertragenen Daten fest, so setzt sie dieses Bit.

TIMEOUT – Timeout Error Bei Auftreten eines Timeout-Fehlers wird das Timeout-Bit auf Eins geschrieben.

PID/DATAPID – PID Error/Data PID Error Liegt eine Bitfolge als PID auf dem Bus, die keine valide PID nach Tabelle 2.1 darstellt, so wird dies von der Hardware durch Setzen des PID- beziehungsweise DATAPID-Bits signalisiert. Das Vorhandensein beider Bits legt nahe, dass dabei zwischen Token- und Handshake- sowie den Data-PIDs unterschieden wird. Das Datenblatt gibt hierzu jedoch keine nähere Auskunft. Da beides im Rahmen dieser Arbeit als fehlerhafte Übertragung angesehen werden muss, können beide Fehler auch gleich behandelt werden.

DATATGL – Bad Data Toggle Error Wird beim Empfang von Paketen ein Data-Toggle-Fehler – also der Empfang von zwei Paketen gleicher Sequenznummer in Folge – festgestellt, so wird dieses Bit gesetzt. Dieser Fehler tritt dann auf, wenn beim Lesen von Daten vom Bus ein Paket zwar erfolgreich empfangen wurde, das ACK jedoch auf dem Weg zum Gerät verloren ging. Das Paket wird in diesem Fall vom AT90USB1287 mit einem ACK bestätigt und als Duplikat verworfen. Danach sind die Sequenznummern von Pipe und Endpunkt wieder synchron[16].

3.6.5.13 UPIENX

In diesem Register befinden sich die Interrupt-Enable-Bits für OVERFI und UNDERFI in UPSTAX sowie NAKEDI, PERRI, TXSTPI, TXOUTI, RXSTALLI und RXINI. Bis auf FLOWERRE, welches das Enable-Bit für UNDERFI und OVERFI darstellt, sind die Zugehörigkeiten unmittelbar erkennbar.

Bit	7	6	5	4	3	2	1	0	
	FLERRE	NAKEDE		PERRE	TXSTPE	TXOUTE	RXSTALLE	RXINE	**UPIENX**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

[16]Siehe zu dieser Problematik auch Abschnitt 2.4.4.1 zum Verhalten von Sequenznummern.

3.6.5.14 UPINT

Anhand dieses Registers lässt sich feststellen, welche Pipe für das Auslösen eines Interrupts verantwortlich war.

Bit	7	6	5	4	3	2	1	0	
		PINT6	PINT5	PINT4	PINT3	PINT2	PINT1	PINT0	**UPINT**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PINT6:0 Ist P_X für das Auslösen eines Interrupts verantwortlich, so wird hier das entsprechende Bit gesetzt. Die Hardware setzt das Bit nach Behandlung des Interrupt – also dem Löschen des Bits, welches den Interrupt auslöste – automatisch wieder zurück.

3.6.6 Übersicht über die Nutzung von Pipes

Das Datenblatt gibt bezüglich der Benutzung und Konfiguration der Pipes einige Hilfestellung, um den Einstieg in die Programmierung zu erleichtern. Der Vollständigkeit halber seien auch diese Hinweise, mitunter mit zusätzlichen Informationen beziehungsweise Klarstellungen, hier wiedergegeben.

3.6.6.1 Aktivierung und Grundkonfiguration

Zur Aktivierung einer Pipe empfiehlt das Datenblatt einen Programmfluss, wie er verfeinert in Abbildung 3.12 dargestellt ist. Das Datenblatt erlaubt alternativ zu diesem Programmfluss auch eine Veränderung des Tokentyps der Pipe, nachdem die Speicherkonfiguration bereits durchgeführt wurde. Die Bits PTOKEN1:0 in UPCFG0X sind also auch nach Erreichen des CFGOK veränderbar. Im Versuch zeigte sich, dass darüber hinaus auch die restlichen Bits des Registers UPCF0X nachträglich modifiziert werden können. Die Nummer des adressierten Endpunktes sowie sowie auch der Pipe-Typ (Control, Isochronous, Bulk oder Interrupt) können also ebenfalls noch nach der Speicherallokation angepasst werden.

3.6.6.2 IN-Pipes

Neben den in der Grundkonfiguration genutzten Bits ist bei Erstellung einer IN-Pipe noch das Bit INMODE von Bedeutung. Je nach Wunsch ist dieses also nach der Grundkonfiguration zu setzen. Danach ist die Pipe einsatzbereit.

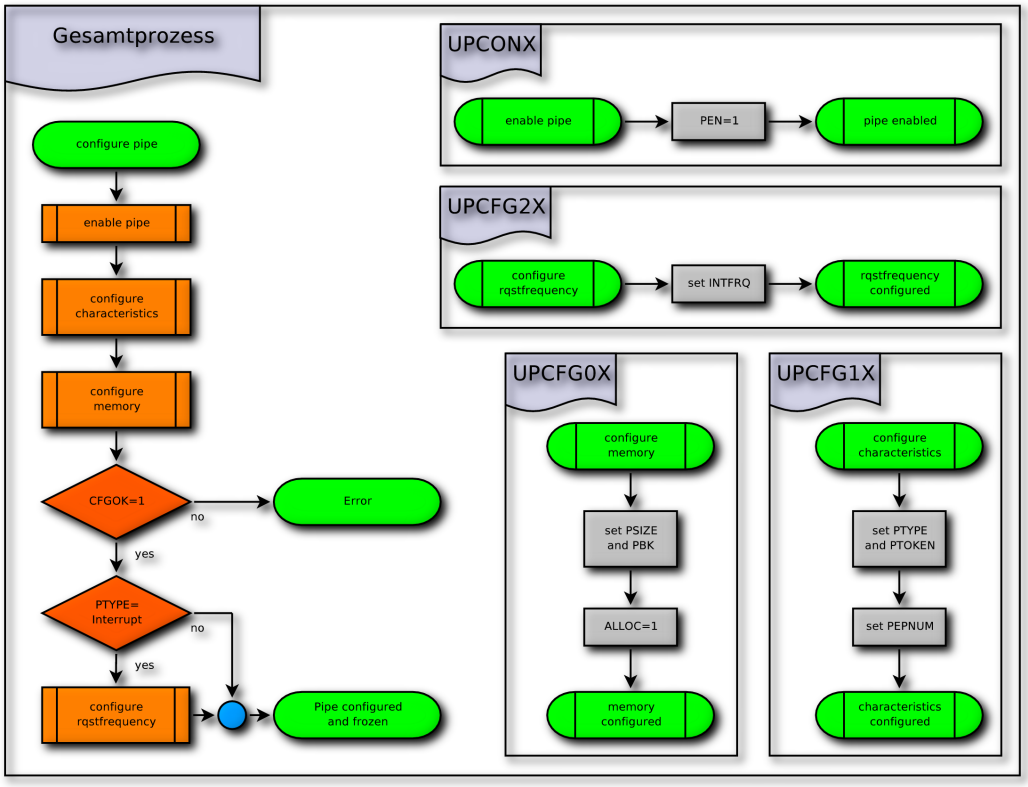


Abbildung 3.12: Beispielhafter Ablaufplan für die Inbetriebnahme einer Pipe.

Die Zeitdiagramme in Abbildung 3.13 beziehungsweise 3.14 illustrieren die Vorgänge beim Datenempfang vom Gerät im Ein- und im Zweibankbetrieb.

Der Host beginnt mit der Anforderung von Daten vom Gerät – also dem Generieren von IN-Requests – sobald die Pipe durch Löschen von PFREEZE zur Arbeit gebracht wird. Hat das Gerät in Reaktion auf das IN-Token Daten gesendet, so zeigt die Hardware dies durch Setzen von RXINI an. RXINI wird von der Software dann als Quittung zurückgesetzt und die empfangenen Daten werden durch Lesen von UPDATX aus dem FIFO entnommen. Ist die gewünschte Datenmenge von der Software entgegengenommen worden, so markiert diese die aktuelle Bank als frei, indem FIFOCON gelöscht wird. Im Zweibankbetrieb schaltet sie damit auch auf die zweite Bank um, aus der dann gegebenenfalls wieder Daten entnommen werden können.

In jedem Fall legt die Hardware jeweils ein neues IN-Token auf den Bus, sobald eine freie Bank zur Verfügung steht. Im Einbankbetrieb hat dies ins-

besondere zur Folge, dass das Senden des IN direkt durch das Löschen von FIFOCON gesteuert wird.

3.6.6.3 OUT-Pipes

Um Daten vom Host zum Gerät zu übertragen, wird die OUT-Pipe durch entsprechende Einstellungen mittels des in Abbildung 3.12 gezeigten Ablaufs konfiguriert. Bevor Daten gesendet werden können, muss die Pipe außerdem durch Löschen von PFREEZE arbeitsbereit gemacht werden.

Die Abbildungen 3.15 und 3.16 zeigen das darauf folgende Verhalten für den Ein- beziehungsweise Zweibankbetrieb.

Hat die Hardware eine freie Bank zur Aufnahme von Daten von der Software zur Verfügung, so signalisiert sie dies durch ein gesetztes TXOUTI-Bit. Die Software quittiert diese Nachricht durch Löschen von TXOUTI und kann die Bank dann mittels wiederholter Schreibzugriffe auf UPDATX füllen. Sind die gewünschten Daten in die Bank gebracht, so bringt die Software die USB-Einheit durch Löschen von FIFOCON zum Versenden der Daten. Beim Versenden generiert der Controller ein OUT-Token und verschickt danach die Daten. Sobald wieder eine freie Bank zur Verfügung steht, setzt die Hardware FIFOCON und TXOUTI auf Eins. Eine Bank wird jeweils dann als frei angesehen, wenn auf das als letztes durch sie erzeugte Datenpaket ein ACK erfolgte.

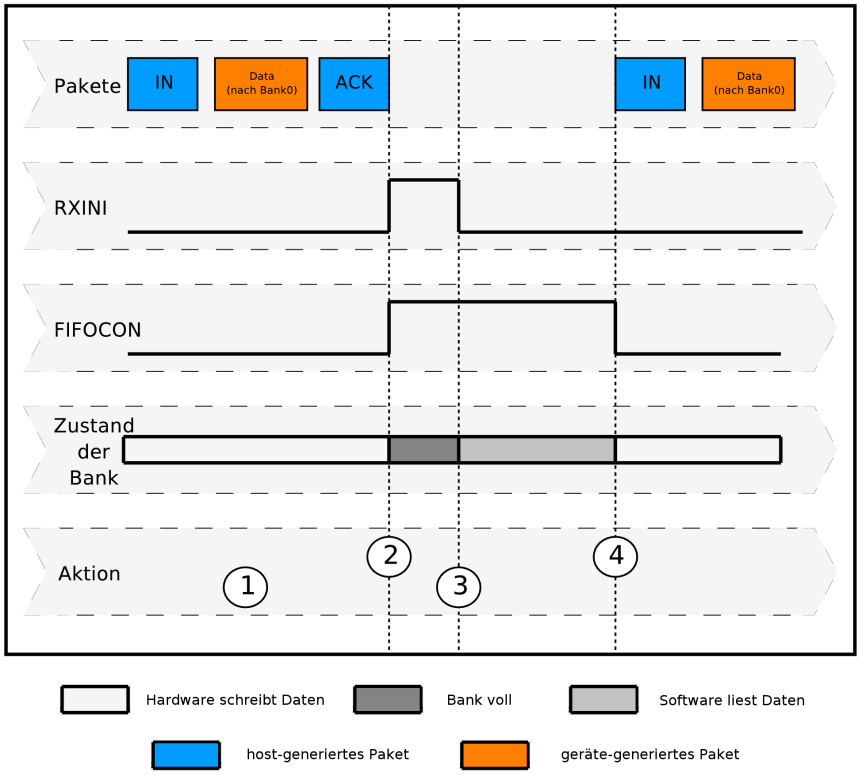
3.6.6.4 Pipe-Verhalten bei NAKs

Sollte ein Geräteendpunkt während der Kommunikation ein NAK an den Host liefern, so setzt die Hardware das NAKEDI-Flag in UPINTX. Der Controller hält jedoch auf ein NAK hin nicht die Generierung von Tokenpaketen an. Die USB-Einheit wird also nach einem NAK (ohne Zutun der Software) weitere Transaktionsversuche vornehmen.

3.6.6.5 Control-Pipes

Das Nutzen von Control Pipes ist – der Natur der Control Transfers entsprechend – aus Sicht der Software etwas komplizierter.

Zur Erinnerung: Control Transfers bestehen jeweils mindestens aus einer Setup- gefolgt von einer Status-Stufe. Je nach Transfer kann zwischen diese beiden auch noch eine Daten-Stufe geschoben sein, in der Daten in beliebiger Richtung übertragen werden. Die benötigten Stufen muss die Software durch Ändern des Tokentyps der benutzten Control-Pipe selbst erzeugen. In jeder Stufe hat die Software dafür zu sorgen, dass spezifaktionsgerecht Daten



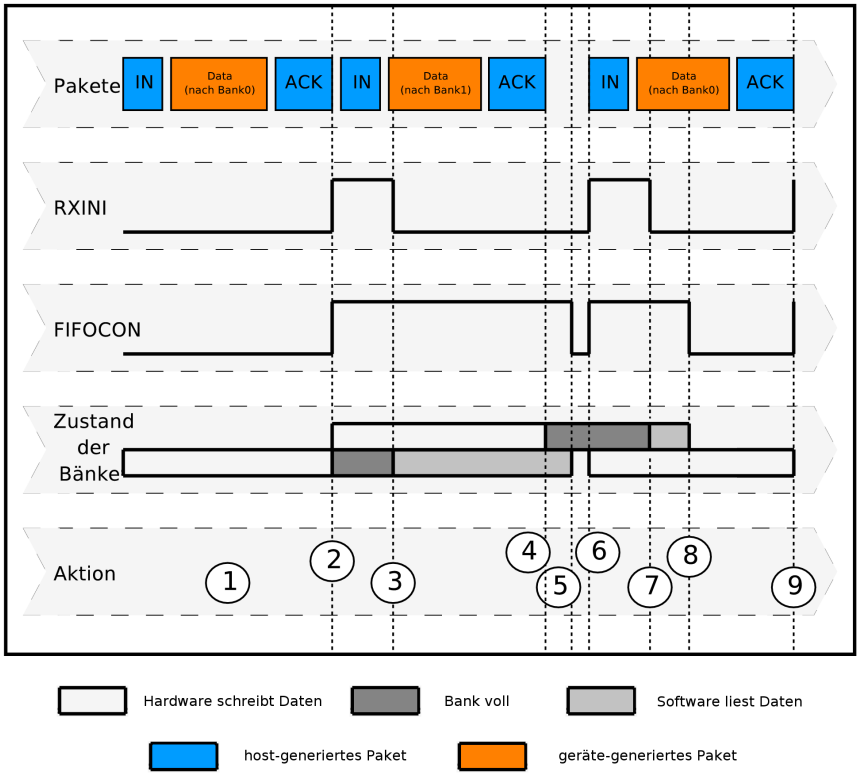
1. Die Bank ist frei und die Hardware hat nach einem im Vorfeld erfolgten Löschen von PFREEZE mit dem Empfangen von Daten begonnen.

2. Nachdem der Controller den Empfang der Daten vom Gerät mittels ACK quittiert hat, setzt die Hardware RXINI und FIFOCON. Die Bank ist nun belegt, wird aber noch nicht ausgelesen.

3. Die Software stellt den Paketeingang anhand von RXINI fest und quittiert diesen durch Löschen des Bits. Nun entnimmt sie die Daten durch lesenden Zugriff auf UPDATX aus dem Pipe-FIFO.

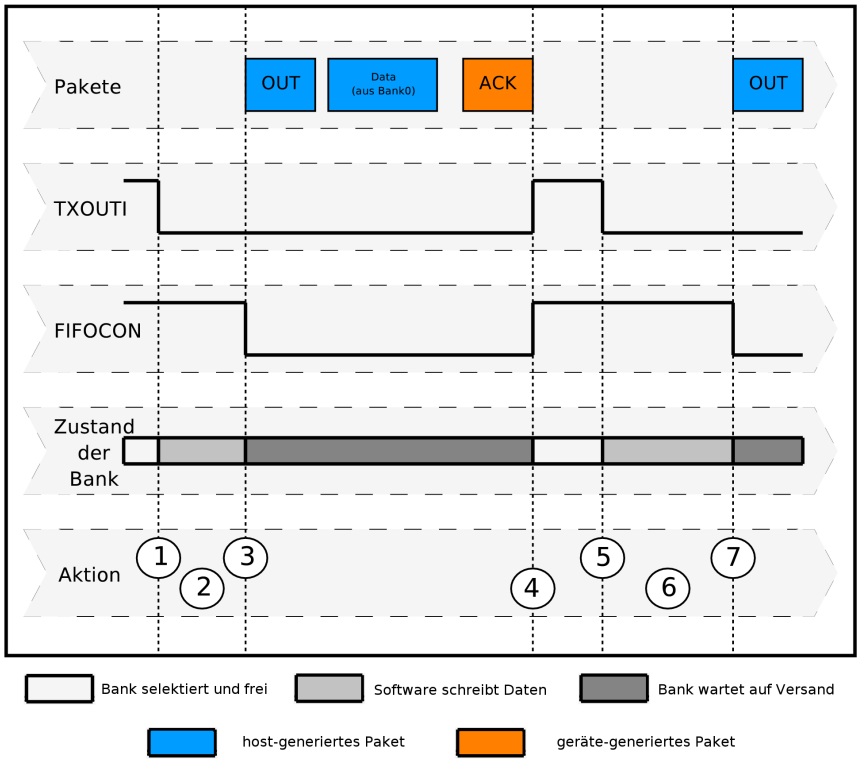
4. Nach Abschluss des Lesevorgangs löscht die Software FIFOCON, um die Bank als frei zu markieren. Die Hardware fährt dann mit dem nächsten IN-Request fort.

Abbildung 3.13: Zeitdiagramm für Pipes des Typs “IN” im Einbankbetrieb.



1. Beide Bänke sind frei und die Hardware hat nach einem im Vorfeld erfolgten Löschen von PFREEZE mit dem Empfang von Daten nach Bank1 begonnen.
2. Nachdem der Controller den Empfang der Daten vom Gerät mittels ACK quittiert hat, setzt die Hardware RXINI und FIFOCON. Bank1 ist nun belegt, wird aber noch nicht ausgelesen. Zugleich wird bereits das nächste Paket vom Gerät angefordert, welches jetzt in Bank2 landet.
3. Die Software stellt den Eingang von Paket1 anhand von RXINI fest und quittiert diesen durch Löschen des Bits. Dann entnimmt sie die Daten durch lesenden Zugriff auf UPDATX aus dem Pipe-FIFO von Bank1.
4. Der Transfer von Paket2 ist abgeschlossen. Bank2 ist damit belegt und auch Bank1 wird noch durch den lesenden Zugriff der Software blockiert. Die Hardware kann also auf dieser Pipe zunächst keine weiteren Daten vom Gerät anfordern.
5. Nach Abschluss des Lesevorgangs löscht die Software FIFOCON, womit Bank1 als frei markiert wird.
6. Die Hardware erkennt die freie Bank1 und kann nun mit dem nächsten IN-Request (Paket3) fortfahren. FIFOCON und RXINI werden gesetzt, da auf Bank2 noch Paket2 auf den Abruf durch die Software wartet.
7. Die Software quittiert das wartende Paket2 durch Rücksetzen von RXINI und liest die Daten dann aus dem FIFO.
8. Diesmal wurde zum Lesen weniger Zeit benötigt. Der aktuelle Lesevorgang wird bereits mittels FIFOCON durch die Software als abgeschlossen markiert, bevor der Transfer von Paket3 abgeschlossen ist. Damit stellt sich eine zu 1. identische Situation ein.
9. Paket3 wurde komplett empfangen, die Software kann wieder auf RXINI reagieren.

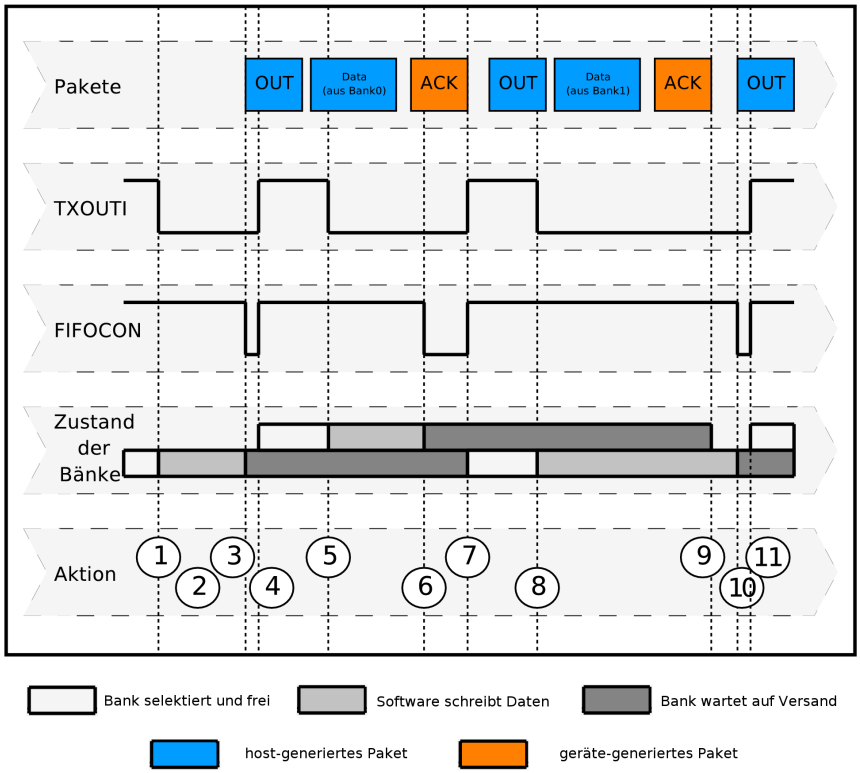
Abbildung 3.14: Zeitdiagramm für Pipes des Typs “IN” im Zweibankbetrieb.



Anfänglich ist die Bank leer, die Hardware macht dies durch FIFOCON und TXOUTI kenntlich.

1. Die Software bemerkt die freie Bank und quittiert mit Löschen von TXOUTI.
2. Die Software schreibt in die Bank durch Zugriff auf UPDATX.
3. Die Software gibt durch Löschen von FIFOCON den Sendebefehl, die Hardware legt daraufhin ein OUT-Token, gefolgt von dem Datenpaket auf den Bus. Dabei ist die Bank blockiert.
4. Nach dem ACK durch das Gerät wird die Bank wieder frei. Die Hardware signalisiert dies durch Setzen von TXOUTI und FIFOCON.
5. Die Software bestätigt die freie Bank durch Löschen von TXOUTI.
6. Die Software schreibt weitere Daten in die Bank.
7. Auch diese Daten werden durch Löschen von FIFOCON zum Versand gebracht. Die Hardware beginnt wieder mit einem OUT-Token.

Abbildung 3.15: Zeitdiagramm für Pipes des Typs “OUT” im Einbankbetrieb.



Anfänglich sind beide Bänke leer, Bank1 ist die nächste zu füllende Bank. Die Hardware macht die Bereitschaft durch FIFOCON und TXOUTI kenntlich.

1. Die Software bemerkt eine freie Bank und quittiert mit Löschen von TXOUTI.
2. Die Software schreibt in Bank1 durch Zugriff auf UPDATX.
3. Die Software gibt durch Löschen von FIFOCON den Sendebefehl, die Hardware legt daraufhin das Datenpaket auf den Bus.
4. Die Hardware stellt fest, dass Bank2 frei ist und signalisiert dies wiederum durch TXOUTI und FIFOCON.
5. Die Software quittiert TXOUTI. Nun schreibt sie Daten in Bank2.
6. Die Software löscht FIFOCON, um auch die neuen Daten zu versenden. Da das Paket aus Bank1 noch nicht quittiert wurde, sind nun beide Bänke blockiert.
7. Das ACK zu Paket1 ist angekommen und Bank1 wird frei. Entsprechend werden wieder TXOUTI und FIFOCON gesetzt. Der Controller fährt schnellstmöglich mit dem Senden des Pakets aus Bank2 fort.
8. Die Software quittiert die freie Bank mit Löschen von TXOUTI. Ab hier kann wieder schreibend auf Bank1 zugegriffen werden.
9. Diesmal benötigt die Software mehr Zeit, um das nächste Paket aufzubauen. Das ACK zu Bank2 trifft ein, Bank2 wird damit intern als frei markiert.
10. Die Software löscht FIFOCON, um die Daten in Bank1 zu senden. Der Controller beginnt mit dem Senden und legt ein OUT-Token auf den Bus.
11. Analog zu 4. bemerkt die Hardware nun wieder die freie Bank2.

Abbildung 3.16: Zeitdiagramm für Pipes des Typs “OUT” im Zweibankbetrieb.

ausgetauscht werden. Eine detaillierte Beschreibung des Ablaufs von Control Transfers ist bereits in Abschnitt 2.4.4.4 erfolgt.

3.6.6.6 Inbetriebnahme von Geräten

Zur Inbetriebnahme von Geräten sind einige grundsätzliche Schritte notwendig, wie sie durch die Bus Enumeration (siehe Abschnitt 2.4.6.1) festgeschrieben sind.

Nach der Einsteckerkennung ist zunächst ein Reset des Geräts herbeizuführen. Dies kann durch Setzen des RESET-Bits berwerkstelligt werden, sofern das fragliche Gerät direkt mit dem AT90USB1287 verbunden ist. Sollte dies nicht der Fall sein – wird also ein Hub verwendet – so ist der Hub zum Senden eines Reset-Signals über den zum Anschluss des Gerätes genutzten Downstream-Port anzuweisen.

Nach einem erfolgreichen USB-Reset muss die Software die Default Control Pipe des AT90USB1287 konfigurieren. Hierzu ist die Pipe mit der Nummer 0 (UPNUM=0) über den in Abbildung 3.12 gezeigten Ablauf auf Pipetyp "Control", Speichergröße 64Byte sowie Bankanzahl 1 einzustellen.

Dann kann mit der Setup-Stufe[17] des ersten Control Transfers begonnen werden. Dieser erste Transfer ist notwendigerweise ein Control-Request vom Typ "SET ADDRESS" mit Zieladresse 0, mit welchem dem Gerät eine freie Bus-Adresse zugewiesen wird. Es ist zu beachten, dass die Hardware des AT90USB1287 das Adressregister UHADDR nach Ausführen eines USB-Resets über das RESET-Bit automatisch auf Null setzt.

Jetzt ist das Gerät einsatzbereit und kann mittels weiterer Requests zur Arbeit bewegt werden.

Hiermit ist die Übersicht über die Hardware abgeschlossen. Mit dem Wissen über die vorgstellten Abläufe und Zusammenhänge kann nun mit der Entwicklung der Software fortgefahren werden.

[17] Der Tokentyp kann bereits bei der Aktivierung der Pipe mit den anderen genannten Einstellungen zusammen auf "SETUP" gebracht werden.

Kapitel 4

Software

4.1 Toolchain zur Programmentwicklung

Bevor mit der Entwicklung der Software begonnen werden kann, muss für eine geeignete Entwicklungsumgebung gesorgt werden. Neben einem geeigneten Quelltexteditor werden für diese Arbeit noch einige weitere Pakete benötigt. Diese sind im einzelnen:

avr-gcc Zum Compilen des in C verfassten Sourcecodes kommt *avr-gcc* zum Einsatz. Hierbei handelt es sich um den bekannten GNU-Compiler, welcher bereits zum Übersetzungszeitpunkt als Crosscompiler konfiguriert wurde, um von der Entwicklungsplattform aus Binärcode für den AVR-Controller zu erzeugen. Das GCC-Paket kann beispielsweise von [Tood] bezogen werden.

avr-binutils Eine für die Erstellung von Binaries für den AVR-Controller verwendbare Version der GNU-binutils. Das Paket enthält unter anderem den Linker *avr-ld* sowie den Assembler *avr-as*. Die binutils können bei [Tooc] heruntergeladen werden.

avr-libc Bei *avr-libc* handelt es sich um eine speziell für die Nutzung mit AVR-Controllern ausgelegte Version der libc. Neben Definitionen für den Umgang mit Ports, Pins und Registern enthält sie auch praktische Funktionen wie printf () und geeignete Funktionen, die den Umgang mit Interrupts erleichtern. Aktuelle Versionen wissen bereits mit dem AT90-USB1287 umzugehen, was die zeitraubende und fehleranfällige Arbeit der Erstellung von Definitionen für beispielsweise Registeradressen, wie sie für neue Controller notwendig ist, obsolet macht. Die *avr-libc* kann über [Toob] bezogen werden.

avrdude Wurde mit Hilfe der oben genannten Werkzeuge Code für den AVR-Controller erzeugt, so muss dieser noch in den Programmspeicher des μC gebracht werden. Hierzu nutzen die Autoren ein Tool namens *avrdude*. *avrdude* kann – wie alle erwähnten Programme kostenlos – von [Toof] heruntergeladen werden. Weitere Informationen finden sich auch auf der privaten Homepage des Entwicklers [Tooa][1].

Die Entwicklung für diese Arbeit fand komplett unter Linux statt. Können die nötigen Programmpakete nicht über das Installationssystem der Distribution eingespielt werden, so sind sie relativ leicht aus den Quellen zu erzeugen. Für Windows stellt das WinAVR-Projekt ([Tooe]) fertige Binaries der kompletten Toolchain zur Verfügung.

4.2 Grundsatzüberlegungen

Zunächst gilt es, sich über grundlegende Eigenschaften der zu programmierenden Software Klarheit zu verschaffen. Hierzu zählen zum einen Überlegungen zu den Aufgaben, die generell durch den zu entwickelnden Code erledigt werden sollen. Zum anderen wird dadurch aber auch die gebotene Programmierschnittstelle für den späteren Anwender beziehungsweise für den Entwurf von klassen- und gerätespezifischen Treibern beeinflusst.

Sicherlich ist es nicht sinnvoll, auf einem Mikrocontroller der AVR-Klasse eine USB-Host-Unterstützung zu entwerfen, die vollständig im Sinne der Spezifikation ist. Dies hat zum einen praktische Überlegungen bezüglich des knappen zur Verfügung stehenden Programm- und Datenspeicherplatzes zur Grundlage. Zum anderen ist aber auch die Komplexität des Systems überschaubar zu halten, um den späteren Einstieg in die Programmierung mit Hilfe der erstellten Software – und damit auch die Realisierung von Folgeprojekten – nicht unnötig zu erschweren. Unnötig vor allem deshalb, weil für den beabsichtigten Einsatzbereich in Klein- und Kleinstgeräten des Embeddedsektors Möglichkeiten wie Bandbreitengarantien für Isochronous Transfers aller Wahrscheinlichkeit nach als vernachlässigbar eingestuft werden können. Auch der Umgang mit Unicode String Descriptoren und ähnlichem kann zunächst getrost vernachlässigt werden.

Ziel bei der Entwicklung muss vielmehr das Erreichen einer Funktionalität sein, welche den Betrieb von Geräten und das Entwickeln von Treibern und darauf basierenden Anwendungen so einfach wie möglich gestaltet.

[1] Da es sich beim AT90USB1287 um einen modifizierten AtMega128 handelt, ist beim Programmieren mit avrdude der Chiptyp auf m128 einzustellen.

4.2.1 Mehrgerätebetrieb

Das Thema "Betrieb von Geräten" führt dabei zu einer den Entwicklungsprozess wesentlich drastischer beeinflussenden Frage: Ist der Betrieb mehrerer USB-Geräte gleichzeitig mit der Hardware des AT90USB1287 realisierbar?

Eine genauere Betrachtung zeigt, dass die Hauptprobleme bei einem Betrieb mehrerer Geräte beim AT90USB1287 aus drei Teilbereichen stammen. Sie betreffen die Adressierung, den Betrieb von Lowspeed-Geräten sowie auch die Sicherung der Übertragung mit Hilfe des Data-Toggles.

Die Probleme ergeben sich aus der Hardware des Controllers, welche augenscheinlich nicht für einen Mehrgerätebetrieb entwickelt wurde. Können die in den Teilbereichen auftretenden Probleme durch geeignete Programmierung gelöst werden oder sind die durch sie verursachten Einschränkungen hinnehmbar, so ist ein Mehregerätebetrieb mit dem AT90USB1287 möglich. Im Folgenden werden die einzelnen Probleme erläutert.

4.2.1.1 Problematik der Adressierung

Bereits bei der Registerbeschreibung ist aufgefallen, dass der AT90USB1287 nur ein einziges Register zur Adressierung des Zielgeräts für Datenübertragungen besitzt[2]. Somit genügt es nicht, ein Gerät (beziehungsweise konkreter: den Endpunkt eines spezifischen Gerätes) an eine der zur Verfügung stehenden Pipes zu binden, so dass dann mit dieser Einstellung fortlaufend Kommunikation betrieben werden kann. Vielmehr muss vor jedem Kommunikationsvorgang dafür Sorge getragen werden, dass das besagte Adressregister wieder mit der richtigen Zieladresse beschrieben wird, da sonst keine korrekte Datenübertragung zustande kommt. Da prinzipell jeder Transfer durch den Host angestoßen wird, scheint dies zunächst auch relativ problemlos möglich zu sein.

Ein Fallstrick bei dieser Überlegung verbirgt sich jedoch in der Hardwareunterstützung für Interrupt-Pipes. Zwar wird natürlich auch die Datenübertragung über Interrupt-Pipes auf Hostseite angestoßen, jedoch ist hierbei nicht die Software unmittelbar für den Zeitpunkt der Datenübertragung verantwortlich, wie es bei der Übertragung über andere Pipetypen der Fall ist. Vielmehr stößt die Hardware die Übertragung zu Zeitpunkten an, die von ihr in Übereinstimmung mit der durch die Software konfigurierten Request-Frequenz gewählt werden[3]. Da die Hardware aber auch über das Interrupt-Modell keine Möglichkeit bietet, unmittelbar vor Ausführung des Interrupt

[2]Siehe hierzu Register UHADDR in Abschnitt 3.6.5.4.

[3]Siehe Registerbeschreibung zu UPCFG2X, Abschnitt 3.6.5.6 sowie Abschnitt 3.6.6.1 zur Pipe-Konfiguration.

Transfers noch einmal eine Änderung des Adressregisters von Softwareseite zu forcieren, scheitert die Mehrgeräteunterstützung für die von Controllerseite angebotenen Interrupt-Pipes.

Ähnlich verhält es sich auch mit den ohnehin nur schwierig zu handhabenden Isochronous-Pipes. Erzeugt die Applikation bei Isochronous-OUT-Pipes nicht eine genügende Anzahl von Daten pro Frame, so würde der AT90USB1287 trotzdem mindestens für ein das Senden eines ZLPs sorgen, um dem Endpunkt Pakete mit der ausgehandelten Frequenz zukommen zu lassen[4]. Ändert sich der Wert im Adressregister zwischenzeitlich aber, so würde auch ein solcher automatisierter Zugriff auf Isochronous-Pipes aufgrund von Fehladressierung fehlschlagen.

Prinzipiell bleibt bei Interrupt-Pipes jedoch die Möglichkeit, deren Vorhandensein mit Hilfe der integrierten Timer des AT90USB1287 zu simulieren. Dabei ergibt sich für den die Bibliothek nutzenden Programmierer der Eindruck und die Funktionalität einer automatisch ablaufenden Datenübertragung. Die Kontrolle über die laufenden Interrupt-Transaktionen wird jedoch nicht mehr von der Hardware, sondern von der im Hintergrund arbeitenden Software übernommen.

Isochronous-Pipes wiederum lassen sich – da ohnehin ohne Bandbreitengarantie gearbeitet wird – nach dem Best-Effort-Prinzip behandeln. Es entfällt lediglich das Senden eines ZLPs, wenn die Applikation Daten nicht schnell genug bereitstellt.

Das Adressierungsproblem lässt sich also durch geeignete Programmierung lösen. Insofern ist ein Mehrgerätebetrieb möglich, ohne dass auf verschiedene Pipetypen verzichtet werden muss.

4.2.1.2 Fehlende Präambel für Lowspeed-Pakete

Als Einschränkung erweist sich die Tatsache, dass der AT90USB1287 keine Möglichkeit zum Senden einer Präambel für Lowspeed-Pakete bietet. Diese Präambel wird unbedingt benötigt, um einen – im Mehrgerätebetrieb ja zwingend vorhandenen – Hub auf das Auftreten einer Lowspeed-Übertragung aufmerksam zu machen (siehe 2.4.3.12). Da die Präambel auch nicht abseits der USB-Funktionalität des Controllers mit vertretbarem Aufwand zu erzeugen ist, hat dies ein Scheitern des Betriebs von Lowspeed-Geräten bei Anschluss an einen Hub zur Folge.

Es kann also immer nur ein Lowspeed-Gerät allein am AT90USB1287 betrieben werden. Für Fullspeed-Geräte ist ein Mehrgerätebetrieb jedoch auch ohne die Präambel möglich, so dass dieser weiterhin anzustreben ist. Kann

[4]Siehe [Dat], Seite 307f.

auch das Problem des Data-Toggles gelöst werden, so steht einem Mehrgerätebetrieb nichts mehr im Wege.

4.2.1.3 Problematik der Data-Toggles

Der AT90USB1287 stellt insgesamt sieben Pipes zur Verfügung, über die frei mit jeweils einem Endpunkt kommuniziert werden kann. Dies ist für die Kommunikation mit nur einem Endgerät in aller Regel auch ausreichend. Beim Anschluss mehrerer Geräte kann es jedoch zu Situationen kommen, in denen die Kommunikation mit mehr als sieben Endpoints notwendig ist. An sich sollte auch das kein Problem darstellen, denn die Zugriffe erfolgen ja nacheinander, womit für die Kommunikation mit verschiedenen Endpunkten jeweils ein und die selbe Pipe genutzt werden könnte. Doch auch hier liegt der Teufel wieder im Detail, denn wie vorangehend beschrieben erwarten die einzelnen Endpunkte der Geräte bei Übertragung von Daten jeweils Pakete mit passender Sequenznummer beziehungsweise passendem Data-Toggle[5].

Folgen somit zwei Transaktionen T_1 und T_2 zwischen Host und einem bestimmten Endpoint aufeinander, so muss das Datenpaket von T_2 mit einem Toggle komplementär zum Datenpaket aus T_1 versehen sein. Dies ist auch dann der Fall, wenn die beiden Transaktionen nicht zu ein und demselben Transfer gehören. Folgen also zwei Transfers aufeinander, so muss das erste Paket des zweiten Transfers mit dem selben Toggle versehen werden, welches auch ein weiteres Paket des vorangegangenen Transfers erhalten hätte. Dabei spielt der zeitliche Abstand zwischen den Transfers oder Transaktionen keine Rolle.

"Teilen" sich mehrere Endpunkte nun eine auf dem AT90USB1287 physisch vorhandene Pipe, so könnte es dazu kommen, dass ein falsches Data-Toggle gesendet beziehungsweise erwartet wird. Ein zwischenzeitlich auf der Pipe ausgeführter Transfer zu einem anderen Endpunkt kann das Toggle der jeweiligen Pipe verändert haben. In diesem Fall kommt es zu einer fehlerhaften Kommunikation, bei der Pakete als Duplikate angesehen und verworfen werden, die in Wirklichkeit Teil einer ordnungsgemäßen Übertragung sind.

Kommunikation über die Default Control Pipe oder zu anderen Control-Endpunkten ist dabei unproblematisch, da die USB-Spezifikation für jeden neuen Control Transfer definierte Toggles vorschreibt. Diese Regeln werden von der Hardware des AT90USB1287 ohne Zutun der Software beachtet. Somit lässt sich die Pipe P_0 problemlos für Control Transfers auch mit mehreren Geräten nutzen.

[5]Sieht man von Isochronous-Endpoints ab, die mit beliebigem Toggle versorgt werden können.

Für Transaktionen, welche nicht über die Default Control Pipe laufen, liegt der Gedanke nahe, das Toggle einfach vor dem Paketversand geeignet zu setzen. Die ist jedoch leider zum Scheitern verurteilt, da die Hardware hierzu schlicht keine Möglichkeit bietet. Ein Rücksetzen der Sequenznummer auf Null ist zwar möglich, aber die Fähigkeit zum Setzen auf Eins sucht man vergeblich. Somit muss auf anderem Weg dafür Sorge getragen werden, dass bei Kommunikation mit einem Endpoint immer jeweils eine Pipe des passenden Toggles verwendet wird.

Eine einfache Möglichkeit ist es, jedem Endpunkt, mit dem kommuniziert wird, fest genau eine der Hardware Pipes zuzuweisen. Auf diese Weise teilen sich niemals zwei Endpunkte eine Pipe und korrekte Sequenznummern sind immer gewährleistet. Dies schränkt den Betrieb etwas ein, denn es kann immer nur mit maximal sechs Nicht-Control-Endpunkten gleichzeitig Kommunikation betrieben werden.

Theoretisch ließe sich die Situation noch etwas verbessern, indem bei jedem Transfer das letzte aufgetretene Toggle gespeichert wird. Das aktuelle Toggle einer Pipes ist ist über Register UPSTAX zugreifbar. Beim nächsten Transfer zum selben Endpunkt wird dann eine "passende" Pipe gesucht, indem das momentane Toggle der Hardware-Pipe mit dem für den Transfer benötigten Toggle verglichen wird. Hiermit ist das Finden einer passenden Pipe zwar nicht immer garantiert, je nach Transferverlauf wäre jedoch auch eine Kommunikation mit mehr als sechs Nicht-Control-Endpunkten möglich.

Leider ergab ein Versuch jedoch, dass das über das Register UPSTAX angegebene Toggle nicht immer korrekt ist. Intern behandelte der AT90USB-1287 zwar alle Pakete mit einer korrekten Sequenznummer, ein Auslesen von UPSTAX ergab jedoch einen fehlerhaften Wert. Damit ist die Suche einer passenden Pipe nicht möglich und es muss auf die starre Zuordnung von Pipes und Endpunkten zurückgegriffen werden.

An dieser Stelle sei jedoch explizit gesagt, dass dies prinzipiell nicht die Zahl der Geräte einschränkt, welche gleichzeitig betrieben werden können. Eine Kommunikation über die Default Control Pipe und weitere Message Pipes ist immer für alle Geräte möglich. Lediglich die Zahl der zusätzlich nutzbaren Endpunkte ist auf sechs zur gleichen Zeit eingeschränkt.

Wie die Betrachtung der Probleme zeigt, unterliegt der Mehrgerätebetrieb mit einem AT90USB1287 zwar gewissen Einschränkungen, ist prinzipiell jedoch möglich. Daher wird die Entwicklung der Software im Folgenden auf den Mehrgerätebetrieb ausgerichtet.

4.2.2 Grundfunktionalität

4.2.2.1 Geräteverwaltung

Funktionen und Strukturen Der nun anzustrebende Mehrgerätebetrieb beeinflusst auch grundlegend die von der Software zu Verfügung gestellten Fähigkeiten für den späteren Anwendungs- und Treiberentwickler. Um den Entwickler nicht mit der Problematik der Toggles zu belasten, muss bei den Kommunikationsvorgängen weitestgehend von den einzelnen, hardwareseitigen Pipes abstrahiert werden. Hierzu muss auf unterster Ebene Funktionalität geboten werden, die das Zuweisen der Pipes für den Gerätezugriff transparent bewerkstelligt.

Auch sollte sich der Anwender später nicht um die Einzelheiten des Adresssetups für Geräte oder ähnliches kümmern müssen. Für solche Aufgaben sind geeignete Funktionen zur Verfügung zu stellen, die die mitunter komplexen Requests kapseln.

Geräte sind in geeigneten Strukturen zu verwalten, so dass für den Programmierer ein einfacher Zugriff auf die Geräteeigenschaften – vor allem also auf die Descriptoren – gewährleistet ist. Insbesondere können besagte Descriptoren bei USB eine im Voraus unbekannte Länge aufweisen, so dass zur Laufzeit eine entsprechende Speicherverwaltung betrieben werden muss.

Geräteerkennung und Hubs Etwas früher ansetzend gibt es noch einen weiteren Punkt zu beachten. Der Vorgang der Erkennung von angeschlossenen Geräten ist geeignet zu verbergen. Dem Nutzer ist dann die Information über das An- und Abstecken zugänglich zu machen. Bei dieser Überlegung ist auch mit einzubeziehen, dass zwischen dem Einstecken eines Gerätes direkt an die vom Experimentierboard zur Verfügung gestellte Buchse – im Folgenden Root-Port genannt – und dem Einstecken an einen Hub unterschieden werden muss. Das Einstecken und Abziehen von Geräten am Root-Port kann über die Hardware des AT90USB1287 selbst festgestellt werden. Für ein Detektieren solcher Vorgänge an einem Hub ist jedoch die Programmierung eines zusätzlichen Hubtreibers notwendig. Dieser Treiber macht sich wiederum auch die von der Basisbibliothek zur Verfügung gestellten Funktionen selbst zu Nutze – Ein Hub ist ja im Sinne von USB ebenfalls ein Gerät, welches eine Adresse und Kommunikationskanäle benötigt.

Funktionen wie die Vergabe einer Adresse via "Set Address"-Request sollen dabei vollkommen unabhängig von der Frage sein, wo sich ein Gerät in der Bustopologie befindet. Der für den Mehrgerätebetrieb nötige Hubtreiber muss also weitestgehend transparent arbeiten[6].

[6] Weitestgehend daher, da sich ohne Hubtreiber ja auch der Mehrgerätebetrieb verbie-

4.2.2.2 Aufbau und Nomenklatur

Kommunikationsfunktionen Was die konkrete Realisierung der Funktionen zur Kommunikation angeht, so bietet sich eine – zumindest entfernte – Anlehnung an das System der BSD-Sockets an. Durch den Bekanntheitsgrad des Socket-APIs würden somit auch im Umgang mit USB-Kommunikation unbedarfte Entwickler eine verhältnismäßig gewohnte Begriffswelt vorfinden.

Natürlich kann es hier nicht um eine genaue Kopie der BSD-Sockets gehen. Die Autoren sind jedoch der Überzeugung, dass alleine durch die gewohnte Namensgebung die Arbeit mit der zu entwickelnden Bibliothek erheblich erleichtert wird.

Zur Assoziation von Hardware-Pipes und Endpunkten der Geräte wird im Folgenden also eine Socket-Struktur dienen.

Konstanten und Strukturen Die USB-Spezifikation beschreibt viele Descriptoren und Konstanten. Diese sind geeignet in C-Quelltext zu überführen. Genau diese Arbeit haben die Entwickler des Linux-Kernels schon vorgenommen, weshalb sich eine Übernahme deren Ergebnisse anbietet. Diesem Vorgehen liegt die Überlegung zugrunde, dass eine Portierung von Gerätetreibern von Code aus dem Linux-Kernel in Richtung AVR-Controller durch Verwendung der Kernel-Definitionen entscheidend vereinfacht wird. Die fraglichen Elemente befinden sich allesamt in Kapitel 9 der USB-Spezifikation und wurden daher von den Linux-Entwicklern im Kernelheader *usb_ch9.h* definiert. Im Falle der Descriptoren kommt `typedef struct` zum Einsatz, bei den Konstanten wurde mit der Präprozessor-Direktive `#define` gearbeitet. Besagter Kernelheader wird – leicht eingekürzt – für die weitere Entwicklung übernommen. Nicht alle gemachten Definitionen werden im Folgenden unbedingt auch durch die Bibliothek direkt genutzt. Sollten sie jedoch einmal durch einen Anwender benötigt werden, so sind sie ohne weiteren Aufwand zugänglich.

4.2.3 Dokumentation des Quellcodes

Als Dokumentationssystem bietet sich das weit verbreitete `doxygen` an. Die Software wird im Quelltext entsprechend kommentiert. Auf diese Weise steht dem zukünftigen Programmierer eine komfortabel nutzbare Quellendokumentation zur Verfügung.

tet, was die Applikation etwas entlasten kann. Bei Betrieb mit dem Hubtreiber muss es aber gleichgültig sein, ob ein Gerät in den Port eines Hubs oder direkt in den Root-Port eingesteckt wird.

4.2.4 Ausgabe und Stringfunktionen

Da der Umgang mit zeilen- beziehungsweise zeichenweiser Ausgabe sich bei Mikrocontrollern etwas vom gewohnten Vorgehen unterscheidet, ist sich auch hier über die zu verwendenden Funktionen klarzuwerden.

Zur Vereinfachung der Ausgabe von Zeichenfolgen stellt die avr-libc einige Hilfe zur Verfügung. So existiert eine Implementierung der bekannten Funktion printf(), um Zeichenketten wie gewohnt formatiert auf einen Stream namens *stdout* schreiben zu können. Die Version der libc für den AVR-Controller sieht jedoch keine vorkonfigurierten Ein- oder Ausgabe-Streams vor, wie dies für andere Systeme der Fall ist. Für *stdout* muss der Programmierer selbst eine Funktion implementieren, welche auf den Stream geschriebene Zeichen geeignet verarbeitet. Üblicherweise wird diese Funktion eintreffende Zeichen über eine geeignete Hardwareeinheit des Controllers an einen externen Rechner versenden. Dies könnte beispielsweise die UART oder auch das TWI/I²C-Interface sein. Der Programmierer hat ferner dafür zu sorgen, dass die verwendete Hardwareeinheit vor der Nutzung des Streams korrekt konfiguriert ist.

Für diese Arbeit werden alle Ausgaben von Zeichenketten über die UART des Controllers an einen PC weitergegeben, der dann die Daten über ein Terminalprogramm oder auch mittels `cat /dev/ttyS0` darstellen kann. Die Konfiguration der UART kann mit Hilfe der Funktion USART_Init() erledigt werden, die in der Datei `usart.c` implemtiert wurde. In `usart.c` ist auch uart_putchar() definiert, welche das Versenden von Zeichen über die USART übernimmt. USART_Init() sorgt ferner dafür, die Ausgabefunktion korrekt an den Stream *stdout* zu binden. Damit sind keine weiteren Aktionen von Seiten des Programmierers notwendig, um printf() und ähnliche Funktionen zu nutzen.

Bei einem Blick in den Quellcode wird auch die Nutzung einer Funktion namens printf_P() anstelle des einfachen printf() auffallen. Hintergrund dabei ist das Vorgehen des C-Compilers, Zeichenketten grundsätzlich als variabel anzusehen und daher zu Programmanfang in den Arbeitsspeicher des Controllers kopieren zu lassen. printf() erwartet entsprechend eine Zeichenkette im RAM als Formatstring. Handelt es sich bei der Zeichenkette aber um einen Konstante, wie dies bei den meisten Formatstrings der Fall ist, so kann mit Hilfe des avr-libc-Makros PSTR() eine Zeichenfolge explizit als im Programmspeicher vorzuhalten markiert werden. Dies spart große Mengen an kostbarem RAM und ist daher im Allgemeinen sinnvoll. Zur Verwendung solcher Strings aus dem Flashspeicher des Controllers ist dann nicht mehr printf(), sondern printf_P() zu nutzen.

Die avr-libc bietet noch eine ganze Reihe weiterer Stringfunktionen mit

dem Suffix _P. Diese verhalten sich allesamt wie ihr Pendant ohne besagtes Suffix, erwarten jedoch den fraglichen String im Programmspeicher statt im RAM.

4.3 Einstiegsbeispiel

Um einen besseren Eindruck von der Nutzung der Bibliothek zu vermitteln, wird an dieser Stelle mit einem konkreten Nutzungsbeispiel eingestiegen. Listing 4.1 zeigt den entsprechenden Programmcode. Die Arbeitsweise der einzelnen Funktionen wird darauf folgend erläutert.

Auf den ersten Blick mag der Quelltext als einfaches Beispiel überfrachtet wirken, jedoch sollte sich bereits aufgrund der Benennung der Funktionen, Variablen und Typen ein gewisses Grundverständnis für die Funktionalität des Codes einstellen.

Initialisierung In Zeile 14 befindet sich die bereits erwähnte Funktion USART_Init(), die die UART initialisiert. Direkt darauf erfolgt eine Ausgabe, anhand derer festgemacht werden kann, ob die Datenübertragung zum Rechner hin erfolgreich ist.

Danach folgt in Zeile 17 das Bereitmachen der USB-Einheit des AT90USB1287. Die Funktion usb_host_init() übernimmt die Initialisierungsaufgaben insoweit, dass nach ihrem Aufruf auf das Anschließen von Geräten gewartet werden kann[7].

Reaktion auf das Einstecken eines Gerätes Nun kommt die eigentliche Applikation zum Zuge. Über root_dev_connected() (Zeile 23) kann in Erfahrung gebracht werden, ob gerade ein neues Gerät an den Root-Port des AVR angesteckt wurde. Ist dies der Fall, so wird in den folgenden Zeilen die programmtechnische Repräsentation eines Gerätes erzeugt. Das Gerät wird sogleich mit Hilfe der Funktion setup_device() konfiguriert. Dabei wird dem Gerät eine Adresse zugewiesen und seine Descriptoren in den Speicher des AT90USB1287 geladen. Der Vorgang bis hierhin entspricht in etwa der USB Bus Enumeration, wobei aber kein Hub zum Einsatz kommt.

Nach dem Gerätesetup wird über socket_create() "auf gut Glück" eine Verbindung zu Endpunkt 1 in Interface 0 des Gerätes hergestellt. Der Endpunkt wird, soweit es das weitere Programm angeht, anhand seines Descriptors identifiziert. Der Descriptor wiederum ist über device_get_endpoint_descriptor() zugänglich.

[7] Siehe dazu auch Abschnitt 3.6.4.2.

Listing 4.1: Quelltext einer einfachen Anwendung

```
#include "core/config.h"
#include "supplement/usart.h"
#include "host/usb_host.h"
#include <stdio.h>
#include <stdlib.h>

//Main programm
int main (void){

  usb_device *dev = NULL;
  int socket;
  usb_endpoint_descriptor *endpoint;

  USART_Init();
  printf_P(PSTR("USART configured properly.\r\n"));

  usb_host_init();

  //the main loop.
  while (1){

//---- handle connects ----
    if(root_dev_connected()){
      printf_P(PSTR("device connected to the rootport!\r\n"));
      dev = device_create();
      setup_device(dev);
      endpoint = device_get_endpoint_descriptor(dev, 0, 1);
      socket = socket_create(dev, endpoint);
    }

//---- handle disconnects ----
    if(root_dev_disconnected()){
      printf_P(PSTR("device disconnected from the rootport!\r\n"));
      cleanup_disconnected_device(dev);
      dev = NULL;
    }

//---- let's try some communication ----
    if (dev) {
      char data[8];
      int bytes = usb_recv(socket, data, 8);
      if (bytes>0){
        printf_P(PSTR("%d bytes: "), bytes);
        for (int i=0; i<bytes; i++) printf_P(PSTR("0x%02x "), data[i]);
          printf_P(PSTR("\r\n"));
        }
    }

  }//end loop
}//end main
```

Damit hat das Programm seine unmittelbare Reaktion auf das Anschließen eines Gerätes abgeschlossen.

Reaktion auf das Abziehen eines Gerätes Ähnlich wie die Funktion root_dev_connected() verhält sich auch root_dev_disconnected() in Zeile 32, jedoch wird hierüber das Abziehen eines Gerätes festgestellt. Entsprechend muss die Applikation dann auf das Ereignis reagieren. In diesem Beispiel lässt sie die Bibliothek das Aufräumen der nun nicht mehr benötigten Speicherstrukturen für das Gerät erledigen. Hierzu ruft die Applikation die Funktion cleanup_disconnected_device() mit der Repräsentation des zu bereinigenden Geräts auf. Die Funktion sorgt gleichzeitig neben der Speicherbereinigung auch für die Beseitigung aller eventuell noch offenen Sockets, die mit dem Gerät in Verbindung stehen. Der beim Anschließen geöffnete Socket wird also beseitigt, so dass durch ihn im Weiteren keine Probleme entstehen können. Dann setzt die Applikation den Gerätezeiger auf NULL zurück, da der Zeiger hier auch als einfacher Marker für das Vorhandensein eines Gerätes dient.

Die eigentliche Kommunikation Nun kommt es endlich zur ersten einfachen Kommunikation mit dem Endgerät. In der Hoffnung, dass der gewählte Endpunkt vom Typ IN ist und Daten liefert[8], werden acht Byte über die Funktion usb_recv() angefordert. Hat die Funktion Daten abgerufen, so gibt sie die fragliche Anzahl als Returnwert zurück. Die Daten werden sogleich mit einer einfachen **for**-Schleife und printf_P() ausgegeben.

Programmtest Um die Funktion des Programms kurz zu testen, genügt der Anschluss einer einfachen USB-Maus an das Experimentierboard. Nach Erkennen des Gerätes wird jede Bewegung der Maus und jeder Tastendruck mit einer Ausgabe entsprechender Daten quittiert. Abbildung 4.1 zeigt einen beispielhaften Durchlauf des Codes.

Nach dem Einstecken wurde die linke Maustaste gedrückt und dann losgelassen (Zeile 4/5), die Maus ein wenig bewegt (Zeile 6) und schließlich das Mausrad nach vorn (7) und hinten (8) bewegt.

Anschließend wurde die Maus wieder vom Port des Experimentierboards abgezogen.

Als Randbemerkung sei gesagt, dass der Ablauf des Programms immer einen sicheren Zugriff auf alle im Speicher vorhandenen Strukturen gewährleistet. Speicherallokationen oder Deallokationen werden niemals von der Bibliothek direkt in Reaktion auf das An- oder Abstecken eines Gerätes durchgeführt. Die Bibliothek handhabt zwar die Speicherverwaltung intern, die

[8] Es fand ja keine Überprüfung der Descriptoren beim Erstellen des Sockets statt.

```
~ $ cat /dev/ttyS0
USART configured properly.
device connected to the rootport!
4 bytes: 0x01 0x00 0x00 0x00
4 bytes: 0x00 0x00 0x00 0x00
4 bytes: 0x00 0xff 0x00 0x00
4 bytes: 0x00 0x00 0x00 0x01
4 bytes: 0x00 0x00 0x00 0xff
device disconnected from the rootport!
```

Abbildung 4.1: Mögliche Ausgabe des Einstiegsbeispiels

Applikation stößt solche Vorgänge jedoch immer explizit durch Funktionen wie cleanup_disconnected_device() an. Auf diese Weise wird verhindert, dass Speicherbereiche für Descriptoren oder ähnliches unbemerkt von der Applikation freigegeben werden, sobald ein Gerät abgezogen wird. Würde darauf nicht geachtet, so würde es unweigerlich zu Speicherzugriffsfehlern kommen.

Das geschilderte Vorgehen überträgt aber auch der Applikation die Verantwortung, die Aufräumprozeduren bei Feststellen des Abziehens tatsächlich anzustoßen. Tut sie dies nicht, so bleiben Bus-Adressen und Speicherplatz im Hintergrund belegt, was mit der Zeit die immer knappen Ressourcen des μC aufbraucht.

Die nächsten Abschnitte beschreiben näher die Funktion und Konfiguration der Bibliothek. Ferner soll im Folgenden die entwickelte Software auch einen Namen erhalten – ob ihrer Funktionalität und Zielplattform wird sie hiermit *libAT90USB* getauft.

4.4 Aufteilung und Konfiguration

Um etwas Übersicht in die Menge der beteiligten Quelldateien zu bringen, wurde unterhalb des Stammverzeichnisses von *libAT90USB* diverse Unterverzeichnisse angelegt.

Verzeichnis `core/`

config.h Das zunächst wichtigste Verzeichnis ist *core*. Hier befindet sich die Datei `config.h`, über die der Programmierer generelle Einstellungen zur verwendeten Platform vornehmen muss.

Listing 4.2: Konfiguration der CPU-Taktfrequenz

```
1 #ifndef F_CPU
2 #define F_CPU            16000000
3 #endif //F_CPU
4
5 #if (!((F_CPU==16000000)||(F_CPU==8000000)))
6 #    error      The clock speed you selected will not result in a functional
  USB-interface. Edit config.h or your makefile to set a correct value.
7 #endif //F_CPU
```

Hierzu zählt vor allem die Angabe der Taktrate des Controllers. Der in Listing 4.2 gezeigte Code erledigt dabei direkt auch eine Prüfung der Taktfrequenz, welche für ein funktionsfähiges USB-Interface entweder 8 oder 16MHz betragen muss. Alternativ kann die Taktrate über F_CPU auch direkt durch die CFLAGS beim Compilen übergeben werten.

usb.h Ebenfalls in `core/` zu finden ist die Datei `usb.h`. Hier befinden sich zum einen zusätzliche Definitionen und Bitmasken für die Handhabung der Register des AT90USB1287, welche die avr-libc nicht mitliefert. Auch Fehldefinitionen, wie sie in der zu Anfang dieser Arbeit aktuellen Version der avr-libc enthalten waren, wurden hier behoben. Zum anderen ist dort aber auch ein funktionsartiges Makro definiert, welches zur Konfiguration der Taktgebereinheit des USB-Controllers dient. Listing 4.3 zeigt das Makro.

Listing 4.3: Konfiguration der PLL für die Taktversorgung der USB-Einheit

```
1 #if (F_CPU == 8000000)
2 #   define PLL_SETTING           (1<<PLLP1)|(1<<PLLP0)
3 #else
4 #   define PLL_SETTING           (1<<PLLP2)|(1<<PLLP0)
5 #endif
6
7 #define config_pll()                                                  \
8     PLLCSR = PLL_SETTING;         /* Configure PLL interface */\
9     PLLCSR |= (1<<PLLE);          /*  Enable PLL */              \
10    loop_until_bit_is_set(PLLCSR,PLOCK);  /* Check PLL lock */
```

Die Konfiguration der PLL wird, wie durch die Registerbeschreibung zu PLLCSR in Abschnitt 3.6.3.3 angegeben, in Abhängigkeit der vorher einge-

stellten Taktfrequenz F_CPU vorgenommen.

`usb.h` sollte nicht direkt durch #include in die spätere Applikation eingebunden werden. Der Gedanke dabei ist, dass die enthaltenen Definitionen durchaus auch für eine Programmierung des AT90USB1287 im Devicemodus brauchbar wären. Dies ist zwar nicht Teil dieser Arbeit, aber diese Option wurde durch eine entsprechende Einteilung der Sourcen schon vorbereitet. Auf diese Weise dürfte eine Implementierung der für den Device-Betrieb fehlenden Funktionen auf Basis dieser Arbeit erleichtert werden. Für den Hostmodus bindet der Applikationsentwickler daher die später beschriebene Datei `usb_host.h` als Header ein, die ihrerseits die benötigten Informationen aus `usb.h` übernimmt.

usb_ch9.h und errorcodes.h Die Umsetzung der Definitionen und Deklarationen zu USB aus dem Linux-Kernel findet sich, wie bei diesem auch, in einer Datei namens `usb_ch9.h`. Hier sind die von der USB-Spezifikation gemachten Deskriptoren und Konstanten in Strukturen und Definitionen festgehalten worden. Die Namensgebung ist dabei weitestgehend sprechender Natur, so dass die Handhabung verhältnismäßig einfach fällt.

Eine globale Fehlercodeliste wurde für das Projekt in `errorcodes.h` abgelegt. Die von *libAT90USB* gebotenen Funktionen liefern im Fehlerfall verschiedene, negative Fehlercodes zurück, die der Aufrufer auswerten kann. Um die teilweise durch mehrere Funktionsaufrufe durchgereichten Fehlercodes beziehungsweise deren Quelle identifizieren zu können, wird über die verwendeten Codes in `errorcodes.h` Liste geführt und diese auch sogleich für doxygen aufbereitet.

4.4.1 Verzeichnis `host/`

Während in `core/` eher allgemeine Definitionen, die allesamt prinzipiell auch für den Betrieb im Devicemodus interessant wären, abgelegt sind, befinden sich unter `host` solche Dateien, die direkt für den Hostmodus genutzt werden.

usb_host.h Zu diesen Dateien zählt insbesondere die `usb_host.h`, welche direkt in die zukünftigen Applikationen eingebunden wird. Neben der Deklaration von diversen Funktionen zur Verwaltung und Kommunikation von und mit USB-Geräten, finden sich hier auch noch einige Einstellungen, die der Programmierer seinen Wünschen entsprechend anpassen kann. Listing 4.4 zeigt die drei wichtigsten Definitionen, die geändert werden können.

Ein Setzen von __USE_UVCONE hat zur Folge, dass die Spannungsversorgung des USB über den Pin *UVCONE* geschaltet wird. Wurde ein Aufbau

Listing 4.4: Konfiguration für *UVCONE*, maximale Zahl der möglichen Geräte und Anzahl der Sockets in `usb_host.h`

```
1  /**
2   * If __USE_UVCONE is set, the AT90USB's UVCON-Pin will be used to
3   * enable/disable vbus-power.
4   * Undef this to be able to use that pin as PINE7.
5   */
6  #define __USE_UVCONE
7
8  /**
9   * Maximum number of devices that can be connected
10  * to the avr at a time. Limits the number of available
11  * addresses, may not exceed 127.
12  */
13 #define USB_MAX_DEVICES 8
14
15 /**
16  * \name Maximum number of sockets.
17  * Use this to configure the maximum number of sockets that
18  * can be managed by libAT90USB simultaneously.
19  **/
20 #define USB_MAX_SOCKETS 6
```

mit nicht-schaltbarem VBus erstellt, so kann durch Aufheben dieser Definition der Pin *UVCONE* von der USB-Funktionalität befreit und als Standard-I/O-Pin *E7* genutzt werden.

USB_MAX_DEVICES definiert die maximale Anzahl von Geräten, die gleichzeitig mit *libAT90USB* vewaltet werden können. Dies wird später zum Beispiel für die Adressvergabe benötigt, welche die Bibliothek automatisch handhabt.

Entsprechend spezifiziert USB_MAX_SOCKETS die maximale Anzahl der Sockets, die gleichzeitig durch *libAT90USB* verwaltet werden können. Mehr als sechs Sockets machen hier aufgrund der durch die Data-Toggles verursachten Probleme keinen Sinn. Wie schon gesagt, bleibt die Anzahl der bedienbaren Geräte von der Anzahl der Sockets prinzipiell unberührt. Listing 4.5 zeigt eine weitere Einstellung, welche den Timer des AT90USB1287 betrifft. Dieser wird zu der schon angesprochenen Simulation der Interrupt-Pipes sowie auch für die später noch beschriebene Geräteanschlusserkennung benötigt. Über die Konstante TIMER_FREQ kann die Frequenz, mit welcher Timer-Interrupts auftreten, konfiguriert werden. So ist beispielsweise die zeitliche Auflösung, mit der Interrupt-Pipes abgefragt werden können, einstellbar.

Bei dem genutzten Prescaler-Wert von 1024 wird der Zähler $\frac{F\_CPU}{1024}$ mal

Listing 4.5: Konfiguration der Konstanten für die Timerkonfiguration

```
1  /**
2   * This constant can be used to configure the frequency of Timer0-
3   * interrupts. A higher frequency means the possibility to check
4   * interrupt pipes more often.
5   *
6   * Default frequency is 100Hz, meaning 1 overflow every 0.01sec.
7   **/
8  #define TIMER_FREQ 100
9
10 #define COUNTS ((F_CPU/1024)/TIMER_FREQ)
11 #if (COUNTS<1)
12 #   warning The TIMER_FREQ you selected is too high. TIMER_FREQ will be (
   F_CPU/1024).
13 #   undef COUNTS
14 #   define COUNTS 1
15 #else
16 # if (COUNTS>256)
17 #   warning The TIMER_FREQ you selected is too low. TIMER_FREQ will be (F_CPU
   /(1024*256)).
18 #   undef COUNTS
19 #   define COUNTS 256
20 # endif
21 #endif
22 #define TIMER_OVF (256 - COUNTS)
```

pro Sekunde inkrementiert. Die Anzahl der Zählschritte, nach denen jeweils ein Interrupt auftreten soll, lässt sich also mit $\frac{F\_CPU/1024}{TIMER\_FREQ}$ berechnen. Der ausgerechnete Wert wird dann von 256 subtrahiert, da der Interrupt ja bei einem Überlauf – also dem Umspringen des Zählers von 255 auf 0 – ausgelöst wird.

Die restlichen Präprozessorbefehle fangen lediglich den Fehlerfall ab, in dem eine Frequenz selektiert wurde, welche der AT90USB1287 auf Basis des 8Bit-Zählers und der eingestellten Taktfrequenz nicht einhalten kann. Dabei wird beim Compilen eine Warnung ausgeben und der Zählwerte auf den Wert gesetzt, der der gewünschten Frequenz am ehesten entspricht. Der Timer wird so immer nur mit "validen" Werten zwischen 0 und 255 versorgt.

Sonstige Dateien in `host/` Die weiteren in `host/` abgelegten Dateien haben für die Konfiguration keine weitere Bewandnis. In ihnen ist Funktionalität implementiert, welche später beschrieben wird.

4.4.2 Verzeichnis `device/`

Hier können zukünftig die für den Devicemodus relevanten Dateien gelagert werden. Von Konfigurationseite aus gesehen ist das Verzeichnis damit natürlich uninteressant für den Hostmodus.

4.4.3 Verzeichnis `supplement/`

Hier befinden sich einige Dateien, welche nicht direkt im Rahmen dieser Arbeit erstellt wurden, aber Funktionalität bereit stellen, welche für später gezeigte Beispielprogramme benötigt wird. Hierzu gehören unter anderem die Dateien `usart.h` und `usart.c`, die wie vorangehend beschrieben den Zugriff auf die USART vereinfachen.

4.4.4 Weitere Verzeichnisse

In weiteren Verzeichnissen wurden – hier nur der Vollständigkeit halber erwähnt – Beispieltreiber abgelegt. Diese werden später noch erläutert.

4.5 Funktionalität der Grundbibliothek

Nach der Übersicht über Aufteilung und Konfigurationsmöglichkeiten wird im Folgenden die Funktionalität der Grundbibliothek erläutert. Dazu zählen Initialisierung, die grundlegenden Kommunikationsfunktionen und deren Realisierung. Auch die Erkennung des An- und Absteckens von Geräten am Root-Port sowie die Weitergabe solcher Ereignisse an die Applikation zählt zu den Basisaufgaben der Bibliothek. Ein grober Umriss eines Teils dieser Funktionen ist schon durch das Einstiegsbeispiel gegeben worden – nun gilt es, Detailinformationen folgen zu lassen.

4.5.1 Initialisierung der USB-Einheit

`usb_host_init()` Die Initialisierung des Host-Controllers soll durch die Funktion usb_host_init() aus `usb_host.c` erledigt werden. Diese ist entsprechend in `host.h` deklariert und steht der Applikation nach Einbinden des Headers zur Verfügung. Nach Ablauf von usb_host_init() ist der erste Teil des in Abschnitt 3.6.4.2 beschriebenen Betriebszyklus abgeschlossen und der Controller kann auf den Anschluss von Geräten warten. Das folgende Listing zeigt die Funktion.

```
1  void usb_host_init(){
2
3    // Configure the PLL interface
4    config_pll();
5
6    //take care of the host-settings
7    usb_host_powerup();
8
9    //enable 8bit timer with 1024 prescaling
10   TCCR0B = (1<<CS02)|(1<<CS00);
11   TIMSK0 = (1<<TOIE0);
12
13   //enable interrupts
14   sei();
15 }
```

Mittels des schon gezeigten Makros config_pll() stellt usb_host_init() zunächst die Taktversorgung für den USB-Controller sicher. Anschließend bedient sie sich der Hilfsfunktion usb_host_powerup() zum Einschalten der USB-Einheit. Danach ist nur noch der 8Bit-Timer zu konfigurieren und das Global Interrupt Enable Flag zu setzen. Ein großer Teil der Funktionalität wird offensichtlich von usb_host_powerup() übernommen.

usb_host_powerup() und vbus_power() In der Tat erledigt diese Funktion – zusammen mit vbus_power() – einen großen Teil der Arbeit, wie das nachfolgende Listing zeigt.

```
1  void vbus_power(char poweron){
2
3    if (poweron){
4
5      //enable VBus
6      USBCON &= ~(1<<FRZCLK);
7      OTGCON |= (1<<VBUSREQ);
8      USBCON |= (1<<FRZCLK);
9      //wait for a valid vbus-supply and reset all interruptflags
10     loop_until_bit_is_set(USBSTA, VBUS);
11     UHINT = 0;
12     USBINT = 0;
13
14   } else {
15     //just turn the VBus off
16     OTGCON |= (1<<VBUSRQC);
17   }
18 }
19
20 void usb_host_powerup(void){
21
22   // Power-on USB pads regulator
23   UHWCON = (1<<UVREGE);
24
```

```
25    // Enable USB interface
26    USBCON = (1<<USBE);
27    UDCON &= ~(1<<DETACH);
28
29    // Configure USB interface
30    // we need OTGPAD for the vbus-sensing.
31    USBCON |= (1<<HOST)|(1<<OTGPADE);
32    UHCON = 0;
33
34  #ifdef __USE_UVCONE
35    //Use PE7 as UVCON
36    UHWCON |= (1<<UVCONE);
37  #else
38    //don't allow the hardware to control the vbus
39    OTGCON |= (1<<VBUSHWC);
40  #endif
41    // it's not very intuitive, but we have to do this even
42    // if the hardware doesn't control UVCON.
43    // we won't get any vbus- or d+/d--sensing without it.
44    vbus_poweron();
45
46  #ifdef __USE_UVCONE
47    //give potentially connected devices a few moments to power up.
48    for(int i=0; i<10; i++) _delay_ms(10);
49  #endif
50
51    //enable interrupts
52    UHIEN  = (1<<RSTE) | (1<<DDISCE) | (1<<DCONNE);
53    OTGIEN = (1<<VBERRE);
54
55    //start the usb-controller's clock.
56    //the controller begins to check for a device and will set
57    //DCONNI/BERRI/VBERRI according to its findings
58    USBCON &= ~(1<<FRZCLK);
59  }
```

Durch vbus_power wird, in Abhängigkeit des Parameters **char** poweron, der VBus an- oder abgeschaltet. Ist poweron logisch Eins, so wird die Spannungsversorgung für den VBus wie in 3.6.3.5 beschrieben angefordert. Anderenfalls wird der VBus abgeschaltet.

Wie die Kommentare zeigen, hat usb_host_init() einen Teil der nach Abschnitt 3.6.4.2 anfallenden Aufgaben an usb_host_powerup() delegiert. Der Pin *UVCON* wird in Abhängigkeit des Wertes von __USE_UVCONE konfiguriert.

Interessant ist dabei noch die Tatsache, dass im Falle eines schaltbaren VBus – also bei gesetztem __USE_UVCONE – nach jeder Aktivierung des VBus durch Zeile 48 für 100ms gewartet wird. Dies verschafft etwaigen bereits angeschlossenen Geräten etwas Zeit, um sich mit der nun zur Verfügung stehenden Stromversorgung vom VBus einzuschalten. Ist der VBus nicht schaltbar, so ist auch dieses Warten nicht nötig, die Geräte sind ja unmittelbar nach ihrem Anschluss mit Spannung versorgt.

Zeile 58 aktiviert letztlich die Taktversorgung der USB-Einheit. Nun versucht der Controller automatisch eine Verbindung zu angeschlossenen Geräten festzustellen.

4.5.2 Geräte-Anschlusserkennung

Es sind nur zwei Fälle denkbar:

1. Es ist ein Geräte angeschlossen
2. Es ist kein Geräte angeschlossen

4.5.2.1 Fall 1: Gerät vorhanden

Betrachten wir zunächst den ersten Fall. Im Folgenden wird also davon ausgegangen, dass ein Gerät mit dem Root-Port des AT90USB1287 verbunden ist.

Die Hardware stellt fest, dass ein Gerät angeschlossen ist und signalisiert dies durch ein Setzen des DCONNI-Bits. Dies löst durch die zuvor gesetzte Interruptkonfiguration einen Interrupt aus. Hierbei wird die Interrupt-Service-Routine ISR(USB_GEN_vect) ausgeführt, die für die Behandlung des USB-Global-Interruptvektors zuständig ist. Folgender Ausschnitt aus besagter ISR ist dabei für die Reaktion auf den Interrupt zuständig.

```
1  /* 8<--------- ISR(USB_GEN_vect) --------- */
2
3    /* D+ and D- of a new device got connected to the port */
4    if ( UHINT & (1<<DCONNI) && UHIEN & (1<<DCONNE) ){
5
6      //make sure the clock is running and begin to send SOFs
7      UHCON |= (1<<SOFEN);
8
9      //reset the device, disable and handshake the interrupt
10     UHCON |= (1<<RESET);
11     UHIEN &= ~(1<<DCONNE);
12     UHINT &= ~(1<<DCONNI);
13   }
14
15 /* --------- ISR(USB_GEN_vect) --------->8 */
```

Die Software beginnt mit dem Senden von Start-Of-Frame-Paketen und löst einen USB-Reset für das neu angeschlossene Gerät aus. Die weitere Interruptbehandlung für DCONNI wird zunächst unterbunden, da bei einigen Geräten bei Anschluss aus unbekannten Gründen mehrmals DCONNI gesetzt

wurde. So wurde die ISR mehrfach hintereinander durch DCONNI angestoßen, was zu einem mehrfachen Versuch des Geräte-Resets und zu Fehlverhalten führte. Nach diesen Aktionen hat die ISR ihre Reaktion auf DCONNI beendet und der Kontrollfluss kehrt zum Hauptprogramm zurück.

Sobald der Gerätereset vollzogen ist, setzt die Hardware RSTI. Dies hat ein erneutes Ausführen von ISR(USB_GEN_vect) zur Folge – Diesmal ist jedoch der nachstehend gezeigte, etwas kompliziertere Teil der ISR für die Behandlung zuständig.

```
1  /* 8<--------- ISR(USB_GEN_vect) --------- */
2
3    /* The device connected to the atmel's port has just been reset */
4    if ( (1<<RSTI) & UHINT && UHIEN & (1<<RSTE) ){
5
6      int status;
7
8      //RSTI disables and resets all pipes, so this is the earliest time that
       the pipes can be initialized
9      status = init_pipe(PIPE_64_0, PSIZE_64, PBK_1) ||
10               init_pipe(PIPE_256, PSIZE_256, PBK_1) ||
11               init_pipe(PIPE_64_1, PSIZE_64, PBK_1) ||
12               init_pipe(PIPE_64_2, PSIZE_64, PBK_1) ||
13               init_pipe(PIPE_64_3, PSIZE_64, PBK_1) ||
14               init_pipe(PIPE_64_4, PSIZE_64, PBK_1) ||
15               init_pipe(PIPE_64_5, PSIZE_64, PBK_1);
16
17     if ( status ) {
18       //in case the pipe creation process fails, bail out and retry.
19       USBCON &= ~(1<<USBE);
20       usb_host_powerup();
21       return;
22     }
23
24     //guarantee a small amount of time for the device so it can recover from
       the reset
25     _delay_ms(2);
26
27     //inform the application
28     root_device_state |= CONNECT_EVENT;
29
30     //handshake and disable the interrupt
31     UHIEN &= ~(1<<RSTE);
32     UHINT &= ~(1<<RSTI);
33   }
34
35 /* --------- ISR(USB_GEN_vect) ---------->8 */
```

Zunächst werden alle Pipes mittels der Funktion init_pipe() initialisiert. Diese übernimmt einen Teil des Pipe-Aktivierungsprozesses, soweit er die Aktivierung und Speicherallokation angeht. Sie beschreibt also UPCFG1X und setzt das ALLOC-Bit in UPCONX. Um die Parameterübergabe zu erleich-

tern, können Definitionen verwendet werden, die entsprechend in `usb_host.h` angelegt wurden. init_pipe() gibt bei Erfolg Null zurück, im Fehlerfall wird nachfolgend der USB-Controller neu gestartet und dann auf Erfolg gehofft.

libAT90USB initialisiert alle Pipes am Anfang der USB-Aktivität mit statischer Größe. Später werden nur noch die Pipe-Tokens und der Pipe-Typ geändert. Auf diese Weise wird ein fehleranfälliges Speichermanagement für die Pipes vermieden, was in Anbetracht der vom AT90USB1287 zur Verfügung gestellten Speicherverwaltung (siehe Abschnitt 3.6.5.1) angebracht erscheint. Die Änderung der von init_pipe() nicht gesetzten Parameter erfolgt später durch den Aufruf einer Funktion namens setup_pipe() beziehungsweise implizit beim Senden oder Empfangen. Damit ist der gesamte Pipe-Aktivierungsprozesses abgedeckt.

Nachdem die Pipes initialisiert wurden, ist sowohl der AT90USB1287 als auch das angeschlossene Gerät betriebsbereit und die ersten Standard Device Requests können über die Default Control Pipe übertragen werden. Es gilt nun nur noch, die Applikation über den Anschluss des Geräts zu informieren, damit diese die entsprechenden Kommunikationsvorgänge anstoßen kann. Diese Benachrichtigung geschieht mit Hilfe von Zeile 28.

Zur Informationsaufbewahrung wird die Variable root_device_state genutzt. Die Applikation kann das Anschlussereignis dann – wie schon im Anfangsbeispiel gesehen – mittels root_dev_connected() erfragen. Zur Erklärung dieser Funktionalität hier die für das Vorgehen relevanten Definitionen:

```
1  #define CONNECT_EVENT         0x01
2  #define DISCONNECT_EVENT      0x02
3  #define APP_STATE_CONNECTED 0x04
4  volatile unsigned char root_device_state = 0;
5
6  unsigned char root_dev_connected(){
7    unsigned char tmp_status = SREG;
8    cli();
9    unsigned char return_value =  (root_device_state & CONNECT_EVENT) &&
10                                 !(root_device_state & APP_STATE_CONNECTED);
11
12   if (return_value){
13     //clear the event
14     root_device_state &= ~(CONNECT_EVENT);
15     //from now on, we need to think of the app as in state: connected
16     root_device_state |= APP_STATE_CONNECTED;
17   }
18
19   SREG = tmp_status;
20   return return_value;
21 }
```

Wird aus dem normalen Programmfluss heraus auf einer Variablen ope-

riert, welche auch aus dem Kontext eines Interrupts heraus modifiziert werden kann, so ist der entsprechende Variablenzugriff geignet gegen sich überschneidende Zugriffe zu sichern. root_dev_connected() übernimmt gerade die Rolle dieser Zugriffssynchronisation auf die wegen des Interruptzugriffs folgerichtig als **volatile** deklarierte Variable root_dev_state. Daher sichert sie ihr Vorgehen entsprechend durch Speichern des Statusregisters SREG – und damit des Global-Interrupt-Enable Flags – und anschließendem Unterdrücken der Interrupts ab.

Gleichzeitig setzt sie ein eventuell vorhandenes CONNECT_EVENT-Flag in root_dev_state auch zurück und vermerkt die Tatsache, dass die Applikation von dem nun angeschlossenen Gerät erfahren hat. Diese Vorgänge müssen ebenfalls atomar geschehen, damit der Variablenzustand korrekt bleibt. Über das APP_STATE_CONNECTED-Flag kann die Bibliothek beispielsweise bei einem Abziehen eines Gerätes feststellen, ob die Applikation überhaupt auf den Geräteanschluss reagiert hat. Anderenfalls muss diese auch nicht weiter über das Abziehen informiert werden.

Am Ende ihrer Arbeit stellt root_dev_connected() dann den ursprünglichen Zustand des Interrupt-Enable-Bits wieder her. Durch das Sichern und Zurückschreiben des Statusregisters kann die Funktion sowohl aus unterbrechbarem als auch aus nicht-unterbrechbarem Kontext heraus aufgerufen werden, ohne an diesem Kontext etwas zu ändern.

Fall 1 ist damit abgeschlossen. Ein angeschlossenes Gerät wurde erkannt und zurückgesetzt.

4.5.2.2 Fall 2: kein Gerät vorhanden

Ist bei Anlaufen der Taktversorgung der USB-Einheit kein Gerät angeschlossen, so stellt der Controller fest, dass keine Verbindungsaufnahme möglich ist. Als Reaktion darauf setzt die Hardware das BCERRI-Bit, um den Verbindungsfehler anzuzeigen.

Um die Interrupt-Behandlung einfach zu halten, wird auf das BCERRI-Flag hin jedoch kein Interrupt ausgelöst. Wie in der Registerbeschreibung gesehen, müsste für das Auslösen eines Interrupts durch BCERRI zusätzlich auch der VBUSTI-Interrupt aktiviert werden, was die zugehörige ISR unnötig kompliziert gestaltet.

Das Überprüfen der Fehlerfälle erfolgt stattdessen in der ISR, die den Overflow des zuvor in usb_host_init() konfigurierten Timer0 behandelt. Der Timer wird ohnehin für die Simulation der Interrupt-Pipes benötigt und kann die Fehlerfälle problemlos ebenfalls handhaben.

Das Gerüst der ISR(TIMER0_OVF_vect) sowie die für die Behandlung des Fehlschlagens der Geräteverbindung benötigte Teilfunktion sind im fol-

genden Listing zu sehen.

```
1  ISR(TIMER0_OVF_vect){
2
3    //used to keep track of the bconnection-timeout
4    static char poll_counter = 0;
5
6  /* 8<--------
7  [...]
8  -------->8 */
9
10   if ( (1<<BCERRI) & OTGINT ){
11
12     //check for the 1.5sec-timeout (default) to retry to connect
13     if ( poll_counter == 150 ){
14
15       //timeout hit, reset the controller
16       poll_counter = 0;
17       USBCON &= ~(1<<USBE);
18       usb_host_powerup();
19
20     } else {
21       //timeout not hit yet, so increment the counter
22       poll_counter++;
23     }
24   }
25
26 /* 8<--------
27  [...]
28 -------->8 */
29
30   //reset the Timer to the overflow-constant
31   TCNT0 = TIMER_OVF;
32 }
```

Standardmäßig tritt der Overflow des Timers nach 10ms auf, was durch die Ausgangskonfiguration von TIMER_FREQ festgelegt wird[9]. Nach jedem Überlauf wird der Timer wieder auf diesen Wert gesetzt, so dass sich der nächste Überlauf nach der gleichen Zeit einstellt. Somit ist eine gleichbleibende Periode der Overflow-Ereignisse sichergestellt.

Stellt die Service-Routine nun ein gesetztes BCERRI fest, so wird zuerst mittels Hochzählen der Variablen poll_counter eine Pause eingelegt. Die Hardware hat beim Auftreten von BCERRI den VBus abgeschaltet und den USB-Controller "eingefroren". Nach Ablauf einer Frist (standardmäßig 1,5s) setzt die ISR die USB-Einheit zurück. Dies geschieht durch Abschalten der Einheit und anschließendem Aufruf von usb_host_powerup(). Danach beginnt die Hardware also erneut, nach einem angeschlossenen Gerät zu suchen.

Ist im erneuten Durchlauf ein Gerät vorzufinden, spielt sich der Vorgang

[9]Siehe 4.5

wie in Fall 1 beschrieben weiter ab. Findet der Controller immer noch kein Gerät, so setzt er erneut BCERRI und der Vorgang wiederholt sich bis zum Anschluss eines Gerätes.

4.5.3 Erkennen des Abziehens eines Gerätes

Das Entfernen eines Gerätes vom Root-Port des AT90USB1287 quittiert die Hardware mit dem Setzen des DISCI-Flags. Entsprechend der durch usb_host_init() vorgenommenen Konfiguration wird hierdurch abermals die ISR(USB_GEN_vect) ausgelöst. Der relevante Teil der Routine ist im Folgenden dargestellt.

```
1  /* 8<--------- ISR(USB_GEN_vect) --------- */
2
3    /* The (root) device has been disconnected from the atmel's port */
4    if ( UHINT & (1<<DDISCI) && UHIEN & (1<<DDISCE) ){
5
6      //freeze all pipes
7      for (int i=0; i<7; i++) {
8          select_pipe(i);
9          freeze_pipe();
10     }
11
12     //stop sending SOFs
13     UHCON &= ~(1<<SOFEN);
14
15     //distribute the information to the app
16     root_device_state |= DISCONNECT_EVENT;
17
18     //handshake and disable the interrupt
19     UHIEN &= ~(1<<DDISCE);
20     UHINT &= ~(1<<DDISCI);
21   }
22
23 /* --------- ISR(USB_GEN_vect) ---------->8 */
```

Dieser Teilausschnitt der ISR übernimmt eine Funktionalität, die praktisch komplementär zu der ist, welche bei Anschluss und Reset eines Geräts auftritt. Die Requestgenerierung aller Pipes wird unterbunden und das Versenden von SOFs oder Keep-Alives eingestellt. Der Applikation wird das Verschwinden des Geräts mit Hilfe von root_device_state angezeigt. Diese kann die Information durch die ebenfalls schon im Einstiegsbeispiel gesehene Funktion root_dev_disconnected() abfragen und quittieren. Die Nutzung ist dabei analog zur vorangehend gezeigten Funktion root_dev_connected().

4.5.4 Verhalten bei Fehlern der Spannungsversorgung des VBus

Zusätzlich zu dem "normalen" Abziehen eines Gerätes kann noch ein weiterer Umstand dazu führen, dass ein Gerät seine Funktionalität nicht mehr weiter zur Verfügung stellt. Dieser Umstand ist der Verlust der Stromversorgung des Gerätes, also bei buspowered Geräten genau dann, wenn die VBus-Spannung unerwartet einbricht.

Auch diesen Fehler zeigt der AT90USB1287 an und zwar – wie in der Registerbeschreibung gezeigt – durch Setzen des VBERRI-Bits. Die Abfrage dieses Flags erfolgt durch die schon erwähnte Interrupt-Service-Routine für den Timer0-Überlauf.

```
1  /* 8<---------- ISR(TIMER0_OVF_vect) ---------- */
2
3  //check for a VBUS error, this takes priority.
4    if ( (1<<VBERRI) & OTGINT ){
5
6      //check if we need to inform the app about the loss of the connected
       device
7      if( root_device_state&APP_STATE_CONNECTED )
8          root_device_state |= DISCONNECT_EVENT;
9
10     //clear pending connect-events
11     root_device_state &= ~CONNECT_EVENT;
12
13     //reset the controller
14     USBCON &= ~(1<<USBE);
15     usb_host_powerup();
16
17   }
18
19 /* ---------- ISR(TIMER0_OVF_vect) ---------->8 */
```

Wie zu sehen ist, führt ein Fehler in der Versorgungsspannung ebenfalls zu einem Rücksetzen der USB-Einheit. Im Gegensatz zum Fall des Verbindungsfehlers ist hier jedoch natürlich keine Wartezeit vorgesehen. Außerdem entscheidet die Routine im VBERRI-Fall, inwiefern die Applikation über den Verlust des angeschlossenen Gerätes zu informieren ist. Dies geschieht nur dann, wenn die Applikation das Gerät im Status "verbunden" führt. Etwaige noch nicht durch die Anwendung quittierte Verbindungs-Ereignisse können an dieser Stelle dann verworfen werden, da das gesamte USB-System zurückgesetzt wird.

Damit ist die Funktionalität zur Initialisierung und Geräteerkennung vollständig beschrieben. Der nun folgende Teil kann sich mit der Kommunikation

und den dafür benötigten Funktionen beschäftigen.

4.5.5 Basis-Kommunikationsfunktionen

Zunächst müssen Kommunikationsfunktionen auf unterster Ebene implementiert werden. Sie regeln das unmittelbare Lesen und Schreiben auf Pipes – und damit auf den Bus – durch direkten Zugriff auf die entsprechenden Register des AT90USB1287.

4.5.5.1 Identifikation der benötigten Funktionen

Betrachtet man die auf dem USB grundsätzlich vorkommenden, verschieden Fälle von Kommunikation, so ergeben sich die folgenden Feststellungen:

- Die unmittelbare Datenübertragung mittels Pipes der Typen Bulk, Interrupt und Isochronous kann – sofern es die Belange der Software betrifft – weitestgehend identisch verlaufen. Zu unterscheiden ist nur zwischen lesendem und schreibendem Zugriff auf den Bus, welche natürlich unterschiedliches Vorgehen verlangen.
- Transfers über Control-Pipes bestehen potentiell aus drei, mindestens jedoch aus zwei Stufen[10]. Die Setup-Stufe folgt einem identischen Prozedere, gleich welcher Richtung der Control Transfer ist. Die Daten- sowie auch die Status-Stufe sind je nach Richtung unterschiedlich.
- Für die Daten- sowie auch für die Status-Stufe gelten im Rahmen der Software Regeln, wie sie auch für die Übertragung von Daten über Pipes anderer Typen gelten.

Dementsprechend wurden in `usb_host.c` vier verschiedene Funktionen entworfen, die jeweils einen Teil der nötigen Lowlevel-Funktionalität bereit stellen:

1. bus_read_data(), die Daten auf unterster Ebene vom Bus anfordert und entgegennimmt. Sie ermöglicht lesenden Zugriff auf alle Pipetypen.
2. bus_write_data(), die das schreibende Pendant zu bus_read_data() darstellt.
3. ctrl_setup_stage(), welche die Setup-Stufe bei Datenübertragung auf Control-Pipes abwickelt.

[10] Siehe hierzu auch 2.4.4.4 und 3.6.6.5.

4. ctrl_transfer(), welche die Funktionalität der vorangehend genannten Funktionen nutzt, um einen kompletten Control Transfer – also das Hintereinanderreihen aller Stufen – zu ermöglichen.

Ob ihrer Wichtigkeit werden diese Funktionen im Folgenden einzeln beschrieben und ihr Quellcode im Grundgerüst dargestellt und erläutert.

Für alle diese Funktionen stellt sich im Laufe ihrer Arbeit eine ganz ähnliche Aufgabe: Sie müssen zu ein oder mehreren Gelegenheiten auf das UPINTX-Register zugreifen und prüfen, ob bestimmte Bits dort gesetzt sind. UPINTX signalisiert ja beispielsweise mittels des RXINI-Bits das Eintreffen eines neuen Paketes oder auch mit Hilfe von TXOUTI das Vorhandensein einer freien Bank zum Paketaufbau und anschließendem Versand.

Dabei genügt es jedoch nicht, beispielsweise mit loop_until_bit_is_set() auf das entsprechende Bit "zu warten". Während der Wartezeit könnte es zu verschiedenen Ereignissen kommen, auf die reagiert werden muss und auf deren Auftreten hin das Warten potentiell abgebrochen werden muss. Schließlich könnte es auch sein, dass – wartet man zum Beispiel auf das Freiwerden einer Bank – das Ereignis aus wie auch immer gearteten Gründen niemals eintritt. Für diese Fälle muss also ein entsprechender Timeout implementiert werden.

Für das aktive Warten mit gleichzeitigem Behandeln verschiedener Ereignisse wurde daher eine eigene Funktion geschaffen, um den Funktionen zur Lowlevel-Kommunikation diese Arbeit abzunehmen. Der Quelltext dieser Funktion findet sich in Listing 4.6 wieder.

wait_UPINTX_flag() wartet auf das ihr durch den Parameter **char** flag mitgeteilte Bit in UPINTX. Für flag kommen dabei RXINI, TXOUTI oder TXSTPI in Frage. Während der Wartezeit wird die Anzahl eventuell vom Gerät gesendeter NAKs mitgezählt und das Warten nach einer durch den Parameter **unsigned long** nak_cnt gegebenen Anzahl von NAKs abgebrochen. Der Parameter **unsigned long** timeout arbeitet ähnlich, mit seiner Hilfe wird die Zahl der beim Warten möglichen Schleifendurchläufe (jeweils zwischen zwei eintreffenden Paketen) begrenzt. Dies ist ein recht grober Mechanismus, aber für diesen simplen Anwendungsfall ausreichend. Ferner wird das Warten bei Fehlern auf der Pipe und bei Empfang eines STALL-Handshakes beendet.

Nach dem Feststellen einer Abbruchbedingung wird die Requestgenerierung durch Einfrieren der Pipe unterbunden und dann nochmals auf das Vorhandensein des abzuwartenden Flags geprüft. Auf diese Weise wird verhindert, dass ein Flag fälschlicherweise verpasst wird, weil der entsprechende Request erst nach Feststellen der Abbruchbedingung erfolgreich war. Bei Vorfinden von RXSTALLI ist dieses Vorgehen nicht notwendig, der AT90-USB1287 sorgt automatisch für ein Anhalten der Pipe in diesem Fall. Sollte

Listing 4.6: Die Funktion wait_UPINTX_flag()

```
1  static int wait_UPINTX_flag(char flag, unsigned long timeout, unsigned long
   nak_cnt){
2
3    unsigned long timeout_cnt = timeout;
4    int return_value = 0;
5
6    while ( !(UPINTX & (1<<flag)) ){
7      if( UPINTX & (1<<NAKEDI) ){
8        UPINTX &= ~(1<<NAKEDI);
9        if (nak_cnt-- == 0)          { freeze_pipe(); return_value=ERR_NAK_CNT;
         break; }
10       timeout_cnt = timeout;
11     }
12     if ( UPINTX & (1<<PERRI) )       { freeze_pipe(); return_value=ERR_PIPE;
          break; }
13     if ( timeout_cnt-- == 0 )        { freeze_pipe(); return_value=ERR_TIMEOUT
       ; break; }
14     if ( UPINTX & (1<<RXSTALLI) )    return ERR_STALL; //stall automatically
       freezes the pipe
15   }
16
17   if ( !(UPINTX & (1<<flag)) ){  //double check whether the flag has been set
      in the meantime
18     return return_value;
19   } else {
20     unfreeze_pipe();
21     return 0;
22   }
23 }
```

das Warten auf das gewünschte Flag erfolgreich verlaufen sein, so meldet wait_UPINTX_flag() dies durch Rückgabe einer Null. Im Fehlerfall erhält der Aufrufer einen Fehlercode, mit dessen Hilfe die Ursache des Abbruchs festgestellt werden kann.

4.5.5.2 bus_write_data()

Die Funktion bus_write_data() ermöglicht das Senden von Daten zu einem Gerät über den Bus. Hierzu benötigt sie neben der Kenntnis über die zu versendenden Daten (sowie auch deren Länge) noch das Wissen über die maximale Paketgröße, die der Zielendpunkt verarbeiten kann. Um das Zerlegen der Daten in passende Einzelpakete kümmert sie sich dann selbst.

Zu beachten ist, dass bus_write_data() sich in keiner Weise mit der Adressierung der zu versendenden Daten befasst. Auch die Auswahl, auf welcher der vom AT90USB1287 zur Verfügung gestellten Hardware-Pipes die Kom-

munikation stattfinden soll, überlässt sie ihrem Aufrufer. Die Kommunikation wird auf der zum Aufrufzeitpunkt aktuellen Pipe ablaufen, also auf der Pipe, deren Nummer gerade in UPNUM eingetragen ist.

Als Rückgabe liefert sie die Anzahl der tatsächlich ans Gerät gesendeten Bytes oder – im Falle eines Fehlers – einen negativen Fehlercode an ihren Aufrufer. Listing 4.7 zeigt den wesentlichen Ausschnitt des Quellcodes der Funktion.

Listing 4.7: Die Funktion bus_write_data()

```
1  static int bus_write_data(char *data, unsigned int byte_count, unsigned int
   max_packet_size){
2
3   int waitstatus;
4   unsigned int bytes_left = byte_count;
5
6   //set the tokentype to "out", so that the device will accept our data
7   set_pipe_tokentype(PTOKEN_OUT);
8
9   do {
10    //wait for bank_ready, so that we can begin to send data
11    waitstatus = wait_UPINTX_flag(TXOUTI, TIMEOUT_WRITE_BANKREADY,
      NAKCNT_WRITE_BANKREADY);
12    if (waitstatus<0) return waitstatus - 4;
13    UPINTX &= ~(1<<TXOUTI);                    //handshake bank_ready
14
15    //fill the fifo with a payload for the packet
16    //we need to respect the maximum packetsize as well as the total amount
      of data.
17    for (int btp=min(bytes_left, max_packet_size); btp>0; btp--){
18        UPDATX = *(data++);
19        bytes_left--;
20    }
21
22    //clear FIFOCON to send the packet
23    UPINTX &= ~(1<<FIFOCON);
24  } while ( bytes_left > 0 ); //...repeat if there's more data
25
26  //wait for bank_ready, just to make sure we received the last ack
27  waitstatus = wait_UPINTX_flag(TXOUTI, TIMEOUT_WRITE_LASTACK,
    NAKCNT_WRITE_LASTACK);
28  if (waitstatus<0) return waitstatus - 5;
29
30  return byte_count - bytes_left;
31 }
```

Bei schreibendem Zugriff auf den Bus muss der Controller ein OUT-Token generieren und danach die zu versendenden Daten an ein DATA0- beziehungsweise DATA1-Paket anhängen. Zunächst wird also in Zeile 7 die aktuelle Pipe zum Senden von Tokens des Typs OUT konfiguriert. Das Ma-

kro set_pipe_tokentype() setzt dazu die erforderlichen Bits in UPCFG0X und sorgt auch direkt für die Freigabe (engl. "unfreeze") der Pipe, wodurch die Tokengenerierung möglich wird.

In Zeile 11 kommt die Funktion wait_UPINTX_flag() zum Einsatz. Sie wartet auf eine freie Bank, so dass danach die zu sendenden Daten in den FIFO gebracht werden können. Der eingesetzte Wert für die Anzahl der erlaubten NAKs ist empirischer Natur und ermöglicht eine Differenzierung zwischen temporärem Unvermögen des Endpunkts, Daten entgegen zu nehmen und einem tatsächlichen Ausfall. Ähnlich verhält es sich mit dem Wert für den Timeout. Sollte es bei einem spezifischen Gerät wiederholt zum Abbruch des Sendevorgangs mit entsprechendem Fehlercode kommen, so kann eventuell ein Verändern der in `usb_host.h` definierten Konstanten helfen.

Werden hier und im Folgenden Konstanten von Fehlercodes abgezogen, so dient dies nur der Erzeugung eindeutiger Fehlercodes für alle denkbaren Abbruchpunkte im Programmfluss. Fehlercodes und Konstanten wurden also so angelegt, dass sich für jeden erzeugten Fehlercode mit Hilfe der in `errorcodes.h` enthaltenen Tabelle der genaue Abbruchpunkt im Quellcode identifizieren lässt.

Ist ein Zustand erreicht, in dem eine freie Bank vorhanden ist, werden die zu sendenden Daten durch schreibenden Zugriff auf UPDATX in den FIFO gebracht. Dabei wird sowohl auf die maximal zu sendende Anzahl von Bytes als auch auf die maximal pro Paket mögliche Payload Rücksicht genommen. Insbesondere werden im Falle eines zu sendenden Zero-Length-Packets keinerlei Daten in den FIFO geschrieben. Anschließend sorgt ein Löschen des Bits FIFOCON für den Beginn des Paketversands.

Sollten nach diesem ersten Paket noch weitere Daten zu übertragen sein, so beginnt der Vorgang von neuem. Nach dem letzten Paket wird wiederum auf das Freiwerden der Bank gewartet, denn dies signalisiert auch den korrekten Eingang des letzten Pakets beim Gerät. Die Bank wird vom AT90-USB1287 ja erst bei Empfang eines ACKs wieder als frei markiert.

Nun kann die Anzahl an Bytes, die zum Gerät übertragen wurden, an den Aufrufer zurückgegeben werden.

4.5.5.3 bus_read_data()

Den Gegenpart zu bus_write_data() übernimmt bus_read_data(). Auch sie kümmert sich nicht um Adressierung oder Pipewahl, sie generiert lediglich IN-Tokens über die aktuelle Pipe und nimmt die daraufhin hoffentlich eintreffenden Datenpakete entgegen. Ist die zu empfangende Anzahl an Bytes gelesen oder signalisiert das USB-Gerät das Ende des aktuellen Transfers durch ein Underlength-Packet, so beendet bus_read_data() ihre Arbeit und

gibt die Anzahl der gelesenen Bytes an den Aufrufer zurück. Listing 4.8 zeigt den Quellcode der Funktion.

Zunächst wird wieder set_pipe_tokentype() genutzt, diesmal jedoch, um die Pipe als IN-Pipe zu konfigurieren. Sogleich startet die Requestgenerierung. Dabei wird von einem gesetztem INMODE-Bit ausgegangen, es finden also solange IN-Requests statt, bis entweder ein Fehler die Pipe zum Halten zwingt, oder die Software diese wieder einfriert.

Ab Zeile 14 wird eine kleinere Unterscheidung notwendig. Handelt es sich bei der aktuellen Pipe um eine solche des Typs "Interrupt", so kann der folgende Aufruf von wait_UPINX_flag() bereits beim ersten Auftreten eines NAKs aussteigen, denn ein durch einen Interrupt-IN-Endpunkt gesendetes NAK signalisiert, dass in diesem Abfrageintervall keine neuen Daten zur Verfügung stehen. Bei anderen Pipes wird eine gewisse – wieder empirisch ermittelte – Anzahl von NAKs toleriert. Mit den eingesetzten Werten waren alle zum Test genutzten Geräte durch die Bus Enumeration zu bringen, ohne dass wiederholte Aufrufe der Sendefunktion notwendig waren. Ein wiederholtes Aufrufen kann jedoch trotzdem einmal nötig sein, wenn auf das Senden von Daten gewartet wird, ein Gerät die Anfrage aber nicht schnell genug beantworten kann. In jedem Fall verhindern Timeout und NAK-Limitation jedoch wieder ein endloses Warten.

Nach dem Rücksetzen von RXINI, welches den Paketeingang signalisiert hat, wird über das Register UPBCX die Anzahl der empfangenen Bytes eingesehen. Ist diese Zahl kleiner als die maximal über den aktuellen Endpunkt lieferbare Bytezahl pro Paket, so ist davon auszugehen, dass es sich um das letzte Paket dieses Transfers handelt. Anschließend werden die empfangenen Daten aus dem FIFO entnommen und in den über den Parameter **char** *data gelieferten Speicherbereich geschrieben.

Ist nach der Schleife die gewünschte Anzahl Bytes gelesen, so kann das Paket – wie schon im Falle der Unterlänge – als letztes Paket betrachtet werden. In Zeile 38 wird die aktuelle Bank als frei markiert, so dass dann ein neues IN-Token gesendet wird. Handelte es sich um das letzte Paket, so ist vor dem Löschen von FIFOCON die Pipe einzufrieren, um ein unerwüschtes überzähliges IN-Token zu unterbinden.

Nach Ablauf der Übertragung wird die Zahl der empfangenen Bytes an den Aufrufer zurückgeliefert.

4.5.5.4 ctrl_setup_stage()

Die Funktion ctrl_setup_stage() übernimmt die in der Setup-Stufe eines Control Transfers nötigen Aufgaben. Dafür sendet sie ein Token des Typs "SETUP" und legt danach einen spezifikationsgemäß aus acht Byte bestehenden

Listing 4.8: Die Funktion bus_read_data()

```
1  static int bus_read_data(char *data, unsigned int bytes_requested, unsigned
   int max_packet_size){
2
3    int waitstatus;
4    char last_packet = 0;
5    unsigned int bytes_read = 0;
6
7    //set the tokentype to "in", the device should now answer to our requests
8    set_pipe_tokentype(PTOKEN_IN);
9
10   do {
11     unsigned int btp;
12
13     //a nak on an interrupt pipe means: no data available this time
14     unsigned long nak_cnt = ((UPCFG0X & PTYPE_MASK) == PTYPE_INTERRUPT) ? 0 :
        NAKCNT_READ_BANKREADY;
15
16     //wait for new data, signaled by hardware through RXINI in UPINTX
17     waitstatus = wait_UPINTX_flag(RXINI, TIMEOUT_READ_BANKREADY, nak_cnt);
18     if (waitstatus==ERR_NAK_CNT && !nak_cnt) return bytes_read;
19     if (waitstatus<0) return waitstatus - 3;
20     UPINTX &= ~(1<<RXINI); //handshake packet_received
21
22     //get the bytecount for this packet
23     //underlength packages mean that this is the last one, no matter how many
        we've seen before
24     btp = UPBCX;
25     if ( btp < max_packet_size ) last_packet = 1;
26
27     //read the packet into the buffer (don't do anything for zero-length)
28     //UPBCX-hardware counting seems to be broken for int-pipes...
29     while ( btp && bytes_read<bytes_requested ){
30         *(data + bytes_read++) = UPDATX;
31         btp--;
32     }
33     //if the number of bytes we've been waiting for is here, mark this packet
        the last one
34     if ( bytes_read >= bytes_requested ) last_packet = 1;
35
36     //get ready for the next packet (bank clear) and put a new token on the
       bus, if needed.
37     if (last_packet) freeze_pipe();
38     UPINTX &= ~(1<<FIFOCON);
39
40   } while ( !last_packet );
41
42   //read completed
43   return bytes_read;
44 }
```

Control-Request auf den Bus. Die Einstellung korrekter Zieladressen wird wie gewohnt einer aufrufenden Funktion überlassen.

Listing 4.9 zeigt den Sourcecode zu ctrl_setup_stage().

Listing 4.9: Die Funktion ctrl_setup_stage()

```
1  static int ctrl_setup_stage(usb_ctrlrequest *ctrl_rqst){
2
3    int i;
4
5    //change the pipe-type to setup
6    set_pipe_tokentype(PTOKEN_SETUP);
7
8    //Wait for an empty bank which can be filled with data
9    i = wait_UPINTX_flag(TXSTPI, TIMEOUT_SETUP_BANKREADY,
     NAKCNT_SETUP_BANKREADY);  //return in the case of an error
10   if (i<0) return i - 1;
11   UPINTX &= ~(1<<TXSTPI); //handshake bank_ready
12
13   //fill the fifo with the controlrequest (8 bytes) and send the packet by
     clearing FIFOCON
14   for (i=0;i<8;i++){
15     UPDATX =  *((char*)(((int)ctrl_rqst) + i));
16   }
17   UPINTX &= ~(1<<FIFOCON);
18
19   // Wait for empty_pipe/paket_sent, signaled by hardware through TXSTPI in
     UPINTX
20   i = wait_UPINTX_flag(TXSTPI, TIMEOUT_SETUP_ACK, NAKCNT_SETUP_ACK);
21   if (i<0) return i - 2;  //return in the case of an error
22   UPINTX &= ~(1<<TXSTPI); //handshake ack_received
23
24   //Setup-stage completed
25   return 0;
26 }
```

Zur Übergabe der Requestinformationen des Control-Requests besitzt die Funktion ctrl_setup_stage() den Parameter usb_ctrlrequest *ctrl_rqst. Die Definition des Datentyps usb_ctrlrequest ist in Listing 4.10 zu sehen. Aufbau und Feldnamen wurden entsprechend Tabelle 2.10 gestaltet.

Bei ihrem Aufruf stellt ctrl_setup_stage() die aktuelle Pipe zunächst auf den Tokentyp "SETUP" ein. Danach werden – analog zum Vorgehen von bus_write_data() – die Bytes des Control-Requests in den FIFO gebracht. Vorher ist jedoch nicht durch TXOUTI auf ein Freiwerden der Bank zu warten, sondern über TXSTPI. Abschließend wird das Paket versendet und auf das ACK des Gerätes gewartet. Dann ist die Setup-Stufe beendet. Ein erfolgreicher Abschluss wird durch Rückgabe von Null an den Aufrufer gemeldet. Einen Fehlerfall stellt ctrl_setup_stage() jeweils über die Rückgabe von

Listing 4.10: Tydefinition zu usb_ctrlrequest

```
1 typedef struct {
2         unsigned char bRequestType;
3         unsigned char bRequest;
4         unsigned int  wValue;
5         unsigned int  wIndex;
6         unsigned int  wLength;
7 } usb_ctrlrequest;
```

wait_UPINTX_flag() fest und gibt einen entsprechenden Fehlercode an ihren Aufrufer zurück.

4.5.5.5 ctrl_transfer()

Die letzte noch benötigte Funktion für den grundsätzlichen Ablauf von Control Transfers ist ctrl_transfer(). Diese sorgt für den korrekten Durchlauf aller Stufen eines Control Transfers. Auch sie geht von einer bereits durchgeführten Konfiguration von Zielgeräteadresse und Endpunktnummer aus. Als Parameter benötigt sie lediglich den zu versendenden Control-Request, einen Speicherbereich, der als Puffer für den Datenaustausch mit dem Gerät dient sowie die maximal auf dem Zielendpunkt verarbeitbare Paketgröße.

Listing 4.11 kann der Sourcecode entnommen werden.

Zunächst wird ctrl_setup_stage() mit der entsprechenden Control-Request-Struktur aufgerufen. Konnte diese ihre Aufgabe erfolgreich verrichten, so wird der Control-Request genauer untersucht, um zu entscheiden, ob es sich um einen schreibenden oder lesenden Transfer handelt (Zeile 13). Handelt es sich um einen lesenden (Control Read), so kann sofort bus_read_data() zum Lesen der geforderten Datenmenge aufgerufen werden.

Sollte jedoch ein schreibender Zugriff vorliegen, so könnte es sein, dass die an das Gerät zu übermittelnden Informationen schon innerhalb der Control-Request-Struktur vorhanden waren und dementsprechend die Datenphase ausbleibt (No-Data Control). Dieser Umstand kann auf Basis von wLength festgestellt werden. Ist die Daten-Stufe länger als null Byte (Control Write), so wird die geforderte Anzahl Bytes mittels bus_write_data() übertragen.

Verlief die Daten-Stufe erfolgreich beziehungsweise war gar nicht vorhanden, so fehlt zur Komplettierung des Transfers nur noch die Status-Stufe. Diese besteht aus der Übertragung eines ZLP in zur Daten-Stufe umgekehrter Richtung, was wieder durch die Funktionen bus_read_data() oder bus_write_data() übernommen werden kann.

Listing 4.11: Die Funktion ctrl_transfer()

```
1  static int ctrl_transfer(void *data, usb_ctrlrequest *ctrl_rqst, unsigned int
    max_packet_size){
2
3    int return_value = 0;
4    int bytes_done = 0;
5
6    /* ---- Enter Setup Stage ---- */
7
8    return_value = ctrl_setup_stage(ctrl_rqst);
9    if (return_value<0) return return_value;
10
11   /* ---- Enter Data Stage ---- */
12
13   if ((ctrl_rqst->bRequestType & USB_DIR_MASK) == USB_DIR_OUT) {
14     if (ctrl_rqst->wLength) bytes_done = bus_write_data(data, ctrl_rqst->
       wLength, max_packet_size);
15   } else {
16       bytes_done = bus_read_data(data, ctrl_rqst->wLength, max_packet_size);
17   }
18   if (bytes_done<0) return bytes_done;
19
20   /* ---- Enter Status Stage ---- */
21
22   if ((ctrl_rqst->bRequestType & USB_DIR_MASK) == USB_DIR_OUT)
23     return_value = bus_read_data(NULL, 0, max_packet_size);
24   else
25     return_value = bus_write_data(NULL, 0, max_packet_size);
26
27   if (return_value<0) return return_value - 3;
28
29   /* Transfer Complete */
30   return bytes_done;
31 }
```

Sind alle Stufen erfolgreich durchlaufen, so ist nur noch die Zahl der zusätzlich zu den acht Byte des Control-Requests behandelten Bytes an den Aufrufer zurückzugeben. Im Fehlerfall kehrt auch ctrl_transfer() mit einem passenden Fehlercode zurück.

4.5.6 Geräteverwaltung

Die im vorherigen Abschnitt vorgestellten Funktionen zur Lowlevel-Kommunikation werden nicht direkt vom Programmierer genutzt. Stattdessen stehen weitere Funktionen bereit, welche auch die Adressierung und Pipezuteilung übernehmen. Diese nutzen wiederum die bereits vorgestellten Funktionen zur Abhandlung der eigentlichen Transfers.

Hierzu wird eine geeignete Abbildung der physischen Geräte in eine Programmstruktur benötigt. Ein USB-Gerät wird beschrieben durch seinen Device Descriptor, seine aktuelle Konfiguration sowie seine Adresse. Die Konfiguration ihrerseits kann der Bytefolge entnommen werden, die das Gerät bei Abfrage eines Configuration Descriptors sendet. Hierbei werden alle für die Konfiguration relevanten Descriptoren direkt mitgeliefert. Diese können zum Beispiel Interfaces oder Endpunkte beschreiben, aber auch klassenspezifischer Natur sein.

Die benötigten Daten werden in einer Struktur des Typs usb_device gespeichert. Listing 4.12 zeigt die entsprechende Typdefinition.

Listing 4.12: Typdefinition zu usb_device

```
1 typedef struct {
2     unsigned char address;
3     usb_device_descriptor* dev_descriptor;
4     usb_config_descriptor* config_descriptor;
5 } usb_device;
```

Angemerkt sei, dass config_descriptor dabei nicht nur auf den eigentlichen Configuration Descriptor, sondern damit auch auf den Anfang der vom Gerät gelieferten Beschreibung der Konfiguration als ganzes zeigt.

Der Applikationsprogrammierer wird jedoch nicht nur auf den Configuration Descriptor, sondern auch auf die darin enthaltenen restlichen Descriptoren zugreifen wollen. Hierzu ist jeweils eine Suche innerhalb des Configuration Descriptors notwendig, die den Anfang des fraglichen Descriptors innerhalb der Konfiguration findet.

Es ist sinnvoll, diese Sucharbeit, die bei jedem Zugriff auf einen Descriptor zu leisten ist, durch Bereitstellung geeigneter Funktionen vor dem Programmierer zu verbergen. Für die Applikation werden dann die Descriptoren aller Interfaces und Endpunkte direkt auf Basis von usb_device zugreifbar.

Um Inkonsistenzen im Umgang mit dem oben vorgestellten Typ zu vermeiden, wurde der Abruf aller Eigenschaften eines Geräts durch Funktionen gelöst. Auf diese Weise sind die tatsächlich in usb_device vorhandenen Komponenten verborgen und der Programmierer muss sich nicht weiter um den Aufbau der Struktur kümmern.

Alle Zugriffsfunktionen erhalten die jeweilige Device-Struktur als ersten Parameter und liefern die erfragten Daten an den Aufrufer zurück. Somit entsteht eine quasi-objektorientierte Kapselung für Zugriffe auf den Geräte-

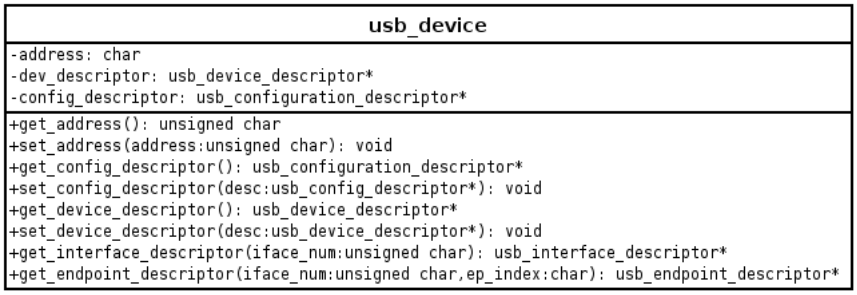


Abbildung 4.2: usb_device in UML-Klassendarstellung.

typ. In diesem Rahmen betrachtet würde eine UML-Darstellung der Klasse usb_device wie in Abbildung 4.2 aussehen.

Zur Implementierung in C wurde der Typ usb_device zusammen mit den darauf operierenden Funktionen in die Dateien `usb_device_struct.h` sowie `usb_device_struct.c` ausgelagert. Die Funktionsnamen wurden jeweils mit dem Präfix device_ ergänzt, um deren Zugehörigkeit hervorzuheben.

Bei den Funktionen, welche ausschließlich Komponenten des usb_device-Typs setzen oder zurückliefern, handelt es sich um einfachste Getter- beziehungsweise Setter-Funktionen. Diese werden folglich hier nicht näher beschrieben. Interessanter sind jedoch die Funktionen get_interface_descriptor() und get_endpoint_descriptor(), da beide tatsächlich Parsingarbeit verrichten müssen. Beide suchen, innerhalb der beim Abrufen des Configuration Descriptors erhaltenen Bytefolge, nach dem Descriptor, den der Aufrufer durch Angabe der entsprechenden Indizes anfordert. Jeder Descriptor ist mit einem Header versehen, der sowohl den Descriptortyp als auch die Descriptorlänge in Bytes angibt. Daher ist es möglich, vom Anfang eines Descriptors innerhalb der Bytefolge auf die Anfangsadresse des nächsten zu schließen und so alle Descriptoren zu durchlaufen. Ferner ist durch die Beschreibung der Descriptoren ab Abschnitt 2.4.5.1 festgelegt, in welcher Abfolge Descriptoren bestimmter Typen in der Bytefolge vorhanden sein können.

Endpunkt Descriptoren eines gewissen Interfaces können beispielsweise nur zwischen dem Descriptor des fraglichen Interfaces und dem Descriptor des darauf folgenden Interfaces vorkommen. Eventuell begrenzt auch die Gesamtlänge der Bytefolge den Suchraum. Im aktuellen Beispiel ist dies der Fall, wenn es um das Auffinden eines Endpunkt Descriptors im letzten Interface geht.

Das Finden eines Descriptors eines bestimmten Typs innerhalb bekannter Grenzen wird von einer Hilfsfunktion übernommen. Listing 4.13 zeigt den Quellcode dieser Funktion. Sie steht im Übrigen später auch Applikationen

oder Treibern zur Verfügung, um beispielsweise klassenspezifische Descriptoren leichter auffinden zu können.

Listing 4.13: Typdefinition zu usb_device

```
1  usb_descriptor_header* get_next_descriptor(usb_device* dev,
   usb_descriptor_header* desc, char search_type, char abort_type){
2
3    usb_config_descriptor* conf_desc = device_get_config_descriptor(dev);
4    int max_address = (int)conf_desc + conf_desc->wTotalLength;
5
6    while (1) {
7      desc = (usb_descriptor_header*) ((int)desc + desc->bLength);
8
9      if ( (int)desc >= max_address ) return (usb_descriptor_header*)ERR_INDEX;
10     if ( desc->bDescriptorType == abort_type) return (usb_descriptor_header*)
       ERR_INDEX;
11     if ( desc->bDescriptorType == search_type) return desc;
12   }
13 }
```

Die Funktion get_next_descriptor() findet den nächsten Descriptor des Typs search_type in der Beschreibung des Gerätes, welcher auf die Position des Descriptors desc folgt. Dabei kann zusätzlich noch ein (die Suche begrenzender) Descriptortyp abort_type angegeben werden, ab dessen Auftreten die Suche als fehlgeschlagen angesehen werden muss. Ferner achtet get_next_descriptor() darauf, nicht über das Beschreibungsende der Konfiguration hinaus nach dem geforderten Descriptor zu suchen, um Fehler im Speicherzugriff zu vermeiden. Mit Hilfe von get_next_descriptor() können die beiden noch fehlenden Funktionen get_interface_descriptor() sowie get_endpoint_descriptor() einfach implementiert werden. Beide Funktionen finden sich in 4.14 wieder.

Wie in device_get_interface_descriptor() zu sehen, werden nur solche Interface Descriptoren beachtet, deren Eintrag für bAlternateSetting Null lautet. Alternative Interfaces werden also ignoriert. Die Funktion zum Finden von Endpunkt Descriptoren arbeitet ganz ähnlich, es ist jedoch zu beachten, dass die Indizierung für Endpunkte mit dem Index Eins beginnt.

Um die Kapselung der Gerätestruktur zu vervollständigen, sind noch zwei weitere Funktionen notwendig. Diese übernehmen Konstruktion und Destruktion eines Gerätes. Um eine Gerätestruktur zu erstellen, soll immer die Funktion device_create() genutzt werden. Sie alloziert Speicher für eine Instanz des Typs usb_device sowie für Device- und Configuration Descriptor. Ferner trägt sie auch den kleinstmöglichen Wert für die maximale Paketgrö-

Listing 4.14: Die Funktionen device_get_interface_descriptor() und device_get_endpoint_descriptor()

```
1  usb_interface_descriptor* device_get_interface_descriptor(usb_device* dev,
   unsigned char iface_num){
2
3    usb_descriptor_header* desc = (usb_descriptor_header*)
     device_get_config_descriptor(dev);
4
5    for (int i=0; i<=iface_num; ){
6      desc = get_next_descriptor(dev, desc, USB_DT_INTERFACE, USB_DT_CONFIG);
7      if ( (int)desc<0 ) return (usb_interface_descriptor*)desc;
8
9      if (((usb_interface_descriptor*)desc)->bAlternateSetting == 0) i++;
10   }
11
12   return (usb_interface_descriptor*)desc;
13 }
14
15
16 usb_endpoint_descriptor* device_get_endpoint_descriptor(usb_device* dev,
   unsigned char iface_num, char ep_index){
17
18   usb_descriptor_header* desc = (usb_descriptor_header*)
     device_get_interface_descriptor(dev, iface_num);
19   if ( (int)desc<0 ) return (usb_endpoint_descriptor*)desc;
20
21   for (int i=0; i<ep_index; i++){
22     desc = get_next_descriptor(dev, desc, USB_DT_ENDPOINT, USB_DT_INTERFACE);
23     if ( (int)desc<0 ) return (usb_endpoint_descriptor*)desc;
24   }
25
26   return (usb_endpoint_descriptor*)desc;
27 }
```

ße der Default Control Pipe von acht Byte ein. Damit ist eine erfolgreiche Erstkommunikation mit dem physischen Gerät, welches durch die Struktur repräsentiert wird, möglich.

Für die Destruktion eines Gerätes ist die Funktion device_destroy() zu nutzen. Sie übernimmt den Gegenpart zu device_create() und befreit den, durch die Gerätestruktur und deren anhängenden Descriptoren belegten, Speicherplatz. Den Quellcode hierzu zeigt Listing 4.15.

4.5.7 Socketkonzept

Die später vom Programmierer genutzten Funktionen zur Kommunikation sollen nun nicht jedesmal mit dem Zielgerät, -Interface und -Endpunkt als Parameter aufgerufen werden. Dies wäre zum einen aus Sicht des Program-

Listing 4.15: Die Funktionen device_create() und device_destroy()

```
1  usb_device* device_create(){
2
3    usb_device* return_value = (usb_device*)calloc(1, sizeof(usb_device));
4    device_set_device_descriptor(return_value, calloc(1, sizeof(
     usb_device_descriptor)));
5    //fill in default value for the ctrl-ep
6    device_get_device_descriptor(return_value)->bMaxPacketSize0 = 8;
7    device_set_config_descriptor(return_value, calloc(1, sizeof(
     usb_config_descriptor)));
8
9    return return_value;
10 }
11
12 void device_destroy(usb_device* dev){
13   free(device_get_device_descriptor(dev));
14   free(device_get_config_descriptor(dev));
15   free(dev);
16 }
```

mierers unpraktisch, zum anderen wird aber auch noch ein Konstrukt zur "Buchführung" über aktuelle Verbindungen zu Geräten benötigt. Dabei ist zum Beispiel auch an die noch zu realisierende Unterstützung von Interrupt-Pipes zu denken. Aber auch an die Frage, welche der Hardware-Pipes die nächste Kommunikation mit einem speziellen Endpunkt übernimmt, kann nur durch entsprechende Verbindungsverwaltung geklärt werden.

Das hierfür genutzte Konstrukt wird im Rahmen von *libAT90USB* als Socket bezeichnet. Die wesentlichen für die Sockets benötigten Teile des Quellcodes wurden in den Dateien `usb_socket.c` und `usb_socket.h` ausgelagert. Eine entsprechende Typdefinition, sowie die Deklarationen der nachstehend beschriebenen Funktionen, findet sich in Listing 4.16.

Grundsätzlich speichert eine Socketstruktur Zielgerät und Endpunkt für Transfers. Die dritte Komponente dient zur Simulation der Interrupt-Pipes und wird später noch erläutert.

Sockets werden global in einem Array der Größe USB_MAX_SOCKETS verwaltet, womit jedem Socket eine einmalige Nummer – nämlich der durch ihn genutze Index – zueigen ist. Ferner ist über den Socket auch die Zuteilung der Hardware-Pipes zu den verschiedenen Endpunkten beziehungsweise Transfers gekapselt. Die Auswahl der zu nutzenden Pipe gestaltet sich gegenwärtig durch die Problematik der Sequenznummern als relativ simpler Vorgang. Jedem Socket wird einfach die Pipe mit der Nummer seines um eins inkrementierten Indizes zugewiesen. Damit ist eine Exklusivnutzung der

Listing 4.16: Typdefinition für usb_socket

```
1  typedef struct {
2      usb_device *device;
3      usb_endpoint_descriptor *ep_desc;
4      virtual_interrupt_pipe *interrupt_pipe;
5  } usb_socket;
6
7  int socket_create(usb_device *device, usb_endpoint_descriptor *ep_desc);
8  int socket_destroy(int socket_number);
9
10 usb_socket* get_socket_by_number(int socknum);
11 int get_suitable_pipe(int socket_number);
```

Pipe durch Transfers über eben diesen einen Socket gewährleistet. In logischer Konsequenz soll zu jedem spezifischen Endpunkt auch immer nur maximal ein Socket geöffnet werden.

Für kommende Controller-Typen sind jedoch auch komplexere Auswahlmethoden denkbar. Dazu könnte beispielsweise auch die Socketstruktur um das letzte verwendete Toggle oder ähnliches erweitert werden.

Zur Erzeugung und Entsorgung von Sockets stehen die beiden Funktionen socket_create() und socket_destroy() zur Verfügung. socket_create() reserviert Speicher für eine Instanz des Typs usb_socket und schreibt eine Referenz auf die erstellte Instanz in das zur Verwaltung dienende Array. Sollte aufgrund der maximal verwaltbaren Anzahl von Sockets kein Index mehr frei sein, so kehrt sie mit einem Fehlercode zurück. Im Erfolgsfall bringt sie auch gleich die zu dem Socket gehörende Pipe sicher in einen Zustand, durch den die erste Kommunikation möglich wird. Dies geschieht durch Ab- und anschließendes Wiederanschalten der Pipe, wodurch alle Statusflags zurückgesetzt werden.

Sockets werden zu ihrem Erstellungszeitpunkt sofort mit Zielgerät und Endpunkt versorgt, weshalb socket_create() die entsprechenden Parameter aufweist. Als Rückgabewert an den Aufrufer liefert sie die Nummer, die den Socket fortan identifiziert. socket_destroy() befreit entsprechend den belegten Speicher und den im Array belegten Index. Der Quellcode der Funktionen kann der beiligenden CD entnommen werden.

Interessanter für die fortlaufende Beschreibung sind die beiden Funktionen get_socket_by_number() sowie get_suitable_pipe(). Erstere liefert zu einem als Parameter gegebenen Integer einen Zeiger auf die Socketstruktur zurück, die durch den fraglichen Integer identifiziert wird. Dabei achtet die Funktion auf eine korrekte Bereichseinschränkung, um fehlerhaften Speicherzugriff zu vermeiden.

Die zweitgenannte Funktion – also get_suitable_pipe – liefert zu einem (durch seine Nummer identifizierten) Socket eine verwendbare Pipe für die nächste Übertragung. Sie implementiert also das weiter oben beschriebene Verhalten und gibt den um eins inkrementierten Index des Sockets zurück.

Wie schon erwähnt ermöglicht eine nächste Generation der Controller eventuell eine andere, freiere Verwendung der Pipes. Sollte dies der Fall sein, so kann der Zuteilungsalgorithmus lokal in der Datei `usb_socket.c` verändert werden. Weitere zusätzlich benötigte Informationen, wie die letzte genutzte Sequenznummer oder ähnliches, sind der Socketstruktur hinzuzufügen.

4.5.8 Höhere Kommunikationsfunktionen

Mit den bisher betrachteten Funktionen kann die Arbeitsweise der höheren, direkt durch den Programmierer zu verwendenden Kommunikationsfunktionen verstanden werden. Dies sind die Funktionen usb_send(), usb_recv() und usb_control_transfer(). Die ersten beiden dienen zum einfachen Senden und Empfangen von Daten über Stream Pipes, die letzte führt Control-Transfers beziehungsweise Kommunikation über Message Pipes durch.

4.5.8.1 usb_send() und usb_recv()

Die beiden Funktionen usb_send() und usb_recv() haben – von der Übertragungsrichtung abgesehen – eine identische Aufgabe zu erledigen. Beide müssen auf Basis des als Parameter zu übergebenden Sockets das Gerät, mit dem kommuniziert werden soll, identifizieren und dementsprechend das Adressregister UHADDR mit der Geräteadresse beschreiben. Zusätzlich ist eine für den folgenden Transfer geeignete Pipe zu wählen und deren Endpunktadresse und Typ einzustellen. Abhängig von der Transferrichtung muss dann entweder bus_read_data() oder bus_write_data() aufgerufen werden, um die eigentliche Datenübertragung abzuwickeln. Natürlich sind in beiden Fällen auch die Ziel- beziehungsweise Quelladresse der Daten, sowie deren Byteanzahl bekannt.

Die in Listing 4.17 gezeigte Funktion do_usb_transfer() übernimmt die usb_send() und usb_recv() gemeinsame Funktionalität. Dabei achtet sie darauf, durch Abschalten des Global-Interrupt-Flags ununterbrechbar zu sein, damit kein weiterer Transfer die zwischenzeitlich getätigten Einstellungen zunichte macht. Die Entscheidung über die Richtung des Transfers trifft do_usb_transfer() auf Basis des Parameters direction.

Theoretisch könnte die Richtung des Transfers auch aus dem Descriptor des Zielendpunktes ermittelt werden. Durch die explizite Angabe der

Listing 4.17: Die Hilfsfunktion do_usb_transfer.

```
1  int do_usb_transfer(int socket_number, void *buffer, int bytecount, char
   direction){
2
3    int return_value = 0;
4    usb_socket *socket;
5    int pipe;
6
7    unsigned char tmp_status = SREG;
8    cli();
9
10   //get the socket-pointer
11   socket = get_socket_by_number(socket_number);
12   if ((int)socket < 0){ SREG = tmp_status; return (int)socket; }
13
14   //now find a good pipe for the transfer
15   pipe = get_suitable_pipe(socket_number);
16   if (pipe<0){ SREG = tmp_status; return pipe; }
17
18   //setup the pipe and the deviceaddress register
19   setup_pipe(
20       pipe,
21       socket->ep_desc->bEndpointAddress & USB_ENDPOINT_NUMBER_MASK,
22       (socket->ep_desc->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)<<6
23   );
24   UHADDR = device_get_address(socket->device);
25
26   //now start the transfer
27   return_value = (direction == USB_DIR_IN) ?
28                   bus_read_data(buffer, bytecount, socket->ep_desc->
                     wMaxPacketSize):
29                   bus_write_data(buffer, bytecount, socket->ep_desc->
                     wMaxPacketSize);
30   return_value = return_value<0 ? return_value-6 : return_value;
31
32   freeze_pipe();
33
34   SREG = tmp_status;
35   return return_value;
36 }
```

Übertragungsrichtung wird jedoch ein versehentliches Auslösen einer falschgerichteten Transaktion, die unter Umständen Datenverlust zur Folge haben könnte, vermieden.

Die in Listing 4.18 gezeigten Funktionen usb_send() und usb_recv() dienen dann nur noch als Wrapper, die für eine Einhaltung der Namenskonventionen und für eine Sicherheit der Übertragungsrichtung sorgen.

Listing 4.18: Die Wrapper usb_send() und usb_recv()

```
1  int usb_send(int socket_number, void *buffer, int bytes_to_send){
2
3    //do a transfer and make sure its direction is OUT
4    return do_usb_transfer(socket_number, buffer, bytes_to_send, USB_DIR_OUT);
5  }
6
7  int usb_recv(int socket_number, void *buffer, int bytes_requested){
8
9    //do a transfer and make sure its direction is IN
10   return do_usb_transfer(socket_number, buffer, bytes_requested, USB_DIR_IN);
11 }
```

4.5.8.2 usb_control_transfer()

Die Funktion usb_control_transfer() wickelt Transfers über Control-Pipes ab. Dabei dürfte es sich in der überwiegenden Mehrheit der Fälle also um Kommunikation mit Default-Control-Endpoints handeln.

Dennoch können Control Transfers auch andere Endpunkte zum Ziel haben, weshalb der Funktion neben dem Zielgerät auch noch der Zielendpunkt als Parameter zu übergeben ist. Der Endpunkt wird dabei über seinen Descriptor identifiziert. Da Endpunkt 0 keinen eigenen Descriptor besitzt, ist für Kommunikation mit ihm die Konstante CTRL_EP zu nutzen[11].

Als letzter Paramter wird noch der durchzuführende Control-Request benötigt, der wie bei ctrl_transfer über eine Instanz des Typs usb_ctrlrequest übergeben wird. Listing 4.19 zeigt die Funktion.

Die Frage, warum für Control Transfers von dem Konzept der Sockets abgesehen wurde, ist einfach zu beantworten: Zum einen handelt es sich ausschließlich um Transfers eines bestimmten Formats, zum anderen sind Transfers über Control-Pipes aber auch immer als ein für sich abgeschlossener Vorgang zu sehen. Anders als bei den sonstigen Transferarten gibt es keine fortlaufenden Sequenznummern oder sonstige (übertragungstechnische) Beziehungen der verschiedenen Transfers zueinander. Jeder Transfer startet von einer sich im definierten Ausgangszustand befindenden Pipe, was usb_control_transfer() zusätzlich noch durch De- und Reaktivieren der Pipe der Nummer Null sicherstellt. Nach der nötigen Konfiguration der Geräteadresse und der Pipe wird die eigentliche Arbeit an ctrl_transfer() abgegeben.

[11] Diese ist intern als NULL definiert und nur der Klarheit halber eingeführt.

Listing 4.19: Die Funktion usb_control_transfer()

```
1  int usb_control_transfer(usb_device *dev, usb_endpoint_descriptor *ep_desc,
   void *data, usb_ctrlrequest *ctrl_rqst){
2
3    int return_value;
4    unsigned int max_packet_size;
5    unsigned char ep_adr, tmp_status = SREG;
6    cli();
7
8    //select the correct values for endpoint-addressing
9    if (ep_desc == CTRL_EP){
10     ep_adr = 0;
11     max_packet_size = device_get_device_descriptor(dev)->bMaxPacketSize0;
12   } else {
13     ep_adr = (ep_desc->bEndpointAddress & USB_ENDPOINT_NUMBER_MASK);
14     max_packet_size = ep_desc->wMaxPacketSize;
15   }
16
17   // select hw-address
18   UHADDR = device_get_address(dev);
19
20   select_pipe(PIPE_64_0);
21   UPCONX &= ~(1<<PEN);
22   UPCONX |= (1<<PEN);
23   setup_pipe(PIPE_64_0, ep_adr, PTYPE_CONTROL);
24
25   return_value = ctrl_transfer(data, ctrl_rqst, max_packet_size);
26
27   freeze_pipe();
28   SREG = tmp_status;
29   return return_value;
30 }
```

4.5.9 Standard Device Requests

Nach Fertigstellung der Funktion usb_control_transfer() konnten – auf diese aufbauend – Funktionen zur Durchführung der Standard Device Requests implementiert werden. Die so entstandenen Funktionen befinden sich in der Datei `standard_device_requests.c`.

Um den Code kürzer zu halten, wurde das Aufbauen des Control-Request-Blocks und das Aufrufen von usb_control_transfer() in einer Hilfsfunktion gekapselt. Beispielhaft für die Requests sind in Listing 4.20 diese Hilfsfunktion usb_Ctrl_Rqst() und der Request Set_Address() im Quellcode gezeigt.

Die Werte innerhalb des Request-Blocks wurden für jeden Request konform zu Abschnitt 2.4.5.2 gewählt.

Zusätzlich zu den "Basisfunktionen" wie Set_Address() enthält die Datei `standard_device_requests.c` noch eine weitere Funktion, die das Abho-

Listing 4.20: Beispiele für Basisfunktionen aus `standard_device_requests.c`

```
1  int usb_Ctrl_Rqst(usb_device *dev, unsigned char bRequestType, unsigned char
   bRequest, unsigned int wValue, unsigned int wIndex, unsigned int size, void*
   data ){
2    usb_ctrlrequest ctrl_rqst;
3
4    // Build up the Request as defined by the specification
5    ctrl_rqst.bRequestType = bRequestType;
6    ctrl_rqst.bRequest     = bRequest;
7    ctrl_rqst.wValue       = wValue;
8    ctrl_rqst.wIndex       = wIndex;
9    ctrl_rqst.wLength      = size;
10   return usb_control_transfer(dev, NULL, data, &ctrl_rqst);
11 }
12
13 int Set_Address(usb_device *dev, unsigned char address){
14   return usb_Ctrl_Rqst(dev, USB_DIR_OUT | USB_TYPE_STANDARD |
     USB_RECIP_DEVICE, USB_REQ_SET_ADDRESS, address, 0, 0, NULL );
15 }
```

len und Speichern des Configuration Descriptors eines gegebenen Gerätes automatisiert. Bei Abruf eines Configuration Descriptors ist zu beachten, dass die Gesamtzahl der zu empfangenen Bytes zunächst unbekannt ist. Erst der Configuration Descriptor selbst gibt Auskunft über die Größe der an den Configuration Descriptor angehangenen Descriptoren. Bei Abruf des Configuration Descriptors muss also potentiell zusätzlicher Speicher alloziert werden. Listing 4.21 zeigt die hierzu erstellte Funktion.

Zunächst werden die ersten neun Byte des Configuration Descriptors abgerufen. Mit dem neunten Byte ist die Gesamtlänge aller zu übertragenden Descriptoren bekannt. Somit kann der benötigte Speicher realloziert und dann mit allen beim Abruf der Konfiguration gelieferten Descriptoren beschrieben werden.

4.5.10 Servicefunktionen

Zum einfacheren Umgang mit Geräten stellt *libAT90USB* zusätzlich zu den reinen Kommunikationsfunktionen Serviceroutinen zum automatisierten Gerätesetup und zum vollständigen Bereinigen von entfernten Geräten zur Verfügung.

Listing 4.21: Die Funktion usb_control_transfer()

```
1  int usb_get_device_configuration_descriptor(usb_device *dev, char
   config_index){
2
3    int return_value = 0;
4    usb_device_descriptor* dev_desc = device_get_device_descriptor(dev);
5    usb_config_descriptor* conf_desc = device_get_config_descriptor(dev);
6
7    if (config_index >= dev_desc->bNumConfigurations) return ERR_INDEX; //
     ERR_INDEX 0 configuration index out of range.
8
9    //Read the first 9 byte to get the number of bytes for the descriptor of
     this configuration
10   return_value = Get_Descriptor(dev, USB_DT_CONFIG, config_index, 0, 9,
     conf_desc);
11   if (return_value<0) return return_value;
12
13   //with the knowledge of the descriptorsize, we can allocate sufficient
     memory and request the whole descriptor now
14   device_set_config_descriptor(dev, (usb_config_descriptor*)realloc(conf_desc
     , conf_desc->wTotalLength ));
15   conf_desc = device_get_config_descriptor(dev);
16   return_value = Get_Descriptor(dev, USB_DT_CONFIG, config_index, 0,
     conf_desc->wTotalLength, conf_desc);
17
18   if (return_value<0) return return_value;
19
20   return return_value;
21 }
```

4.5.10.1 Gerätesetup

Die Bus Enumeration für ein Gerät kann eine Applikation komplett durch Aufruf der Funktion setup_device() übernehmen lassen. Diese lädt den Device Descriptor vom fraglichen Gerät herunter, weist ihm eine noch freie Adresse zu und aktiviert die erste Konfiguration des Geräts.

Listing 4.22 zeigt die Funktion. Als Parameter erwartet sie lediglich eine Referenz auf die Gerätestruktur, welche das zu konfigurierende Gerät repräsentiert. Diese sollte zuvor mittels device_create() erzeugt worden sein. Der Ablauf von setup_device() sollte anhand des Codes nachzuvollziehen sein. Die noch unbekannte Funktion reserve_address() reserviert eine noch freie Adresse aus dem zur Verfügung stehenden Adresspool. Den Erfolgsfall meldet setup_device() durch Rückgabe von Null an ihren Aufrufer. Ansonsten wird ein, wie gewohnt negativer, Fehlercode zurückgegeben.

Listing 4.22: Die Funktion setup_device()

```
1  int setup_device(usb_device *dev){
2
3    int return_value = 0;
4    int address = 0;
5
6    //first we need to read the device-descriptor
7    //the 8th byte determines the real max_packet_size of the default-ctrl-ep
8    return_value = Get_Descriptor(dev, USB_DT_DEVICE, 0, 0, 8,
     device_get_device_descriptor(dev));
9    if (return_value<0) return return_value;
10
11   //get the whole thing
12   return_value = Get_Descriptor(dev, USB_DT_DEVICE, 0, 0, 18,
     device_get_device_descriptor(dev));
13   if (return_value<0) return return_value;
14
15   //next, we need to assign an address to the device
16   address = reserve_address();
17   if (address<0) return address;
18
19   return_value = Set_Address(dev, address);
20   if (return_value<0) return return_value;
21   device_set_address(dev, address);
22
23   _delay_ms(10);
24
25   //address set, now get some more descriptors
26   return_value = usb_get_device_configuration_descriptor(dev, 0);
27   if (return_value<0) return return_value;
28
29   _delay_ms(10);
30
31   //finally, we can set the device config to the first available one
32   return_value = Set_Configuration(
33                    dev,
34                    device_get_config_descriptor(dev)->bConfigurationValue
35                  );
36
37   return return_value<0 ? return_value : 0;
38 }
```

4.5.10.2 Aufräumen nach Geräteentfernung

Wird ein Gerät vom Bus abgezogen, so sind alle mit diesem Gerät in Verbindung stehenden Sockets zu entfernen. Auch der durch die Gerätestruktur und seine Descriptoren belegte Speicher ist freizugeben. Hierfür kann die Applikation sich der Funktion cleanup_disconnected_device() bedienen, welche, wie schon setup_device(), als Parameter eine Referenz auf das zu bereinigende Gerät erwartet.

Wie in Listing 4.23 zu sehen, durchsucht die Funktion alle vorhandenen Sockets nach einem Bezug zu dem vom Bus entfernten Gerät. Sollte ein solcher vorhanden sein, so wird der Socket durch Aufruf von socket_destroy() entfernt. Anschließend wird die nun wieder freie Adresse des Geräts in den Adresspool zurückgegeben. Dies erledigt die Funktion release_address(), die somit den Gegenpart zu reserve_address() übernimmt. Letzlich kann auch das Gerät selbst über einen Aufruf von device_destroy() aus dem Speicher entfernt werden.

Listing 4.23: Die Funktion cleanup_disconnected_device()

```
1  void cleanup_disconnected_device(usb_device *device){
2    unsigned char tmp_status = SREG;
3    cli();
4
5    //go through all sockets and remove those associated with the device
6    for (int i=0; i<USB_MAX_SOCKETS; i++){
7      if (sockets[i] != NULL) {
8        if (sockets[i]->device == device) socket_destroy(i);
9      }
10   }
11
12   //mark the devices address as free...
13   release_address(device_get_address(device));
14   //get rid of the device
15   device_destroy(device);
16
17   SREG = tmp_status;
18 }
```

4.5.11 Simulation der Interrupt-Pipes

In Abschnitt 4.5.7 wurde die Beschreibung der Simulation von Interrupt-Pipes ausgelassen. Diese kann mit dem zwischenzeitlich vermittelten Wissen über die Kommunikationsfunktionen nun erfolgen.

Jeder Socket kann durch Aufruf von socket_interrupt_request() dazu veranlasst werden, in bestimmten Abständen neue Daten bei dem mit ihm assoziierten Endpunkt anzufragen[12]. Die Funktion bindet eine Struktur vom Typ virtual_interrupt_pipe an den ihr übergebenen Socket. Die entsprechende Typdefinition ist in Listing 4.24 wiedergegeben. Der Quellcode der Funktion socket_interrupt_request() findet sich in Listing 4.25.

[12] Auf ein automatisches Versenden von Daten über Interrupt-Pipes wurde wegen der geringen Anwendbarkeit im μC-Umfeld verzichtet.

Listing 4.24: Typdefinition zu virtual_interrupt_pipe

```
typedef struct {
    void *buffer;
    int buffer_size;
    int polling_msec;
    volatile int counter;
    volatile unsigned char new_data;
} virtual_interrupt_pipe;
```

Listing 4.25: Die Funktion socket_interrupt_request()

```
int socket_interrupt_request(int socket_number, void *buffer, int buffer_size
, int polling_msec){

    usb_socket *sck;
    unsigned char tmp_status = SREG;
    cli();

    //get the socket-pointer
    sck = get_socket_by_number(socket_number);
    if ((int)sck < 0){ SREG = tmp_status; return (int)sck; }

    //now check whether it already is an interrupt-socket, free if needed
    if ( sck->interrupt_pipe != NULL ) {
      free(sck->interrupt_pipe);
    }

    sck->interrupt_pipe                 = malloc(sizeof(
    virtual_interrupt_pipe));
    sck->interrupt_pipe->buffer         = buffer;
    sck->interrupt_pipe->buffer_size    = buffer_size;
    sck->interrupt_pipe->polling_msec   = polling_msec;
    sck->interrupt_pipe->counter        = 0;
    sck->interrupt_pipe->new_data       = 0;

    SREG = tmp_status;
    return 0;
}
```

Neben dem Socket benötigt die Funktion noch einige weitere Informationen. Zum einen ist über die Parameter *buffer und buffer_size ein Speicherbereich anzugeben, der die vom Endpunkt gesendeten Daten aufnimmt. Zum anderen ist auch ein einzuhaltender Abstand zwischen aufeinanderfolgenden Abfragen des Endpunkts mitzugeben. Dies geschieht über den Parameter polling_msec, der das Abfrageintervall in Millisekunden enthält. Die tatsäch-

lich erreichte zeitliche Auflösung hängt hierbei jedoch von TIMER_FREQ ab und beträgt standardmäßig nur 10ms. Sollte eine genauere Abfragefrequenz benötigt werden, so ist TIMER_FREQ entsprechend anzupassen.

Ist ein Socket einmal als Interruptsocket eingerichtet, so übernimmt die im Hintergrund immer wieder durch die ISR des Timer0 angestoßene Funktion do_interrupt_sockets() die weitere Arbeit. Listing 4.26 zeigt den dabei relevanten Sourcecode.

Listing 4.26: Die Funktion do_interrupt_sockets()

```
1  void do_interrupt_sockets(void){
2
3    for(int i=0; i<USB_MAX_SOCKETS; i++){
4
5      usb_socket *sck = sockets[i];
6
7      if (sck!=NULL){
8        virtual_interrupt_pipe *pipe = sck->interrupt_pipe;
9        if (pipe != NULL){
10         pipe->counter += 1000/TIMER_FREQ;
11
12         //check whether the pipe needs to look for new data
13         if (pipe->counter >= pipe->polling_msec){
14           if( !(pipe->new_data) ){
15             //receive data and set new_data, if new data is available
16             int status = usb_recv(i, pipe->buffer, pipe->buffer_size);
17             if ( status > 0 ) pipe->new_data=status;
18           }
19           //reset the counter for the pipe
20           pipe->counter = 0;
21         }
22       }
23     }
24   }
25 }
```

Die Funktion durchläuft alle Sockets und untersucht, ob es sich bei diesen um Interruptsockets handelt. Dies ist über die Komponente interrupt_pipe festzustellen. Handelt es sich bei ihr nicht um einen Nullpointer, so muss der Socket weiter behandelt werden.

In jeder Struktur des Typs virtual_interrupt_pipe befindet sich ein Zähler, in welchem die seit dem letzen Übertragungsversuch über den Socket vergangene Zeit gespeichert wird. Da der Zähler die vergangene Zeit in Millisekunden angibt und do_interrupt_sockets() mit einer Frequenz von TIMER_FREQ angestoßen wird, ist bei jedem Aufruf der Zähler um $\frac{1000}{TIMER\_FREQ}$ zu erhöhen. Übertrifft der Zähler die in polling_msec eingetragene Zeitspanne, so ist

es Zeit für eine neue Anfrage. Die Anzahl der dabei zurückgegebenen Bytes wird in der Komponente new_data gespeichert. Bei Lowspeed-Geräten liegt die maximal mögliche Bytezahl pro Abfrage über Interruptsockets auf diese Weise bei acht, bei Fullspeed-Geräten bei 64 Byte. Daher reicht hier der Typ **char** vollkommen aus.

Der Applikation ist durch die Funktion socket_newdata() nun die Möglichkeit gegeben, zu untersuchen, ob – und falls ja, wie viele – neue Daten im Puffer eines Interruptsockets vorliegen. Dazu fragt die Funktion im wesentlichen die new_data-Komponente ab und gibt deren Wert an die Applikation zurück. Die Applikation kann im Puffer enthaltene Daten dann verarbeiten.

Solange der Puffer mit (unverarbeiteten) Daten beschrieben ist, finden keinesfalls neuen IN-Requests für den abzufragenden Endpunkt statt. Die Applikation kann also ohne weiteres auf dem Puffer operieren.

Nach der Datenverarbeitung markiert die Applikation den Puffer mit Hilfe der Funktion socket_next_interrupt_request() als frei. Im nächsten Abfrageintervall des Sockets wird daraufhin ein neuer IN-Request durchgeführt.

Diese einfache Implementation der Abfrage von Interrupt-Pipes wurde mit Hinblick auf Eingabegeräte und ähnliches entwickelt, welche pro Interrupt Transfer immer nur eine Transaktion benötigen. Sollte eine Applikation eine komplexere Behandlung von Interrupt Transfers benötigen, so muss der Programmierer diese selbst implementieren.

Die hier vorgestellte Methode nimmt dem Programmierer aber vor allem auch die Arbeit ab, dafür zu sorgen, dass Interrupt Transaktionen nicht mit einer zu hohen Frequenz gestartet werden. Ein zu niedriges Abfrageintervall belegt unnötigt kostbare Buszeit, welche oft von anderen Transfers benötigt wird.

Der Quelltext zu socket_newdata() und socket_next_interrupt_request() kann der beiligenden CD entnommen werden.

4.5.12 Hinweise zur Nutzung der Grundfunktionalität

An dieser Stelle soll noch einmal das Vorgehen beschrieben werden, welches zur Nutzung der Basisfunktionalität notwendig ist.

1. Im Quellcode der Applikation wird der Header `usb_host.h` eingebunden.

2. usb_host_init() wird zur Initialisierung der USB-Einheit des AT90USB-1287 aufgerufen.

3. Die Applikation prüft mittels root_dev_connected(), ob ein Gerät an den AT90USB1287 angeschlossen wurde.

4. Ist ein Gerät angeschlossen, so ruft die Applikation device_create() auf, um eine Gerätestruktur zu erstellen.

5. Über die Funktion setup_device() wird die Bus Enumeration durchlaufen.

6. Nun sind beliebige weitere Requests über die Default Control Pipe möglich. Hierzu stehen beispielsweise in `standard_device_requests.h` diverse Funktionen bereit.

7. Die Applikation überprüft die Descriptoren. Diese erhält sie über Funktionen wie device_get_endpoint_descriptor() in `usb_device_struct.h`.

8. Auf Basis der gesammelten Informationen können weitere Endpunkte für Control Transfers gefunden werden. Diese werden (ohne Socket) mit usb_ctrl_transfer() durchgeführt.

9. Zu anderen Endpunkten wird mittels socket_create() ein neuer Socket geöffnet.

10. Über die Funktionen usb_send() oder usb_recv() wird über die erstellten sockets kommuniziert.

11. Sockets zu Interrupt Endpunkten können zur automatischen, periodischen Abfrage genutzt werden. Hierzu sind socket_interrupt_request(), socket_newdata() sowie socket_next_interrupt_request() zu verwenden.

12. Sockets können explizit mit socket_destroy() geschlossen werden. Dies wird normalerweise nicht benötigt.

13. Die Applikation prüft während der Betriebsdauer des Gerätes mit Hilfe von root_dev_disconneted(), ob das Gerät vom Bus abgezogen wurde. Ist dies der Fall, so schließt sie alle Operationen ab, für welche das Gerät notwendig ist.

14. Die Applikation lässt die Bibliothek Speicher und Sockets über die Funktion cleanup_disconnected_device() aufräumen.

Die Funktionen, welche der Programmierer auf diese Weise direkt nutzt, rufen wie beschrieben ihrerseits Funktionen wie bus_read_data() oder auch bus_write_data() auf, um die Lowlevel-Kommunikation über USB zu ermöglichen. Auch die Funktionen zur Adressvergabe und ähnlichem arbeiten im normalen Betrieb unbemerkt vom Programmierer im Hintergrund. Es steht dem Programmierer jedoch frei, solche Funktionen direkt für seine Zwecke einzusetzen, auch wenn dies üblicherweise nicht notwendig ist.

4.6 Gerätetreiber

Die reine Möglichkeit zur Übertragung von Daten über USB bringt noch keinen unmittelbaren Nutzen. Um sinnvolles mit diesen Übertragungen bewerkstelligen zu können, müssen zusätzlich noch Gerätetreiber entwickelt werden. Gleichzeitig dienen die Treiber auch als Test der darunter liegenden Funktionen von *libAT90USB*.

Als Beispiele für Gerätetreiber dienen an dieser Stelle ein Hubtreiber, ein Treiber für die Basisfunktionalität von HID-Devices sowie ein weiterer für Geräte der Mass-Storage-Klasse. Der Hubtreiber ist eine absolute Notwendigkeit für den Mehrgerätebetrieb und musste daher zwangsläufig im Rahmen dieser Arbeit erstellt werden. Jedoch bietet er als Treiber den Vorteil, ein für sich abgeschlossenes Konstrukt zu sein.

Bei HID- und Mass-Storage-Treibern stellt sich die Lage hingegen ungünstiger dar: So müssen HID-Geräte nicht nur als solche identifiziert werden, für einen tatsächlichen praktischen Einsatz sind die durch sie gelieferten Daten auch noch zu interpretieren. Davon wird im Folgenden abgesehen. Das Protokoll einer Maus beispielsweise ist einfach genug, um die korrekte Funktion der Übertragung auch auf Basis der Rohdaten erkennen zu können.

Komplizierter jedoch ist der Mass-Storage-Treiber. Nicht nur muss das Kommunikationsprotokoll für Geräte dieser Klasse eingehalten werden, zusätzlich sind die Daten auf jeden Fall zu interpretieren, um einen sinnvollen Nutzen aus dem Gerät ziehen zu können. Insbesondere ist hierfür ein zusätzlicher Dateisystemtreiber vonnöten.

Allgemein sei zu den hier vorgestellten Treibern gesagt, dass es sich eher um Beispieltreiber als um vollständige Implementationen im Sinne der Geräteklassenspezifikation handelt. Auch ist eine detailliertere Vorstellung der einzelnen Spezifikationen an dieser Stelle nicht möglich. Im Falle des Hubs wurden die nötigen Informationen bereits ab Abschnitt 2.4.7 vermittelt. Im Falle von HID und Mass-Storage umfasst die Summe der verwendeten Spezifikationen selbst wieder mehrere hundert Seiten. Daher wird hier jeweils nur die Grundfunktion und das entsprechende Vorgehen der Treiber erläutert, um ein Verständnis für den erstellten Quellcode zu schaffen.

4.6.1 Konventionen für Treiber

Ein Treiber sollte im Allgemeinen eine Möglichkeit bieten, um herauszufinden, ob mit seiner Hilfe ein bestimmtes Gerät beziehungsweise eines dessen Interfaces betrieben werden kann. Jeder der hier vorgestellten Treiber besitzt daher eine Funktion, welche diese Aufgabe übernimmt.

Um dem Programmierer das Erkennen dieser Funktion zu erleichtern,

wurde sie in jedem Treiber mit dem Suffix _probe versehen. Abhängig von der zu betreibenden Geräteklasse wird der Funktion ein Gerät oder auch ein Interface als Parameter übergeben, welches auf Betriebsmöglichkeit zu testen ist.

Liefert eine solche Funktion Erfolg zurück, so kann der Aufrufer davon ausgehen, dass das Gerät mit dem fraglichen Treiber bedienbar ist. Bei den hier gezeigten Treibern werden durch die _probe-Funktionen auch direkt alle weiteren (klassen- oder gerätespezifischen) Aufgaben erledigt, welche für den folgenden Betrieb des Geräts notwendig sind. Das Gerät ist bei Erfolg der _probe-Funktion also bereit, seine Arbeit zu beginnen.

Sollte das Gerät hingegen nicht mit Hilfe des jeweiligen Treibers bedienbar sein, so teilen die _probe-Funktionen dies ihrem Aufrufer durch Rückgabe des Codes ERR_NODEV mit. Der Aufrufer kann dann nach eigenem Ermessen weiter mit dem Gerät verfahren.

4.6.2 Hubtreiber

Bei dem für *libAT90USB* implementierten Hubtreiber handelt es sich um eine Portierung des Treibers, welcher im Linux-Kernel Verwendung findet. Bei der Anpassung zahlte sich das Vorgehen aus, Konstanten und Strukturen möglichst nahe an die dort verwendeten anzulehnen.

Der Treiber übernimmt alle Aufgaben, die zum Betrieb eines Hubs notwendig sind. Hierzu gehören unter anderem Initialisierung, Konfiguration, aber auch das regelmäßige Abfragen der Portzustände, um Anschluss- und Absteckereignisse zu erkennen.

4.6.2.1 Änderungen für die Applikation

Für die Applikation muss sich das Vorgehen bei Verwendung des Hubtreibers insofern ändern, als dass durch Verwenden eines Hubs nun nicht mehr immer nur ein Gerät zur Zeit zum Bus hinzustoßen kann, sondern sofort mehrere. An jedem Port eines eingesteckten Hubs könnte ja ein Gerät – oder wahlweise auch noch ein Hub, für den dann wieder das gleiche gilt – vorhanden sein. Eben solches gilt auch beim Abziehen von Geräten. Wird ein Hub entfernt, so muss die Applikation geeignet über das Verschwinden der an diesem angeschlossenen Geräte informiert werden. Hubs selbst hingegen sind der Applikation nicht als Geräte zu melden. Der Hubtreiber übernimmt vollständig die Kontrolle über Hubs und verbirgt damit deren Vorhandensein vor der Applikation.

Zur Übergabe von Informationen über Geräte an die Applikation wird im Rahmen des Hubtreibers eine FIFO-artige Struktur genutzt. In diese schreibt

der Hubtreiber jeweils ein Tupel aus Gerät und dessen neuem Zustand. Die Applikation kann diese Tupel dann abrufen und wird so über Veränderungen am Bus informiert[13].

Die Größe der zur Übergabe genutzten Struktur lässt sich in `usb_hub.h` konfigurieren. Ebenfalls einstellbar ist auch die maximale Anzahl der Ports, die bei einem Hub verwaltet werden können, sowie die maximale Anzahl von gleichzeitig verwaltbaren Hubs. Listing 4.27 zeigt die entsprechenden Definitionen.

Listing 4.27: Wesentliche Konfigurationsoptionen für den Hubtreiber

```
1  /** Maximum number of hubs which can be connected at a time */
2  #define USB_MAXHUBS              3
3
4  /** Maximum number of additional devices which can be handled per hub */
5  #define USB_MAXCHILDREN          4
6
7  /**
8   * Maximum number of events (@see usb_evt) that can be waiting for the
    application to be processed.
9   * This somewhat limits the maximum number of devices which can be connected
    at a time, since
10  * an event needs to be queued for each newly connected and disconnected
    device. This might lead
11  * to problems if a hub with a total number of children which is bigger than
12  * USB_EVENTQUEUE_SIZE becomes connected and almost instantaniously
    disconnected again.
13  **/
14 #define USB_EVENTQUEUE_SIZE      2*USB_MAX_DEVICES
```

Aus Applikationssicht sind bei möglicher Verwendung von Hubs nur zwei Funktionen relevant. Zunächst zu nennen ist hier hub_poll(), welche intern das Prüfen auf Anschluß oder Abziehen von Geräten anstößt. Als Rückgabewert liefert sie die Anzahl der daraufhin auf Verarbeitung durch die Applikation wartenden Ereignisse.

Die zweite Funktion ist queue_get_event(), über welche die Applikation die entsprechenden Ereignisse der Reihe nach entgegen nehmen kann. Bemerkenswert dabei ist, dass der Hubtreiber intern schon für ein Setup der gefundenen Geräte sorgt. Die Applikation kann also neu angeschlossene Geräte sofort nutzen.

[13] Tatsächlich handelt es sich nicht wirklich um einen FIFO. Die gewählte Implementation arbeitet aber äquivalent zu einem solchen, wenn die Applikation sich wie im Weiteren beschrieben verhält. Daher wurde auf den Overhead, welcher ein vollständig implementierter FIFO verursacht, verzichtet.

Listing 4.28 zeigt das Vorgehen aus Applikationssicht.

Listing 4.28: Verwendung von Hubs aus Applikationssicht

```
1  int main (void){
2      int state, event_cnt=0;
3      usb_device *dev;
4
5      usb_host_init();
6
7      while (1){
8          event_cnt = hub_poll();
9          if (event_cnt<0) continue;
10
11         //for each event...
12         for(int i=0; i<event_cnt ; i++){
13
14             //read the event
15             queue_get_event( &dev, &state);
16
17             if(state == CONNECTED) {
18                 //device *dev got connected
19             }
20             if(state == DISCONNECTED){
21                 //device *dev got disconnected
22             }
23         }
24     }
25 }
```

Wie in Zeile 15 zu sehen ist, modifiziert queue_get_event() die beiden Variablen dev und state, um die Applikation über ein aufgetretenes Ereignis zu informieren. Die Konstanten CONNECTED sowie DISCONNECTED sind in der Datei `usb_hub.h` vordefiniert.

4.6.2.2 Interna des Treibers

Was den internen Aufbau des Hubtreibers angeht, soll an dieser Stelle nur die Funktion hub_poll() genauer betrachtet werden. Sie bedient sich direkt der durch die Grundfunktionalität von *libAT90USB* gebotenen Geräteerkennung und zeigt, wie der Treiber Hubs vor der Applikation verbirgt. Listing 4.29 zeigt das Gerüst des Quellcodes der Funktion.

Die Funktion hub_poll() kann durch Aufruf von root_dev_connected() feststellen, dass ein Gerät an den Root-Port des AT90USB1287 angeschlossen wurde (Zeile 9). Ist dies der Fall, so wird sofort eine neue Geräterepräsentation erzeugt und das Gerät mit Hilfe von setup_device() in einen betriebsbereiten Zustand gebracht.

Listing 4.29: Die Funktion hub_poll()

```
int hub_poll(void){

    unsigned char tmp_status = SREG;
    cli();

    static usb_device *root_device = NULL;

      //transparently handle connects for hubs and usb-functions
      if(root_dev_connected()){

          int return_value;

          //create and setup the device
          root_device = device_create();
          return_value = setup_device(root_device);
          if(return_value < 0){ SREG = tmp_status; return return_value; }

          //test whether it's a hub and configure it if so
          return_value = hub_probe(root_device);

          if( return_value == ERR_NODEV ){ //it's a usb-function of some kind
              queue_event(root_device, CONNECTED);

          }else if(return_value < 0){ //we encountered some problems.
              SREG = tmp_status; return return_value;
          }

          SREG = tmp_status;
          return queue_status();
      }

      //transparently handle disconnects for hubs and usb-functions
      if(root_dev_disconnected()){

        usb_disconnect(root_device);
        root_device = NULL;

        SREG = tmp_status;
        return queue_status();
      }

    //Cycle through all hubs and collect their events
    for(int i=0; i<USB_MAXHUBS; i++){
        hub_events(hub_devs[i]);
    }

    SREG = tmp_status;
    return queue_status();
}
```

Als nächstes lässt hub_poll() durch die Funktion hub_probe() untersuchen, ob das Gerät direkt mit dem Hubtreiber bedienbar ist. In diesem Fall handelt es sich um einen Hub, dessen Existenz der Applikation nicht weiter bekannt gemacht werden muss. hub_probe() stößt dabei sogleich die Inbetriebnahme des Hubs an, so dass dieser seine Arbeit aufnimmt. Kann der Hubtreiber das angeschlossene Gerät nicht bedienen (das Gerät ist also kein Hub), so stellt er das Anschlussereignis in die Warteschlange ein (Zeile 22). Dort wartet es auf Bearbeitung durch die Applikation.

Beim Abziehen des Gerätes vom Root-Port räumt der Treiber mit Hilfe von usb_disconnect() auf. Diese Funktion untersucht, ob es sich bei dem entfernten Gerät um einen Hub handelte. In diesem Fall entfernt sie nicht nur das Gerät selbst mittels cleanup_disconnected_device(), sondern geht auch noch sukzessive alle dessen Kinder durch, um so den ganzen vom Bus entfernten Baum aufzuräumen. Dabei wird für jedes Gerät ein Ereignis in die Warteschlange eingereiht, wenn es sich bei dem Gerät nicht um einen Hub handelte.

Sollte bezüglich des Root-Ports kein Ereignis aufgetreten sein, so ruft der Treiber für jeden angeschlossenen Hub die Funktion hub_events() auf. Diese aktualisiert den Status der Ports aller Hubs durch Auswertung der Daten, welche diese über ihren Interrupt Endpunkt liefern. Dabei werden neu angeschlossene oder entfernte Geräte erkannt und analog zum Vorgehen von hub_poll() behandelt. Es können also problemlos ganze Kaskaden von Hubs und Endgeräten angesteckt beziehungsweise abgezogen werden.

Abschließend liefert hub_poll() immer die Anzahl der auf Verarbeitung wartenden Ereignisse an die Applikation zurück. Diese muss sich nun entsprechend der geänderten Situation verhalten.

Der komplette Hubtreiber ist auf der beiligenden CD zu finden.

4.6.2.3 Zusammenfassung zur Nutzung des Hubtreibers

Zusammenfassend fallen bei Nutzung des Hubtreibers folgende Arbeiten für den Programmierer an.

1. Im Quellcode der Applikation wird der Header `usb_hub.h` eingebunden.
2. usb_host_init() wird zur Initialisierung der USB-Einheit des AT90USB-1287 aufgerufen.
3. hub_poll() wird regelmäßig aufgerufen, um Geräteanschluss und -trennung durch den Hubtreiber feststellen zu lassen.
4. hub_poll() gibt dabei die Anzahl der auf Verarbeitung wartenden Ereignisse an die Applikation zurück.

5. Über queue_get_event() werden diese Ereignisse aus der Warteschlange entnommen.

6. Handelt es sich bei einem Ereignis um den Anschluss eines Gerätes an den Bus, so hat das Gerät bereits die Bus Enumeration durchlaufen und ist betriebsbereit. Die Applikation kann dann wie in den Punkten 6 bis 12 in Abschnitt 4.5.12 verfahren, um mit dem Gerät zu kommunizieren.

7. Handelt es sich bei einem Ereignis um das Abziehen eines Gerätes vom Bus, so darf die Applikation nicht weiter auf dieses zugreifen. Alle mit dem Gerät in Verbindung stehenden Strukturen sind zu entfernen. Die Gerätestruktur selbst sowie die Descriptoren und Sockets wurden bereits durch den Hubtreiber entfernt. Dies ist bei korrektem Verhalten der Applikation problemlos möglich, da Speicherallokation und -deallokation über hub_poll() nur synchron zum Programmablauf auftreten können.

4.6.3 Treiber für HID-Geräte

4.6.3.1 Die HID Geräteklasse

Geräte der HID-Klasse dienen zur Interaktion des Menschen mit der Maschine. Dieser Zweck wurde schon im Namen der Geräteklasse festgehalten: die Abkürzung HID steht für *Human Interface Device*.

Als Beispiele solcher Geräte nennt die HID-Spezifikation [Speb] Mäuse, Tastaturen und Datenhandschuhe aber auch weniger offensichtliche Gerätschaften wie Voltmeter oder Barcodeleser. Auch Joysticks und Joypads gehören in der Regel dieser Klasse an, jedoch sind Force-Feedback-Geräte wegen ihrer veränderten Echtzeitanforderungen explizit aus dem Geltungsbereich der HID-Spezifikation ausgenommen. Für sie existiert eine gesonderte Geräteklasse namens PID (Physical Interface Device).

4.6.3.2 Treibersoftware

Der hier gezeigte Treiber für HID-Geräte dient nicht unbedingt dazu, ein spezielles Gerät seinem Bestimmungszweck entsprechend zum Einsatz zu bringen. Vielmehr handelt es sich um eine Sammlung von Funktionen, welche generell beim Betrieb von HID-Geräten benötigt werden.

Dies liegt zum einen an der mangelnden direkten Anwendbarkeit – eine Maus oder ein Joypad wird, angeschlossen an einen μC, tendenziell eher zweckentfremdet genutzt werden. Zum anderen ist aber auch das Protokoll solcher Geräte eher einfach, so dass die empfangenen Daten durchaus direkt

an die Applikation weitergereicht werden können, die diese dann für ihre Zwecke nutzt.

Der Treiber enthält Funktionen für die Durchführung der HID-klassenspezifischen Requests *Get-* und *Set_Report*, *Get-* und *Set_Idle* sowie *Get-* und *Set_Protocol*, wie sie in der HID-Spezifikation [Speb] zu finden sind.

Ferner lässt sich mit Hilfe der Funktion get_hid_descriptor() der HID-Descriptor eines Interfaces finden. Diese Funktion arbeitet analog zu den beiden schon bekannten Funktionen get_endpoint_descriptor() beziehungsweise get_interface_descriptor().

Natürlich stellt der HID-Treiber auch eine Funktion zur Verfügung, die prüft, ob es sich bei einem bestimmten Interface eines Gerätes um ein HID-Interface handelt. HID-Geräten steht es frei, mehrere voneinander getrennt nutzbare Interfaces anzubieten. Bei Tastaturen können zum Beispiel mittels der Endpunkte des ersten Interfaces Informationen über Standardtasten übertragen werden. Ein zweites Interface wird dann für Multimedia- oder sonstige, bei modernen Keyboards immer häufiger anzutreffende, Sondertasten genutzt. Die Funktion, welche ein Interface auf Verwendbarkeit mit dem HID-Treiber prüft, heißt hid_probe(). Auch hier wird also das Suffix _probe() für die Funktion zum Betriebstest genutzt. Die Funktion erhält die Gerätestruktur und die Nummer des zu untersuchenden Interfaces als Parameter.

Nachdem ein Interface mittels hid_probe() als HID-Interface erkannt wurde, ist es der Applikation möglich, auf die durch das Interface zur Verfügung gestellten Daten zuzugreifen. Dies geschieht mit Hilfe der Funktion hid_get_data(). Der Treiber sorgt dabei im Hintergrund mittels eines (durch hid_probe() angelegten) Interruptsockets für eine ständige Aktualisierung der Daten. hid_get_data() kopiert die auf dem Socket gelieferten Daten in einen durch die Applikation angegebenen Puffer. Als Parameter benötigt sie daher diesen Zielpuffer und dessen Fassungsvermögen. Außerdem sind noch Gerät und Interface, von welchem Daten zu lesen sind, anzugeben. hid_get_data() liefert abschließend die Anzahl der im Puffer neu zur Verfügung stehenden Bytes an den Aufrufer zurück.

Wird ein Gerät vom Bus abgezogen, so muss der HID-Treiber mit Hilfe der Funktion hid_disconnect() zum Aufräumen bewegt werden. Die Funktion erhält dabei einen Zeiger als Parameter, welcher vor dem Abziehen des Gerätes als Referenz auf die dieses Gerät repräsentierende Struktur diente.

Der Treiber kann maximal eine zur Compilezeit festgelegte Anzahl von HID-Geräten verwalten. Auch die maximal pro Geräte bedienbare Zahl an HID-Interfaces wurde aus Gründen der Einfachheit eingeschränkt. Beide Konfigurationsoptionen befinden sich in der Datei `usb_hid.h`. Die Standardwerte sind Listing 4.30 zu entnehmen.

Listing 4.30: Standardkonfiguration des HID-Treibers

```
1  /**
2   * Maximum Number of HIDs which can be handled by the driver.
3   **/
4  #define MAX_HID_DEVS          1
5
6  /**
7   * Maximum Number hid-interfaces which can be handled per device.
8   **/
9  #define MAX_HID_INTERFACES    3
```

Um die Anwendung des HID-Treibers zu demonstrieren, zeigt Listing 4.31 ein kleines Beispielprogramm, welches auch den vorangehend gezeigten Hubtreiber einsetzt. Dies dient leider tatsächlich eher der Demonstration und hat nur begrenzten Nutzen. Bei den meisten HID-Geräten handelt es sich um Lowspeed-Geräte, die aus dem in Abschnitt 4.2.1.2 genannten Grund jeweils nur am Root-Port des AT90USB1287 zu betreiben sind. Die Beispielapplikation verwaltet daher auch nur ein HID-Gerät.

Die Ausgabe des Programms bei Anschluss einer USB-Tastatur ist in Abbildung 4.3 zu sehen. Die Tastatur kommuniziert über zwei Interfaces mit dem Host. Auf Inferface0 sind die Standardtasten zu sehen. Interface1 wird bei der verwendeten Tastatur nur für einige wenige Sondertasten genutzt, weshalb dieses Interface nur jeweils zwei Byte überträgt. Sowohl das Drücken als auch das Loslassen einer Taste löst jeweils ein Ereignis aus, welches die Tastatur an den Host weitergibt. Beispielsweise wurde nach dem Anschluss der Tastatur die Taste t gedrückt und dann sofort wieder losgelassen, was zu den Ausgaben in Zeile 5 und 6 führte.

Der komplette Code für den HID-Treiber findet sich auf der beiliegenden CD.

4.6.4 Treiber für Mass-Storage-Geräte

4.6.4.1 Geräteklasse und Funktionsbeschreibung

In die Klasse “Mass-Storage” fallen – wie der Name schon verrät – solche Geräte, die in irgendeiner Form als Massenspeicher fungieren können. Hierzu gehören reine Speichergeräte, wie die bekannten USB-Speichersticks, externe Festplatten oder sogenannte Cardreader. Aber auch Digitalkameras, Mobiltelefone oder MP3-Player nutzen mitunter das Mass-Storage-Interface, um Daten mit einem PC auszutauschen. Die Spezifikation zur Mass-Storage-

Listing 4.31: Beispiel für die Verwendung des HID-Treibers

```

int main (void){
  int state, event_cnt=0;
  usb_device *dev, *hid = NULL;

  USART_Init();
  printf_P(PSTR("USART configured properly.\r\n"));
  usb_host_init();

  while (1){
    event_cnt = hub_poll();
    if (event_cnt<0) continue;

    for(int i=0 ; i<event_cnt ; i++){

      queue_get_event( &dev, &state);

      if(state == CONNECTED) {

        //go through all the interfaces
        for(int j=0; j<device_get_config_descriptor(dev)->bNumInterfaces; j
        ++){
          int probe = hid_probe(dev, j);
          if ( probe>=0 ){
              printf_P(PSTR("HID found with iface %d.\r\n"), j);
              hid = dev; //hid-dev, let's remember it
          }
        }
      }

      if(state == DISCONNECTED){
        if (hid==dev){
          printf_P(PSTR("HID got disconnected.\r\n"));
          hid_disconnect(dev);
          hid = NULL;
        }
      }
    }//end loop through events

    //do stuff for hid-devices
    if(hid){
      char buffer[8];

      //simply loop through all the interfaces.
      //if it's not registered as a hid, the driver will return an ErrorCode
      for (int j=0; j<min(MAX_HID_INTERFACES, device_get_config_descriptor(
      hid)->bNumInterfaces); j++){

        //read some data into to buffer
        int buffsize = hid_get_data(hid, j, buffer, 8);
        if (!buffsize) continue;

        printf_P(PSTR("hid-interface %d "), j);
        for(int k=0; k<buffsize; k++){
          printf_P(PSTR(" | %02x "),(unsigned int) buffer[k]);
        }
        printf(PSTR("\r\n"));
      }

    }//end hid
  }//end loop
  return 0;
}
```

```
1  ~ $ cat /dev/ttyS0
2  USART configured properly.
3  HID found with iface 0.
4  HID found with iface 1.
5  hid-interface 0  | 00  | 00  | 16  | 00  | 00  | 00  | 00  | 00
6  hid-interface 0  | 00  | 00  | 00  | 00  | 00  | 00  | 00  | 00
7  hid-interface 0  | 00  | 00  | 17  | 00  | 00  | 00  | 00  | 00
8  hid-interface 0  | 00  | 00  | 00  | 00  | 00  | 00  | 00  | 00
9  hid-interface 1  | 02  | 01
10 hid-interface 1  | 02  | 00
11 hid-interface 1  | 02  | 02
12 hid-interface 1  | 02  | 00
13 HID got disconnected.
```

Abbildung 4.3: Mögliche Ausgabe des Beispiels für den HID-Treiber

Klasse ist unter [Spee] zu finden.

Grundsätzlich lässt die Spezifikation verschiedene Subklassen für Mass-Storage-Geräte zu, nennenswert verbreitet sind jedoch nur solche, welche das "SCSI transparent command set" nutzen.

Für diese sind wiederum zwei verschiedene Protokolle zum Datenaustausch mit dem Host definiert. Hierbei handelt es sich um das *Bulk-Only-Transfer*- sowie das *Control, Bulk, Interrupt*-Protokoll (BOT und CBI).

Bulk-Only-Transfer Geräte, welche das BOT-Protokoll einsetzen, kommunizieren mit dem Host über zwei einfache Bulk-Transfer-Endpunkte. Einer der beiden Endpunkte dient dem Gerät dabei zur Annahme von Daten vom Host, der andere wird zum Versenden von Daten genutzt.

Der Datenaustausch mit Mass-Storage-Geräten erfolgt in drei Stufen. In der ersten Stufe erteilt der Host dem Gerät ein Kommando. Hierdurch wird das Gerät beispielsweise zu lesendem oder schreibendem Zugriff auf das Medium angewiesen. Aber auch Informationen wie die Speicherkapazität des Mediums kann der Host so vom Gerät anfordern.

Bei den Kommandos selbst handelt es sich um SCSI-Befehle, wie sie in der Spezifikation "*MultiMedia Command Set*" ([Speg]) definiert sind. Der Host verpackt das SCSI-Kommando in einen sogenannten *Command Block Wrapper* (CBW), der dann über den USB versendet wird.

Üblicherweise folgt dann eine zweite Stufe, in der Daten entsprechend des vorher gesendeten Kommandos "roh" zum Gerät beziehungsweise vom Gerät übertragen werden.

In der letzten Stufe erhält der Host vom Gerät eine Rückmeldung als

Statusinformation über den Transfer. Diese wurde zum Transport über den Bus in einen sogenannten *Command Status Wrapper* (CSW) eingepackt.

Die Nutzung der SCSI-Kommandos hat den Vorteil, dass bei Einführung der Geräteklasse bereits existierende Software zur Durchführung der Kommunikation genutzt werden konnte. Ein SCSI-Treiber und alle auf diesen aufbauende Schichten können problemlos zur Kommunikation mit dem Gerät eingesetzt werden, weshalb sich Mass-Storage-Geräte der Subklasse "SCSI transparent command set" als gutes Beispiel für den Erfolg des Schichtenprinzips darstellen.

Control, Bulk, Interrupt Geräte, welche das CBI-Protokoll nutzen, informieren den Host zusätzlich über den Status eines Transfers mittels eines Interrupt-Endpunktes. Ihre sonstigen Eigenschaften entsprechen denen der BOT-Geräte.

Das CBI-Protokoll hat jedoch kaum Verbreitung gefunden und darf laut Spezifikation nur für Floppy-Laufwerke im Fullspeed-Modus verwendet werden. Bei Entwicklung neuer Geräte jedweder Art soll CBI nicht mehr eingesetzt werden.

Vorsicht ist jedoch geboten bei der Auswertung der Descriptorinformationen, die ein Mass-Storage-Gerät liefert. Nicht selten behauptet ein Gerät von sich, nur das BOT-Protokoll zu unterstützen, besitzt aber dennoch einen zusätzlichen Interrupt-Endpunkt und ist scheinbar auch in der Lage, wie ein CBI-Gerät zu operieren. Mehr noch, der erste BOT-konforme Transfer scheint auf diesen Geräten zum Scheitern verurteilt, was bei der Treiberimplementierung zu beachten ist.

Überhaupt existieren innerhalb der Mass-Storage-Klasse einige Kuriositäten oder auch einfach Fehlimplementierungen, die den Umgang mit diesen Geräten nicht eben einfacher gestalten. Eine gute Quelle für Informationen ist hier [Axe], wo die verbreitetsten Probleme zusammengetragen wurden.

4.6.4.2 Treibersoftware

Dateisystemtreiber Für den Betrieb von Mass-Storage-Geräten müssen Treiber auf verschiedenen Ebenen zusammenarbeiten. Soll sinnvoll mit einem Speicherstick oder einem ähnlichen Medium umgegangen werden, so muss die Firmware des AT90USB1287 in der Lage sein, mit dem auf dem Medium vorhandenen Dateisystem umzugehen.

Ein relativ einfaches und weit verbreitet Dateisystem ist das FAT-System, welches ab Werk auf vielen Mass-Storage-Geräten Verwendung findet. Erfreulicherweise existiert hierfür auch bereits eine Vielzahl an frei verfügbaren Dateisystemtreibern, welche auch für Kleinstsysteme geeignet sind.

Somit kann hier auf eine komplette Neuimplementierung der Software für die Dateisystemebene verzichtet werden.

Die Wahl beim Dateisystemtreiber fällt im Rahmen dieser Arbeit auf DOSFS[14], welcher eine vergleichsweise einfache Handhabung und effiziente Speichernutzung mitbringt. Er bietet der Applikation den gewohnten Zugriff auf Dateien und Verzeichnisse. Dabei kann er sowohl lesende als auch schreibende Zugriffe auf dem Dateisystem ausführen. Was die Nutzung des Treibers aus Applikationssicht angeht, so sei an dieser Stelle auf die dem Treiber beiliegenden Hinweise beziehungsweise auf den sich auf der beiliegenden CD befindenden Sourcecode verwiesen. Auf eine genauere Darstellung der Funktionsweise wird hier verzichtet, da das FAT-System nicht näherer Bestandteil dieser Arbeit ist.

Wichtig ist jedoch die Kenntnis über die Funktionen, die der Treiber als Schnittstelle zu der unter ihm arbeitenden Schicht voraussetzt. Hier begnügt sich DOSFS mit nur zwei Funktionen, genannt DFS_ReadSector() und DFS_WriteSector(). Diese dienen jeweils zum Lesen beziehungsweise Schreiben eines Sektors des Speichermediums. Listing 4.32 zeigt die Deklarationen der beiden Funktionen.

Listing 4.32: Deklariationen der beiden von DOSFS benötigten Schnittstellenfunktionen

```
1 uint32_t DFS_ReadSector(uint8_t unit, uint8_t *buffer, uint32_t sector,
  uint32_t count);
2 uint32_t DFS_WriteSector(uint8_t unit, uint8_t *buffer, uint32_t sector,
  uint32_t count);
```

SCSI-Ebene DFS_ReadSector() liest, startend mit Sektornummer sector, eine durch count gegebene Anzahl Sektoren in den durch *buffer angegebenen Puffer. DFS_WriteSector() funktioniert entsprechend in die andere Richtung: Aus dem Puffer werden Daten auf das Medium geschrieben. Den Erfolg der Aktionen geben beide Funktionen durch einen Returnwert von Null an DOSFS zurück. DOSFS unterscheidet zusätzlich verschiedene Medien anhand einer Unit-Nummer. Somit kann zum Beispiel zwischen den verschiedenen Slots eines Cardreaders differenziert werden. Für die verschiedenen Nummern muss daher später beim Einsatz mehrerer Geräte eine geeignete Adressumsetzung stattfinden. Die benötigten SCSI-Kommondos wurden

[14] Frei zu beziehen bei [Dri]

entsprechend der Spezifikation [Speg] (analog zum Vorgehen bei den USB Device Requests) als C-Strukturen implementiert.

Zusätzlich zu den einfachen Lese- und Schreiboperationen mussten noch einige zusätzliche Kommandos umgesetzt werden, die von einigen Geräten zur Initialisierung benötigt werden. Hierzu gehört beispielsweise das Abfragen der Mediengröße, obwohl diese für den Betrieb des DOSFS-Treibers keine weitere Rolle spielt. Der Treiber operiert ausschließlich auf einem bereits vorhandenen Dateisystem (beziehungsweise dessen File Allocation Table), über welches die verwendbare Mediengröße bereits gegeben ist.

Die beiden durch DOSFS benötigten Routinen zum sektorweisen Lesen und Schreiben erstellen eine passende SCSI-Kommandostruktur und reichen diese an den eigentlichen Mass-Storage-Treiber weiter.

Mass-Storage-Treiber Als Schnittstelle zur darüberliegende SCSI-Ebene stellt der Mass-Storage-Treiber die Funktion wrapCommand() zur Verfügung. Diese verpackt die SCSI-Kommandos in einem *Command Block Wrapper* und führt dann den eigentlichen Transfer durch. Die Datenübertragung stützt sich letzten Endes auf die durch *libAT90USB* zur Verfügung gestellten Grundfunktionen usb_recv() und usb_send().

Dieser Abriss der Funktionalität soll hier zur Vermittlung des Grundverständnisses der Arbeitsweise des Treibers genügen. Genauere technische Einzelheiten sind dem kommentierten Sourcecode der beiliegenden CD zu entnehmen. Zusätzlich verdeutlicht noch einmal Abbildung 4.4 die Aufrufstrukturen bei Verwendung von Mass-Storage-Geräten mit FAT-Dateisystem.

Ferner verfügt auch der Mass-Storage-Treiber über eine Funktion, welche die grundsätzliche Eignung eines Gerätes zum Betrieb mit dem Treiber feststellen kann. Im Falle des Mass-Storage-Treibers heißt diese storage_probe(). Bei Auffinden eines Interfaces mit Mass-Storage-Fähigkeiten liefert sie die Anzahl der auf dem Gerät zur Verfügung stehenden Einheiten (Units) – bei einem Cardreader zum Beispiel die Anzahl der vorhandenen Slots – zurück. Am Beispiel des Cardreaders lässt sich dabei jedoch noch eine weitere wichtige Erkenntnis gewinnen. Zum einen muss sich nicht unbedingt hinter jeder Einheit ein Medium verbergen – der fragliche Slot des Geräts, welcher mit der Unitnummer assoziiert ist, könnte ja zum Anschlusszeitpunkt leer sein. Zum anderen bedeutet aber auch ein vorhandenes Medium nicht zwangsläufig eine erfolgreiche Benutzung: Ist kein FAT-System auf dem Medium vorhanden, so kann der DOSFS-Treiber natürlich auch nicht mit dem Medium umgehen.

Die beiliegende CD enthält ein Beispielprogramm, welches bei Anschluss eines Mass-Storage-Gerätes dessen Inhalt anzeigt. Sollte dann noch ein zwei-

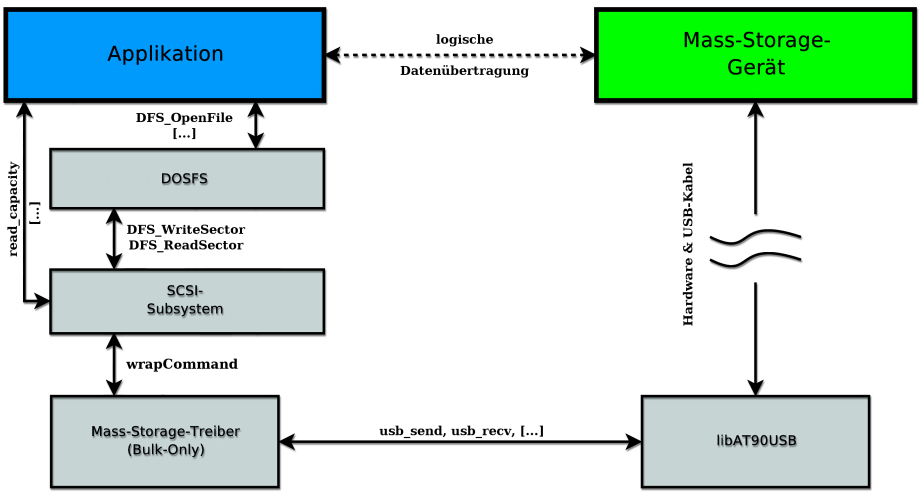


Abbildung 4.4: Aufrufstruktur und Datenfluss bei Mass-Storage-Nutzung

tes Gerät angeschlossen werden, so kopiert die Applikation eine Datei vorher festgelegten Namens von Medium Nummer eins auf Medium Nummer zwei. Abbildung 4.5 zeigt eine Beispielsitzung des Programms nach Anschluss eines USB-Sticks.

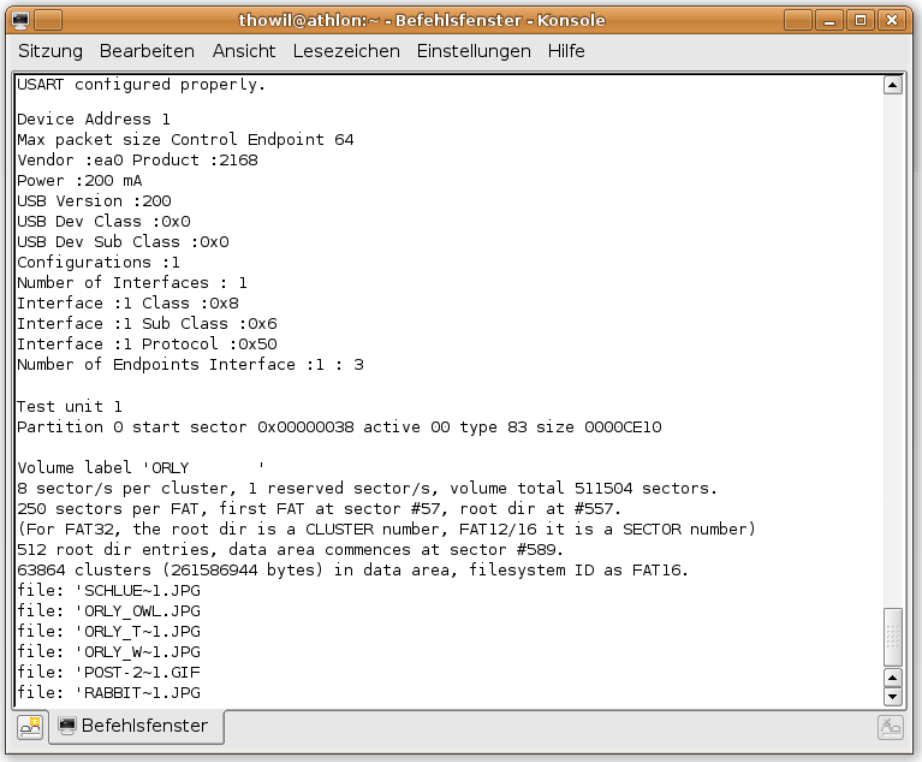


Abbildung 4.5: Beispielsitzung mit Listing des Root-Verzeichnisses.

Kapitel 5

Zusammenfassung

In dieser Arbeit wurde eine freie, unter der GPL stehende Softwarebibliothek entwickelt, welche einen einfachen Einsatz des USB-Hostmodus eines AT90-USB1287 erlaubt. Eine solche Software war bisher nicht verfügbar und kann nun als fundierte Basis für den Betrieb von USB-Geräten aller Art dienen.

Die Software wurde leicht konfigurierbar gehalten, so dass ihr Einsatz im Betrieb mit verschiedenen Taktraten und Platinendesigns möglich ist.

Über die Aufgabenstellung hinaus wurde die Bibliothek so entwickelt, dass ein Betrieb mehrerer USB-Geräte gleichzeitig möglich ist. Dies konnte erreicht werden, obwohl der AT90USB1287 augenscheinlich nicht für einen solchen Mehrgerätebetrieb entwickelt wurde.

Für den Mehrgerätebetrieb wurde, auf der Bibliothek aufbauend, ein Hubtreiber entwickelt. Hierbei zahlte sich das Vorgehen aus, Datenstrukturen und Nomenklatur bei Entwicklung der Bibliothek nahe an denen des Linux-Kernels auszurichten. Der Hubtreiber konnte auf diese Weise mit geringem Aufwand aus den Quellen des entsprechenden Kernel-Moduls portiert werden. Eine Portierung weiterer Gerätetreiber aus dem Linux-Kernel ist auf Basis dieser Erfahrung einfach möglich.

Als weiterer Beispieltreiber wurde ein Treiber für HID-Geräte entwickelt. Dieser erlaubt den Anschluss und das automatische Erkennen von HID-Geräten. Auf diese Weise sind Daten von beispielsweise Eingabegeräten einfach einzulesen und zu verarbeiten.

Zusätzlich wurde auch ein Treiber für Geräte der Mass-Storage-Klasse implementiert. Zusammen mit einem Dateisystemtreiber erlaubt dieser den lesenden und schreibenden Zugriff auf die weit verbreiteten USB-Massenspeicher-Geräte. Der Treiber nutzt zudem auch die Mehrgerätefähigkeit der Bibliothek. Eine dieser Arbeit beiliegende Applikation ermöglicht so beispielsweise das Kopieren von Dateien von einem angeschlossenen Gerät auf ein weiteres.

Ferner wurde dem Leser eine Experimentierplatine vorgestellt, deren Aufbau er zur schnellen Entwicklung eigener Projekte nutzen kann. Um dies so weit wie möglich zu vereinfachen, befinden sich im Anhang sowohl Ätzvorlagen als auch Bestückungslisten für die Platine. Im üblichen Fall kann das Design einfach übernommen werden, da alle Pins des AT90USB1287 über Wannenbuchsen zur Verfügung stehen.

Im Laufe der Entwicklung eigener Treiber und Programme kommt dem Leser auch die in Kapitel 2 erfolgte Aufarbeitung der USB-Spezifikation zugute. Hier wurden die für die Arbeit mit AT90USB1287 relevanten Teile der Spezifikation erklärend zusammengefasst. Auch die Erläuterung zur Hardware des AT90USB1287 in Kapitel 3 empfiehlt sich als Nachschlagewerk bei der Entwicklung eigener Projekte. Das Kapitel bietet eine, im Vergleich zum vorläufigen Datenblatt des AT90USB1287 verbesserte, Detailbeschreibung der relevanten Teile der USB-Einheit.

Betrachtet man rückblickend den Entwicklungsprozess, so bleibt festzustellen, dass vor allem die mitunter ungenaue Dokumentation der Hardware den Autoren unnötig Steine in den Weg legte. Auch ist anzumerken, dass die Hardware durch die fehlende Möglichkeit zur Generierung der Lowspeed-Präambel und – allem voran – der fehlenden Kontrolle über das Data-Toggle einer Pipe in ihren Möglichkeiten eingeschränkt ist. Gerade Probleme mit dem Data-Toggle sorgten immer wieder für einen fehlerhaften Verlauf der Kommunikation. Letztlich konnte jedoch die korrekte Handhabung der Toggles durch die Bibliothek gewährleistet werden.

Trotz der genannten Einschränkungen stellt der AT90USB1287 – vor allem mit der hier entwickelten Software – eine äußerst interessante Plattform für Entwicklungen rund um USB dar. Durch seine günstigen Beschaffungskosten und das breite Softwareangebot ist er sowohl für kommerzielle als auch für private Nutzer eine geeignete Wahl für Projekte, welche Unterstützung für USB benötigen. Die zusätzliche Möglichkeit, ihn wahlweise auch im Devicemodus zu betreiben, erweitert das Feld der Anwendungsmöglichkeiten abermals drastisch.

Dem USB-Standard selbst scheint ein weiter anhaltender Siegeszug gewiss. Dank seiner Variabilität und Einfachheit für den Nutzer wird er auch in Zukunft in einer immer weiter wachsenden Anzahl von Geräten Verwendung finden. Auch hält USB inzwischen Einzug in zunächst nicht verwandte Standards. Als Beispiel hierzu kann die ExpressCard des PCMCIA-Gremiums angeführt werden. Dort dient USB-2.0 neben PCI Express als Anschlussmöglichkeit der ExpressCard an das Host-Chipset.

Aber auch neben dem PC-Bereich wird die Zahl der Anwendungen von

USB weiter steigen. Egal ob Gamepad für Spielkonsole oder Verbindung von Digitalkamera zu Drucker – wie schon einleitend gesagt, verwenden auch immer mehr Kleinst- und Embeddedgeräten USB zur direkten, eigenständigen Datenübertragung.

Für eine einfache und kostengünstige Entwicklung solcher Geräte auf Basis des AT90USB1287 und der ihm verwandten Modelle wurde in dieser Arbeit gesorgt.

Kapitel 6

Glossar und Abkürzungsverzeichnis

Default Control Pipe Eine für alle USB-Geräte vorhandene Pipe, welche für Status- und Kontrollübertragungen sowie zur Gerätekonfiguration dient.

μC ..... **M**ikro**c**ontroller

USB-Systemsoftware Betriebssystemspezifische Software, welche die alleinige Kontrolle über den USB und den darüber stattfindenden Datenverkehr übernimmt.

A/D-Wandler .. **A**nalog/**D**igital-Wandler

ACK ..... Handshake-Paket, welches den erfolgreichen Empfang eines Datenpaketes quittiert.

ADB ..... **A**pple **D**esktop **B**us

ADDR ..... Ein Feld eines Paketes, welches die Adresse des angesprochenen Gerätes enthält.

ALU ..... **A**rithmetic **L**ogical **U**nit

Apple Desktop Bus Ein durch den Hersteller Apple definiertes serielles Bussystem zum Anschluss von externen Geräten an einen Computer.

Best-Effort-Prinzip Zusicherung an den Client, eine Datenübertragung im Rahmen der zur Verfügung stehenden Ressourcen vorzunehmen.

BOT ..... **B**ulk **O**nly **T**ransfer

BSD ..... **B**erkeley **S**oftware **D**istribution

Bulk Only Transfer Ein durch [Sped] definiertes Protokoll für Geräte der

Mass-Storage-Geräte Klasse, bei welchem zur Übertragung von Daten aus oder zu dem durch das Gerät zur Verfügung gestellten Massenspeicher lediglich Bulk Transfers eingesetzt werden.

Bus Enumeration Aktionsfolge, welche der Host zur Initialisierung eines an den Bus angeschlossenen Gerätes durchführen muss, damit dieses betriebsbereit wird und am Datenverkehr teilnehmen kann.

CAN ..... **C**ontroller **A**rea **N**etwork

CBI ..... **C**ontrol **B**ulk **I**nterrupt

CBW ..... **C**ommand **B**lock **W**rapper

Command Block Wrapper Eine durch [Sped] definierte Datenstruktur, in welcher SCSI-Kommandos zur Kommunikation mit BOT-Geräten für den Versand über USB gekapselt werden.

Command Status Wrapper Eine durch [Sped] definierte Datenstruktur, welche Statusrückgaben von BOT-Geräten enthält.

Composite Device Ein USB-Endgerät, welches mittels mehrerer Interfaces mehr als eine Funktion für den Anwender bereitstellt. Im Gegensatz zu Compound Devices handelt es sich jedoch bei Composite Devices bezüglich des Busses um ein einziges Geräte, welches beispielsweise auch nur eine Adresse belegt.

Compound Device Ein Gerät, bei welchem in einem Gehäuse mehrere USB-Geräte angeschlossen an einen Hub verbaut wurden. Dies könnte beispielsweise eine Tastatur mit eingebautem Touchpad sein.

Control Bulk Interrupt Ein durch [Spea] definiertes Protokoll für Mass-Storage-Geräte, bei welchem zur Kommunikation über USB Control, Bulk und Interrupt Transfers eingesetzt werden.

CPU ..... **C**entral **P**rocessing **U**nit

CRC5/16 ..... Paketfelder, welche CRC-Checksummen für die übertragenen Daten enthalten.

CRC ..... **C**yclic **R**edundancy **C**heck

CSW ..... **C**ommand **S**tatus **W**rapper

Cyclic Redundancy Check Verfahren zur Prüfsummenbildung, um Fehler bei einer Datenübertragung erkennen zu können.

D+ und D− .... Zur Datenübertragung genutzte Adern des USB-Kabels.

Data-Toggle .... Als Sequenznummer genutztes Bit innerhalb der PID

von Datenpaketen.

DATA0/1-Paket Enthält Nutzdaten, welche über den Bus zu übertragen sind.

DATA ..... Feld von Datenpaketen, welches Nutzdaten aufnimmt.

Descriptor ..... Datenstruktur, mittels welcher USB-Geräte sowie deren Fähigkeiten und Eigenschaften beschrieben werden.

Differentielle Signalisierung Eine Art der Signalübertragung über Kabel, bei welcher ein Adernpaar eingesetzt wird. Das Signal ergibt sich aus der Differenz der beiden auf den Adern geführten Spannungen.

DIL ..... **D**ual **I**n-**L**ine

Downstream-Port Anschluss eines Hubs oder Hosts, durch welchen weitere Geräte mit dem Bus verbunden werden können.

DPRAM ..... **D**ual **P**orted **RAM**

EEPROM ..... **E**lectrically **E**rasable **P**rogrammable **R**ead **O**nly **M**emory

End Of Packet . SE0-Signal einer Dauer von etwa zwei Bitzeiten, gefolgt von einem Übergang in den Zustand *J*. Dies markiert das Ende eines Paketes auf dem Bus.

Endpoint (auch Endpunkt) Quelle oder Ziel einer jeden über USB stattfindenden Datenübertragung auf Geräteseite. Ein Gerät kann mehrere Endpunkte besitzen.

ENDP ..... Ein Feld eines Paketes, welches die Nummer des angesprochenen Endpunktes enthält.

EOP ..... **E**nd **O**f **P**acket

FIFO ..... **F**irst **I**n **F**irst **O**ut

Force Feedback Eingabegeräte für Spielkonsolen oder PCs, welche den Spieler der Wirkung einer physikalischen Kraft aussetzen. Ein bekanntes Beispiel sind Lenkräder mit eingebauten Motoren zur Simulation der wirkenden Lenkkräfte bei Rennsimulationen.

Frame ..... Ein Intervall der Dauer 1ms, welches Geräte und Host als gemeinsame Zeitbasis nutzen.

Fullspeed ..... Die bei USB mögliche Datenrate von 12Mbit/s

Function ..... Ein USB-Endgerät, welches dem Anwender eine spezifische Funktion zur Verfügung stellt.

Geräteklasse ... Geräteklassen schreiben für Geräte, welche der fraglichen Klasse angehören, genaue Eigenschaften und Verhaltensweisen bezüglich des USB vor. Geräteklassen können sich weiter in verschiedene Unterklassen gliedern. Alle Geräte

einer Klasse (beziehungsweise Unterklasse) können mit dem gleichen Treiber betrieben werden.

GPL ..... GNU **G**eneral **P**ublic **L**icense

Handshake-Pakete Pakete, welche der Empfangsquittierung und Flusssteuerung dienen.

HID ..... **H**uman **I**nterface **Device**

Highspeed ..... Die bei USB mögliche Datenrate von 480Mbit/s

Host ..... Der Busmaster des USB, welcher die angeschlossenen Geräte verwaltet und die Zugriffe der Geräte auf den Bus koordiniert.

Hub ..... Ein spezielles USB-Gerät (kein Endgerät), welches zum Anschluss weiterer Geräte an den Bus dient.

Human Interface Device Ein speziell für die Mensch-Maschine-Interaktion entworfenes USB-Gerät, welches der in [Speb] gegebenen Spezifikation folgt. Typische Beispiele sind hier Mäuse oder Tastaturen mit USB-Anschluss.

I²C ..... **I**nter **I**C Bus

I/O Request Packet Eine Datenstruktur, welche der USB-Systemsoftware anzeigt, dass eine Clientsoftware Daten über den USB zu übertragen wünscht. Die konkrete Implementierung obliegt dem Betriebssystem.

I/O ..... **I**nput / **O**utput

IC ..... **I**ntegrated **C**ircuit

Interface ..... Logische Gruppierung von Endpoints, welche in ihrer Gesamtheit eine durch den Anwender nutzbare Gerätefunkion bereitstellen.

IN ..... Token-Paket, welches ein Gerät zur Übertragung von Daten zum Host auffordert.

IRP ..... **I**/O **R**equest **P**acket

ISR ..... **I**nterrupt **S**ervice **R**outine

Jitter ..... Unvermeidbare zeitliche Variation der Signallaufzeit auf einem Medium durch Störeinflüsse.

JTAG ..... **J**oint **T**est **A**ction **G**roup

J ..... Zustand des USB, bei welchem sich D+ und D− (senderseitig) nahe am Ruhepotential befinden.

K ..... Zustand des USB, bei welchen D+ und D− (senderseitig) im Vergleich zum Ruhepotential invertierte Pegel führen.

Least Significant bit Das niederwertigste Bit einer mehrere Bits umfassenden Zahlwert-Repräsentation.

LED ..... **L**ight **E**mitting **D**iode

Lowspeed ..... Die bei USB mögliche Datenrate von 1,5Mbit/s

LSb ..... **L**east **S**ignificant **b**it

Mass-Storage-Device USB-Gerät, welches der Mass-Storage-Klasse angehört. Diese Klasse ist in [Spee] definiert und beschreibt den Aufbau von USB-Massenspeichergeräten.

Message Pipe .. Eine bidirektionale Pipe, welche die Daten nach einem Request/Data/Status-Paradigma überträgt. Ein Beispiel für eine solche Pipe ist die Default Control Pipe. Die übertragenen Daten folgen einer durch die USB-Spezifikation genau festgelegten Struktur.

MLF ..... **M**icro **L**ead **F**rame

Most Significant bit Das höchstwertige Bit einer mehrere Bits umfassenden Zahlwert-Repräsentation.

MSb ..... **M**ost **S**ignificant **b**it

NAK ..... Handshake-Paket, durch welches ein Geräte eine temporäte Unfähigkeit signalisiert, Daten zu senden oder zu empfangen.

Non Return to Zero Invert Methode zur bitseriellen Datenübertragung, bei welcher eine Eins im Bitstrom durch gleichbleibenden Pegel, eine Null hingegen durch einen Pegelwechsel codiert ist.

Non Return to Zero Methode zur bitseriellen Datenübertragung, bei welcher der Null und der Eins im Bitstrom feste Pegelwerte zugeordnet sind.

NRZI ..... **N**on **R**eturn to **Z**ero **I**nvert

NRZ ..... **N**on **R**eturn to **Z**ero

On-The-Go ..... Erweiterung der USB-Spezifikation, durch welche USB-Endgeräte unter anderem eingeschränkte Host-Fähig- keiten erlangen und damit auch ohne PC miteinander kommunizieren können ([Speh]).

OTG ..... **O**n **T**he **G**o

OUT ..... Token-Paket, welches ein Gerät zur Annahme von Daten vom Host anweist.

Packet-Identifier Feld eines Paketes, welches den Typ eines Paketes (und auch dessen Format) bestimmt.

Payload ..... Bezeichnung für die bei einer Datenübertragung übersendeten Nutzdaten.

PDIP ..... **P**lastic **D**ual **I**n-line **P**ackage

Physical Interface Device Eine auf HID aufbauende Klassendefinition für USB-Geräte, welche speziell auf die Echtzeitbedürf-

nisse von Force-Feedback-Geräten ausgerichtet ist.

PID ..... Je nach Kontext **P**acket-**Id**entifier oder **P**hysical **I**nterface **D**evice.

Pipe ..... Logische Verbindung zwischen dem Host und einem Endpunkt, über welche Daten übertragen wird.

PLCC ..... **P**lastic **L**eaded **C**hip **C**arrier

Propagation Delay Beschreibt die Zeit, welche ein Signal benötigt, um von seiner Quelle zu seinem Ziel zu gelangen.

Präambel ..... Im Kontext des USB eine Folge von SYNC und PID, welche eine nachfolgende Paketübertragung im Lowspeed-Modus anzeigt.

PWM ..... **P**uls **W**idth **M**odulation, dt. Pulsweitenmodulation

RAM ..... **R**andom **A**ccess **M**emory

Request ..... Im USB-Kontext eine Bezeichnung für eine acht Byte große Datenstruktur, über welche der Host einem Gerät Kommandos erteilt.

RISC ..... **R**educed **I**nstruction **S**et **C**omputer

Root-Hub ..... Ein fest mit dem Host verbundener Hub.

Round Trip Time Zeit, welche nach dem Versand eines Datenpaketes bis zum Erhalt der darauf erfolgenden Antwort verstreicht.

SCSI ..... **S**mall **C**omputer **S**ystem **I**nterface

SE0 ..... **S**ingle **E**nded **0**

SETUP ..... Token-Paket, welches ein Gerät auf den Beginn eines neuen Control Transfers hinweist.

Single Ended 0 Zustand des USB, bei welchem die Adern D+ und D− Lowpegel führen.

Single-Ended Signalisierung Einfache Art der Übertragung elektrischer Signale über ein Kabel. Eine einzelne Ader führt dabei eine variable Spannung welche das Signal repräsentiert. Eine weitere Ader dient als Bezugspotential (üblicherweise Massepotential).

Small Computer System Interface Standardisierte parallele Busschnittstelle zur Datenübertragung.

SMD ..... **S**urface **M**ounted **D**evice

SOF ..... **S**tart **O**f **F**rame

SOP ..... **S**tart **O**f **P**acket

SPI ..... **S**erial **P**eripheral **I**nterface

SRAM ..... **S**tatic **RAM**

STALL ..... Handshake-Paket, durch welches ein Gerät einen Fehler signalisiert.

Start Of Frame Paket, welches den Beginn eines neuen Frames anzeigt.

Start Of Packet Übergang des Busses aus dem Ruhezustand in den Zustand *K*. Hiermit wird der Beginn eines Paketes auf dem Bus angezeigt.

Stream Pipe ... Eine unidirektionale Pipe, welche Daten als Bytestrom eines nicht durch die USB-Spezifikation beschriebenen Formats überträgt.

SYNC ..... Eine am Beginn jeden Paketes stehende Bitfolge, welche zur Synchronisation der Kommunikationspartner dient.

Token-Pakete .. Durch den Host generierte Pakete, welche der Bussteuerung dienen.

TQFP ..... **T**hin **Q**uad **F**lat **P**ack

UART ..... **U**niversal **A**synchronous **R**eceiver / **T**ransmitter

Underlength Packet Datenpaket mit einer Nutzdatenmenge, welche die maximal zulässige Nutzdatenmenge unterschreitet.

Universal Serial Bus Schnittstelle, welche für den Anschluss von bis zu 127 Peripheriegeräten an einen PC entwickelt wurde.

Upstream-Port Anschluss eines USB-Gerätes (Function oder Hub), mit dessen Hilfe das Gerät an den Bus angeschlossen wird. Jedes Gerät besitzt nach USB-Spezifikation genau einen Upstream-Port.

URB ..... **U**SB **R**equest **B**lock

USART ..... **U**niversal **S**ynchronous **A**synchronous **R**eceiver / **T**ransmitter

USB Request Block Konkrete Implementierung des IRP-Konzeptes bei Linux und Windows.

USB Reset ..... Zurücksetzen der USB-Komponente eines Gerätes. Ein solches Reset wird für ein Gerät durch einen mindestens 10ms andauernden SE0-Zustand des Busses erzwungen.

USB-IF ..... Bezeichnung für das USB-Implementers-Forum. Dies ist eine Organisation, die sich der Weiterentwicklung und Verbreitung von USB verschrieben hat.

USB ..... Universal Serial Bus

VBus ..... Bezeichnung für die Ader eines USB-Kabels, welche die positive Versorgungsspannung von 5V führt.

Zero Lenght Packet Datenpaket ohne Nutzdaten (Payloadlänge Null)

ZLP ..... **Z**ero **L**ength **P**acket

Literaturverzeichnis

[Axe] Axelson, Jan: *The Mass Storage Page.* http://www.lvr.com/mass_storage.htm.

[Dat] *Datasheet - AT90USB1286, AT90USB1287, AT90USB646, AT90USB647 Preliminary Complete, Revision D.* http://www.atmel.com/dyn/resources/prod_documents/ ↪ doc7593.pdf.

[Doc] *Flash Microcontrollers with Integrated USB Controller.* http://www.atmel.com/dyn/resources/prod_documents/ ↪ doc4036.pdf.

[Dri] *The DOSFS Free FAT12/FAT16/FAT32 Filesystem.* http://www.zws.com/products/dosfs/index.html.

[FAQ] *USB Cables, Connectors, and Networking with USB (FAQ).* http://www.usb.org/about/faq/ans5.

[Fora] *avrfreaks.net – Internationale AVR-Community.* http://www.avrfreaks.net/.

[Forb] *Mikrocontroller.net – Deutsche µC Community.* http://www.mikrocontroller.net/.

[Myk] Myklebust, Dr. Gaute: *The AVR Microcontroller and C Compiler Co-Design.* http://www.atmel.com/dyn/resources/ ↪ prod_documents/COMPILER.pdf.

[Spea] *Control/Bulk/Interrupt (CBI) Transport.* http://www.usb.org/developers/ ↪ devclass_docs/usb_msc_cbi_1.1.pdf.

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

[Spec] *Interface Association Descriptor Engineering Change Notice.* http://www.usb.org/developers/docs/
↪ InterfaceAssociationDescriptor_ecn.pdf.

[Sped] *Mass Storage Bulk Only 1.0.* http://www.usb.org/developers/
↪ devclass_docs/usbmassbulk_10.pdf.

[Spee] *Mass Storage Overview 1.2.* http://www.usb.org/developers/
↪ devclass_docs/usb_msc_overview_1.2.pdf.

[Spef] *Mini-B Connector Engineering Change Notice.* http://www.usb.org/developers/docs/edn1.pdf.

[Speg] *MultiMedia Command Set - 6 (MMC-6).* http://t10.org/ftp/t10/drafts/mmc6/mmc6r00.pdf.

[Speh] *On-The-Go Supplement to the USB 2.0 Specification.* http://www.usb.org/developers/onthego.

[Spei] *PullUp-/PullDown-Resistors Engineering Change Notice.* http://www.usb.org/developers/docs/resistor_ecn.pdf.

[Spej] *Universal Serial Bus Revision 1.1 specification.* http://www.usb.org/developers/docs/usbspec.zip.

[Spek] *Universal Serial Bus Revision 2.0 specification.* http://www.usb.org/developers/docs/usb_20_05122006.zip.

[Tooa] *Brian S. Dean's website – Private Webseite des Initiators von avrdude.* http://www.bsdhome.com/.

[Toob] *Homepage der avr-libc.* http://www.nongnu.org/avr-libc/.

[Tooc] *Homepage der binutils.* http://www.gnu.org/software/binutils/.

[Tood] *Homepage des GCC.* http://www.gnu.org/software/gcc/gcc.html.

[Tooe] *Homepage des WinAVR-Projekts.* http://winavr.sourceforge.net/.

[Toof] *Homepage von avrdude.* http://www.nongnu.org/avrdude/.

[Toog] *Homepage von gentoo Linux.* http://www.gentoo.org.

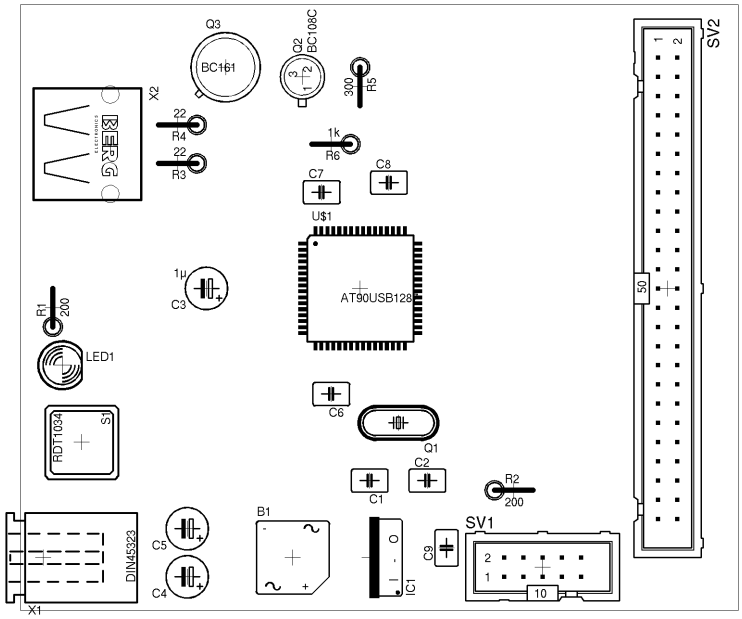
[UB02] Uwe Brinkschulte, Theo Ungerer: *Mikrocontroller und Mikroprozessoren.* Springer Verlag, September 2002. s.a. http://ipr.ira.uka.de/800.php.

Kapitel 7

Anhang

7.1 Materialien zum Experimentierboard

7.1.1 Bestückungsliste



Nr	Bauteil	Wert	Bauart
IC1	AT90USB1287		TQFP64
IC2	7805		78XXS
B1	Graez		
LED1	LED	grün	LED5mm
Q1	Crystal	16MHz	HC49/S
Q2	BC108C		
Q3	BC161		
S1	Taster		RDT1034
X1	Buchse		737992-5
X2	USB-Buchse	Typ A	
SV1	Wannenbuchse	10polig	ML10
SV1	Wannenbuchse	50polig	ML50
R1	Widerstand	200Ω	

Nr	Bauteil	Wert	Bauart
R2	Widerstand	1200Ω	
R3	Widerstand	22Ω	
R4	Widerstand	22Ω	
R5	Widerstand	300Ω	
R6	Widerstand	$1k\Omega$	
C1	Kondensator	$22pF$	
C2	Kondensator	$22pF$	
C3	Elko	$1\mu F$	
C4	Elko	$4.7nF$	
C5	Elko	$1\mu F$	
C6	Kondensator	$100\mu F$	
C7	Kondensator	$100\mu F$	
C8	Kondensator	$100\mu F$	
C9	Kondensator	$100\mu F$	

7.1.2 Ätzvorlage

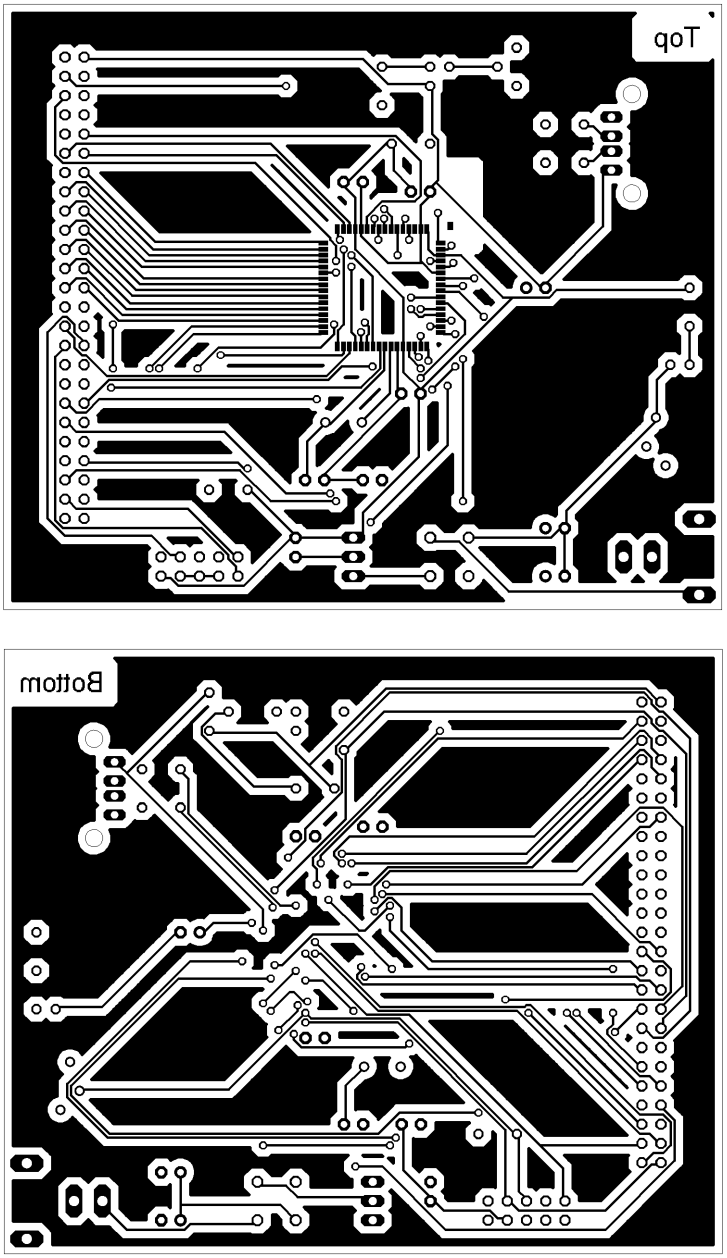
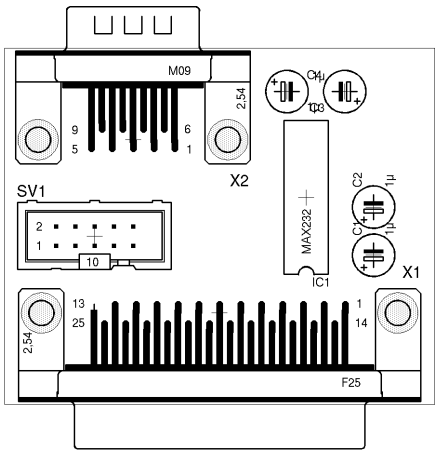


Abbildung 7.1: Ätzvorlage für das vorgestellte Experimentierboard – Oben: Top-Layer, Unten: Bottom-Layer.

7.2 Materialien für die Adapterplatine

7.2.1 Bestückungsliste



Nr	Bauteil	Wert	Bauart
IC1	MAX232		DIL16
C1	Elko	$1\mu F$	
C2	Elko	$1\mu F$	
C3	Elko	$1\mu F$	
C4	Elko	$1\mu F$	
SV1	Wannenbuchse	10polig	ML10
X1	SubD-Buchse	25polig	F25H
X2	SubD-Stecker	9polig	M05H

7.2.2 Ätzvorlage

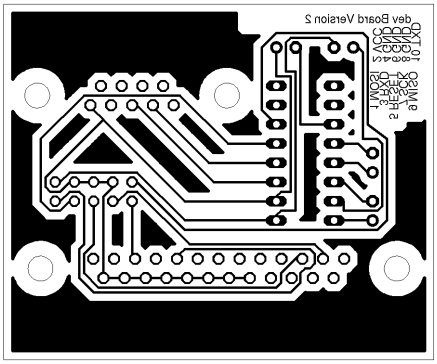


Abbildung 7.2: Ätzvorlage für den Adapter für Programmierung und Kommunikation über die RS232-Schnittstelle (Bottom-Layer).