Diplomarbeit

zum Thema

Die Kommunikation von Linux Applikationen mit generischer Hardware über das USB-Subsystem, praktisch realisiert am Beispiel einer USB-zu-Mikroprozessor und einer USB-zu-CAN Schnittstelle

Themensteller: Dr. Hubert Högl

Verfasser: Anatolij Gustschin
Matrikel Nr.: 799018

Abgabesemester: SS 2004

Abgabedatum: 17. Mai 2004

Erklärung

Diplomarbeit gemäß $ 31 der Rahmeprüfungsordnung für die Fachhochschulen in Bayern (RaPO) vom 18.09.97 mit Ergänzung durch die Prüfungsordnung (PO) der Fachhochschule Augsburg vom 15.12.94.

Ich versichere, dass ich die Diplomarbeit selbständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Augsburg, den 14. Mai 2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

1.1 Problemstellung

Im Zentrum der vorliegenden Diplomarbeit steht eine kleine Schaltung, die an den USB-Bus eines PC angeschlossen werden kann. Zur Ankopplung an den USB-Bus wird in dieser Schaltung der Baustein AN2131SC von Cypress Semiconductors verwendet, der aus einem frei programmierbaren Kern mit dem 8051 Prozessor besteht. Der Baustein verfügt über ausreichend freie I/O-Leitungen, die zur Steuerung von externer Hardware verwendet werden können - in unserem Fall sind daran der CAN-Controller SJA1000 (zur Kommunikation über das CAN-Feldbus) und ein Atmel ATmega8L Microcontroller (im folgenden AVR) angeschlossen. Zum einen dient diese Schaltung als Prototyp eines USB-CAN-Bus Adapters, zum anderen realisiert sie die Anbindung eines Microcontrollers ohne USB-Interface an den USB-Bus. Abbildung 1 zeigt den groben Aufbau der Prototypen-Platine.

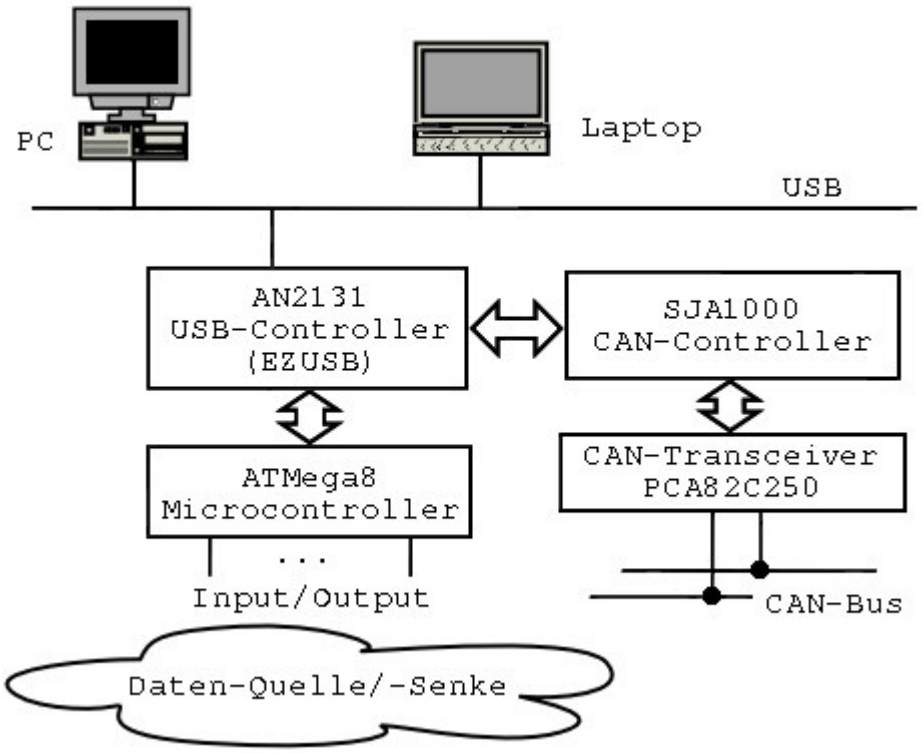

Abbildung 1: Grober Aufbau des Prototypen

Ziel der Diplomarbeit ist die Entwicklung von Software für die Prototypen-Platine unter Linux. Hierzu zählt die Entwicklung von:

- einem Gerätetreiber, der die Kommunikation mit dem AN2131 in das Betriebssystem integriert
- Software für 8051 Prozessor-Kern im AN2131, die als Schnittstelle zwischen USB und CAN-Bus agiert
- Software für 8051 Prozessor-Kern im AN2131, die eine Schnittstelle zwischen USB und Atmel ATmega8 Mikrocontroller realisiert
- Programmrahmen für Atmel AVR Microcontroller zur Kommunikation mit AN2131

Die entwickelte Software soll unter der Lizenz GPL (s. [7]) verfügbar sein.

1.2 Anforderungen

1.2.1 Anforderungen an USB-AVR-Schnittstelle

Die Datenübertragung von/zu AVR-Applikation soll über Puffer (64, 128, 256 KB) im AVR-SRAM erfolgen. Ein AVR-Puffer wird der Anwendung auf der Host-Seite über USB zugänglich gemacht. Die Anwendung soll in den Puffer schreiben, Pufferinhalt auslesen und Pufferstatus (Anzahl gültigen Bytes im Puffer) abfragen können. Hierzu sind entsprechende Funktionen bereitzustellen. Da der Pufferzugriff von AVR blockiert werden kann, muss in diesem Fall eine Fehlermeldung erfolgen. Die verwendete Variante von AVR - ATmega8L - verfügt über reduzierte Anzahl von I/O-Pins. Deshalb ist die Breite des Datenbusses zwischen AN2131 und AVR auf 4-bit beschränkt. Die Übertragung von Pufferdaten wird auf Aufforderung von AN2131 gestartet und erfolgt synchron. Weiterhin sollten mehrere USB-AVR Geräte an USB angeschlossen und von Software auf der Host-Seite unterstützt werden können.

1.2.2 Anforderungen an USB-CAN-Interface

Der in der Schaltung eingesetzte stand-alone CAN-Controller SJA1000 besitzt die grundsätzliche Struktur eines CAN-Protokollcontrollers nach dem BasicCAN-Konzept. Er kann auch im sogenannten PeliCAN-Modus betrieben werden, in dem er die Anforderungen der CAN-Spezifikation 2.0B erfüllt. Somit lassen sich im PeliCAN-Modus auch Nachrichten mit erweitertem Identifier von 29-Bit (Extended Frame Format) übertragen. Ein weiteres Feature von PeliCAN-Mode ist ein 64 Byte großer Empfangspuffer, der den Empfang neuer Nachrichten ermöglicht, während der bedienende Controller zuvor empfangene Nachricht ausliest. Außerdem ermöglicht dieser Mode eine erweiterte Fehleranalyse und Fehlerbehandlung. Daher sollte der CAN-Controller im PeliCAN-Modus betrieben werden.

Die Programmierung des 8051 Prozessor-Kerns im AN2131 erfolgt in Assembler. Unter Umständen können Teile des Programms auch in C geschrieben werden. Das USB-CAN Gerät soll den Anforderungen an USB-Geräte genügen, das heißt unter anderem, dass das Programm im AN2131 die Anfragen des Hosts zur Geräte-Identifikation und Abfragen der Geräteeigenschaften bearbeiten können soll.

1.2.3 Anforderungen an Hostsoftware

Wie jedes USB-Gerät, soll auch USB-CAN nach dem Anschließen an den USB-Bus erkannt und ein entsprechender Gerätetreiber geladen werden können. Normalerweise befindet sich ein Programm zur Gerätesteuerung (Firmware) in einem nichtflüchtigen Speicher innerhalb des Geräts. Dieses Programm wird nach dem Anschließen des Geräts ausgeführt, beantwortet die Anfragen des Hosts an das Gerät während der Erkennungsphase und sorgt anschließend für weitere Gerätesteuerung und Kommunikation mit dem Host. Der AN2131 Controller ist universell und wird in vielen verschiedenen USB-Geräten eingesetzt. Ein Programm im AN2131 repräsentiert das eigentliche Gerät. AN2131 kennt deswegen eine weitere Möglichkeit der Firmwarespeicherung. Die Firmware kann auf dem Host abgelegt werden und erst nach dem Anschließen des Geräts in den internen 8 KByte großen Programm-/Datenspeicher geladen und ausgeführt werden. Besonders während der Entwicklung ist diese Variante vorzuziehen. Die Software auf dem Host sollte deswegen unter anderem auch das Laden des Geräts mit entsprechender Firmware übernehmen.

1.3 Überlegungen zur Realisierbarkeit und Vorgehensweise

1.3.1 Gerätetreiber

Angefangen mit dem Kernel 2.4.x wird USB von Linux unterstützt (Entwicklungen für Kernel 2.2, so genannte back-ports, sind auch vorgenommen worden). Für die gängigen USB-Hostcontroller (EHCI, OHCI, UHCI) sind die Gerätetreiber vorhanden. Der Entwicklung eines spezifischen USB-Gerätetreibers steht somit nichts im Wege. Außerdem stellt Linux eine Schnittstelle zur Entwicklung von so genannten USB-User-Mode Treibern bereit. Auf dieser Schnittstelle baut die Bibliothek *libusb* (Funktionen für User-Mode USB Treiber) auf. Es soll daher die Möglichkeit erwogen werden, ob auf die Entwicklung von Kernel-Mode Treibern verzichtet werden kann. Dafür sprechen z.B. folgende Punkte:

- Reduktion der Entwicklungszeit (auch sollte man das Rad nicht immer wieder neu erfinden)
- man sollte immer, soweit es möglich ist, bereits vorhandenen und stabilen Kernelcode verwenden. Dieser ist meistens genug getestet worden und wird höchstwahrscheinlich von erfahrenen Programmierern gepflegt und weiterentwickelt
- an dem USB-Subsystem wird derzeit noch vieles geändert. Somit ist die Wahrscheinlichkeit sehr groß, dass der Code später modifiziert oder gar neu geschrieben werden muss, was natürlich mit weiterem Aufwand verbunden ist.

Die Argumente gegen User-Mode USB-Treiber könnten sein:

- unzureichende Performance
- nicht alle USB-Transferarten unterstützt (z.B. noch kein Interrupt-Transfer mit *libusb-0.1.7*)
- Treibercode/-daten können ausgelagert werden
- man will unbedingt Erfahrungen mit Linux-Kernel Programmierung sammeln (learning by doing)

Für das USB-AVR-Interface sollte ein User-Mode Treiber völlig ausreichen. Hier wird nur auf Aufforderung einer Host-Anwendung der Datentransfer bzw. Statusabfrage durchgeführt.

Beim USB-CAN-Interface treffen die Nachrichten vom CAN-Bus asynchron ein und müssen möglichst schnell an den Host weitergeleitet werden. Dieser sollte möglichst effizient eine sinnvolle Pufferung und Signalisierung an die Interessenten organisieren. Ein Kernel-Mode Gerätetreiber erscheint für USB-CAN-Interface besser geeignet zu sein. Außerdem unterstützt die derzeitige Version 0.1.7 von *libusb* keine Interrupt-Transfers, die z.B. für Gerätestatus-Abfragen von entscheidender Bedeutung sind.

1.3.2 Werkzeuge für AN2131 und AVR Programmierung

Die Entwicklung von Firmware für 8051-Kern des AN2131 kann unter Linux erfolgen. Der hierzu notwendige Assembler und C-Compiler ist auch für Linux verfügbar (*asx8051*, *sdcc*). Mit dem *avr-gcc* Paket und dem *avrdude* Programm-Downloader sind auch die Werkzeuge für Entwicklung von AVR-Programmen unter Linux vorhanden.

1.3.3 AN2131 Firmware-Download und Debugging

Unter Linux existieren bereits zwei Firmware-Downloader für AN2131: *ezusb2131* und *fxload*. *ezusb2131*-Downloader ist als Kernel-Mode Treiber realisiert, *fxload* baut auf der Kernel-Schnittstelle für USB-User-Mode Treiber auf. Beide könnten zum Firmware-Download ohne weiteres verwendet werden.

Es erscheint sinnvoll, ein Werkzeug zum Debuggen von AN2131-Programmen zu entwickeln. Dieses sollte z.B. die USB-Register, Endpoint-FIFOs und internes Programm-/Datenspeicher auslesen können. Eine Möglichkeit, in den Programm-/ Datenspeicher und Endpoint-FIFOs zu schreiben, ist auch erwünscht. Dasselbe Werkzeug sollte auch die Formulierung von USB-Control-Requests ermöglichen (analog der *Ez-Usb Control Panel* Applikation unter Windows).

2 Grundlagen

2.1 USB 1.1 Grundlagen

In diesem Abschnitt werden einige USB-Konzepte kurz erläutert. Die Beschreibung ist im wesentlichen an die Ausführungen zu USB-Grundlagen in [1] angelehnt.

2.1.1 USB-Topologie

Der USB besitzt als Topologie eine kaskadierte Sternstruktur. Im Ursprung befindet sich ein Host mit integriertem Root-Hub, von dem die Verzweigungen der ersten Ebene (Host-Ebene 0) ausgehen. Zu weiteren Verzweigungen und Bereitstellung von mehreren Bus-Anschlüssen dienen Hubs. Die Endgeräte (wie Tastatur, Drucker, Modem, etc.) werden Functions genannt. Insgesamt können bis zu 127 USB-Geräte (Functions) angeschlossen werden. Neben der Host-Ebene können bis zu vier weitere Ebenen aufgebaut werden (s. Abb. 2)

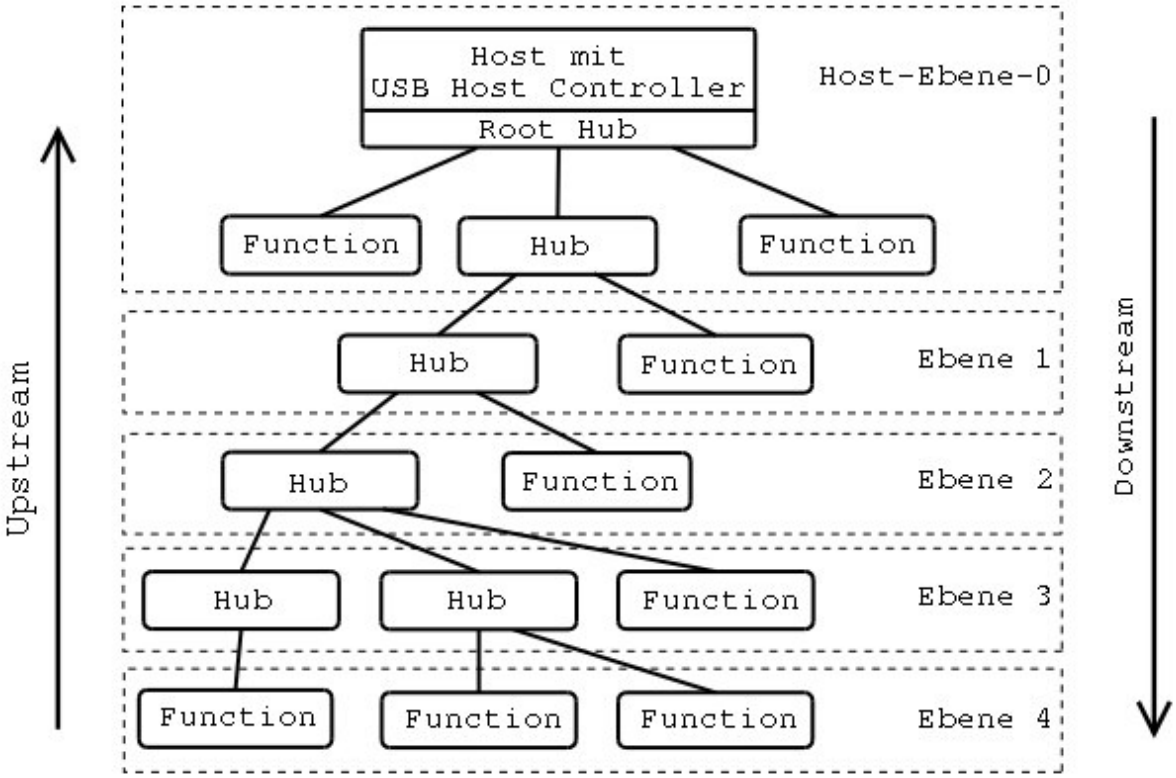

Abbildung 2: USB-Topologie

Der USB ist ein sog. Single-Master Bus. Das heißt, dass nur der Host die Master-Funktion besitzt und Datentransfers initiieren kann. Hubs und Functions dürfen ohne Anforderung durch den Host nicht senden. Ein Datentransfer vom Host zu Function (in Downstream-Richtung) erfolgt im Broadcast-Modus, d. h. alle vom Host gesendeten Datenpakete werden über Hubs an alle Functions verteilt. Die Empfänger-Adresse ist im sog. Token-Paket kodiert, das vor dem Datenpaket geschickt wird. Nur diejenige Function, die ihre Adresse im Token-Paket erkennt, antwortet. Diese Antwort wird dann von der Function zum Host (in Upstream-Richtung) übertragen. Die Antwort ist von der Transferart abhängig. Beim Datentransfer in Downstream-Richtung antwortet die Function mit Übernahme von empfangenen Daten in internes FIFO und anschließendem Absenden eines sog. Handshake-Pakets. Bei Aufforderung vom Host an die Function, die Daten zu

senden, antwortet die Function mit Absenden von Daten oder falls keine Daten zum Absenden vorliegen, mit einem Not-Acknowledge-Token. In diesem Fall kann der Host zu einem späteren Zeitpunkt die Daten von der Function anfordern.

2.1.2 Endpoint- und Pipe-Konzept

Physikalisch besteht die Verbindung zu einem USB-Gerät lediglich aus zwei Leitungen (D+,D-). Der USB unterstützt jedoch die Einrichtung mehrerer logischer Datenkanäle, den sogenannten Pipes. Jede Pipe endet in einem Endpoint. Jedes USB-Gerät kann mehrere Endpoints und somit mehrere Pipes unterstützen. Ein Endpoint ist physikalisch gesehen eine FIFO mit einer festgelegten Tiefe, welche USB-Daten empfangen oder senden kann. Bei der Adressierung von USB-Daten auf dem Bus wird neben der 7-Bit USB-Adresse auch eine 4-Bit breite Endpoint-Nummer geschickt, wodurch die angesprochene FIFO in einem USB-Gerät eindeutig identifiziert werden kann. Weiterhin dient die Richtung eines Datentransfers implizit als zusätzliches Adressierungsmerkmal für den entsprechenden Endpoint. Dieses Konzept erlaubt die Einrichtung von mehreren, logisch unabhängigen Geräten innerhalb eines physikalischen USB-Geräts.

Alle USB-Geräte müssen einen Control-Endpoint EP0 unterstützen. Dieser Endpoint ist der einzige bidirektionale Endpoint, über den Daten sowohl empfangen als auch gesendet werden können. Alle anderen Endpoints arbeiten unidirektional und können Daten entweder empfangen oder senden.

Die Richtung eines Datentransfers wird immer aus Sicht des Hosts bestimmt. Endpoints in Geräten, die Daten empfangen können, werden OUT-Endpoints genannt. Datentransfers zu OUT-Endpoints (OUT-Transfers) erfolgen immer in Downstream-Richtung. Bei einem Datentransfer in Upstream-Richtung von Function zu Host spricht man von einem IN-Transfer. Function-Endpoints, die nur Daten senden können, heißen entsprechend IN-Endpoints.

2.1.3 Transferarten und deren Zuordnung zu den Endpoints

Um die unterschiedlichen Geräte und Applikationen zu unterstützen, definiert die USB-Spezifikation vier verschiedene Transferarten:

- Control-Transfer wird für spezielle Anfragen (Requests) an ein USB-Gerät benutzt. Dies geschieht insbesondere in der Konfigurationsphase (unmittelbar nach dem Anschließen eines USB-Geräts. Control-Transfer wird auch für die Übermittlung von Kommandos vom Host an das USB-Gerät verwendet. Einem Control-Transfer folgt typischerweise weiterer Datentransfer (Control-Daten, Setup-Daten). Die Daten werden immer garantiert angeliefert, da Control-Transfers integrierten Fehlerkorrektur des USB-Protokolls unterliegen.
- Interrupt-Transfer wird für Geräte verwendet, die in einem PC klassische Interrupts auslösen würden. Derartige Geräte müssen per Polling abgefragt werden, da USB keine Hardware-Interrupts unterstützt. So wird z.B. bei einer USB-Tastatur kein Interrupt beim Drücken einer Taste ausgelöst, sondern auf periodische Abfrage geantwortet. Das Polling-Intervall kann ein Vielfaches von 1ms betragen. Interrupt-Transfer ist für die Übertragung kleiner Datenmengen (Geräte-Status) vorgesehen. Interrupt-Transfers unterliegen der integrierten Fehlerkorrektur des USB-Protokolls und sind ab USB 1.1-Spezifikation für beide Richtungen definiert.
- Bulk-Transfer dient hauptsächlich der Übertragung großer Datenmengen, die nicht periodisch sind und keine Echtzeitanforderungen stellen. So ist z.B. bei einem Scanner die Übertragungsgeschwindigkeit wichtig, der genaue Zeitpunkt

der Datenübertragung spielt dagegen keine bedeutende Rolle. Der Bulk-Transfer unterliegt der Fehlerkorrektur des USB-Protokolls, d.h. die Daten werden garantiert fehlerfrei übertragen. Diese Transferart ist für beide Richtungen spezifiziert.

- Isochronous-Transfer ist für Daten vorgesehen, die Anforderungen an die Bandbreite stellen. So ist z.B. bei Übertragung von Audio-Daten zu einem Lautsprecher zeitliche Synchronität und eine hohe Kontinuität des Datenstroms erforderlich. Diese Übertragungseigenschaften sind wichtiger als eventuelle Übertragungsfehler. Isochrone Datentransfers unterliegen keiner Fehlerbehandlung. Somit kommt diese Transferart nur in Frage, wenn die Echtzeitübertragung wichtiger ist als absolut korrekte Datenübermittlung.

Dem Endpoint EP0 ist immer der Control-Transfer zugeordnet. Zu allen anderen Endpoints lässt sich jeweils eine beliebige Transferart frei zuordnen. Ein Gerät könnte z.B. folgendes Endpoint-Layout besitzen:

Endpoint-Nummer	Richtung	Endpoint-Adresse	Transferart
EP0	BIDIRECT	0x00	Control
EP1	IN	0x81	Interrupt
EP1	OUT	0x01	Bulk
EP2	IN	0x82	Bulk
EP3	OUT	0x03	Isochronous

Tabelle 2.1: Beispiel für Endpoint-Layout

Neben der 4-Bit Endpoint-Nummer dient die Richtungsangabe zur eindeutigen Identifizierung eines Endpoints. Deshalb konnte im obigen Beispielgerät der Endpoint EP1 zweimal vergeben werden. Zwei Endpoints EP1 in eine Richtung wären dagegen nicht erlaubt.

Die maximale Anzahl der pro Gerät unterstützten Endpoints ergibt sich aus der 4-Bit Endpoint-Nummer und der zugeordneten Richtung. Neben dem bidirektionalen Control-Endpoint EP0 können somit maximal 15 IN- und 15 OUT-Endpoints ein einem Gerät implementiert werden.

Der USB unterscheidet zwei verschiedene Pipe-Konzepte:

- Message Pipes
- Stream-Pipes

Daten, die über Message-Pipes übertragen werden, besitzen eine durch USB-Spezifikation definierte Struktur und werden von Firmware des jeweiligen Empfängers interpretiert. Message-Pipes werden mittels Control-Transfers realisiert und nicht periodisch durch den Host initiiert. Ein USB-Gerät muss den Host darüber informieren, ob die über Message-Pipe übertragene Control-Daten erfolgreich interpretiert werden konnten.

Der in jedem USB-Gerät vorhandene Endpoint EP0 ist immer mit dem Control-Transfer assoziiert. Die mit dem Endpoint EP0 etablierte Pipe ist somit immer eine Message-Pipe. Die hier übertragenen Daten sind durch die USB-Spezifikation als Standard-Device-Requests definiert.

Stream-Pipes übertragen Datenströme, die keine durch die USB-Spezifikation definierte Struktur besitzen. Sie eignen sich für das Übermitteln von aller möglichen Daten. Stream-Pipes arbeiten immer unidirektional und können mittels Bulk-, Isochronous- oder Interrrupt-Transfers realisiert werden.

2.1.4 Standarddeskriptoren

Um ein echtes Plug-and-Play zu verwirklichen, muss der Host ein neu an den USB angeschlossenes Gerät identifizieren können. Zur Identifikation und Beschreibung von physikalischen und logischen Eigenschaften eines Gerätes dienen Deskriptoren. Diese werden vom Host nach dem Anschließen abgefragt.

Ein Deskriptor ist ein durch Spezifikation definiertes Bytefeld, in dem die Eigenschaften eines Geräts codiert sind. USB 1.1 unterscheidet fünf verschiedene Arten von Standarddeskriptoren:

Deskriptor-Art	Code
Device-Deskriptor	0x01
Configuration-Deskriptor	0x02
String-Deskriptor	0x03
Interface-Deskriptor	0x04
Endpoint-Deskriptor	0x05

Tabelle 2.2: Codierung der Deskriptoren

Deskriptoren sind hierarchisch angeordnet (s. Abb. 3). Jedes Gerät besitzt genau einen Device-Deskriptor mit Beschreibung von grundlegenden Eigenschaften. Der Device-Deskriptor enthält unter anderem die Hersteller- und Geräte-ID, Geräteklasse, die Anzahl möglicher Konfigurationen und die FIFO-Tiefe des Control-Endpoints EP0. Jedes Gerät kann mehrere Konfigurationen unterstützen, jedoch können nicht mehrere Konfigurationen gleichzeitig aktiviert sein. Die Auswahl einer Konfiguration geschieht durch den Standard-Device-Request SetConfiguration. Für die Beschreibung unterstützter Konfigurationen besitzt ein Gerät ein oder mehrere Configuration-Deskriptoren, die dem Device-Deskriptor untergeordnet sind.

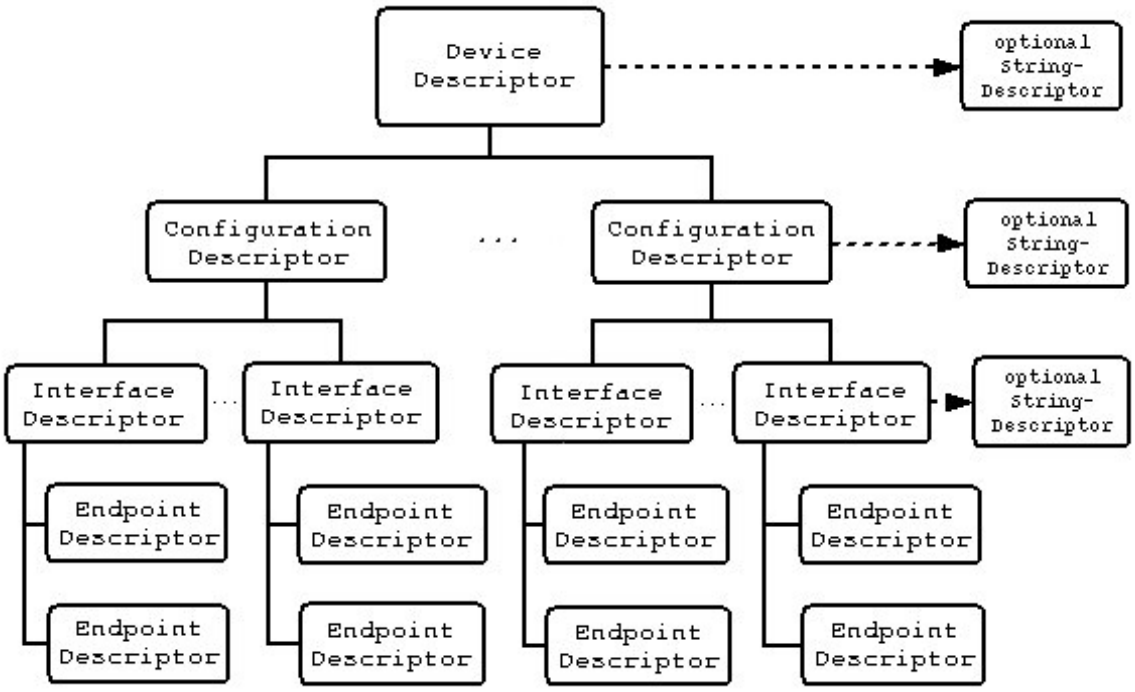

Abbildung 3: Hierarchie der Deskriptoren

Eine weitere Ebene tiefer befinden sich Interface-Deskriptoren. Innerhalb einer Konfiguration können mehrere Interfaces gleichzeitig existieren. Einzige Einschränkung bei deren Implementierung ist die Verwendung von sich gegenseitig ausschließenden Endpoints (ausgenommen Control-Endpoint EP0). Die Verwendung von mehreren Interfaces ermöglicht die Implementierung von USB-Geräten, die mehrere, gleichzeitig vorhandene, logische Funktionen unterstützen.

Zu einem Interface gehören ein oder mehrere Endpoints. Die physikalischen und logischen Eigenschaften der Endpoint-FIFOs werden mittels Endpoint-Deskriptoren beschrieben. Endpoint-Deskriptoren befinden sich auf der untersten Ebene der Deskriptor-Hierarchie. Alle in einem Gerät vorhandene Endpoints (außer Control-Endpoint EP0) werden jeweils durch einen Endpoint-Deskriptor beschrieben. Der Control-Endpoint wird bereits im Device-Deskriptor beschrieben.

String-Deskriptoren dienen für zusätzliche textuelle Beschreibung vom Hersteller, Gerät, Serien-Nummer usw. Außer String-Deskriptoren müssen alle anderen Deskriptortypen implementiert sein. Neben diesen Standard-Deskriptoren gibt es noch klassenspezifische Deskriptoren, die hier nicht erläutert werden, da sie in unserem Fall nicht vorkommen. Auf die Beispiele für Deskriptoren wird hier auch verzichtet, da später im Abschnitt "5.2 Definition von Deskriptoren" auf den Deskriptor-Aufbau näher eingegangen wird.

2.1.5 Standard-Device-Requests

Die Standard-Device-Requests stellen eine Untermenge der USB-Device-Requests dar. Alle USB-Geräte müssen auf Standard-Device-Requests antworten. USB-Device-Requests und die dazugehörigen Parameter bilden zusammen ein 8 Byte langes Datenfeld (SETUP-Data), das wie folgt belegt ist:

Offset	Feldbezeichnung	Größe in Byte	Belegung
0	*bmRequestType*	1	Bitmap
1	*bRequest*	1	Byte
2	*wValue*	2	16 Bit Wort
4	*wIndex*	2	16 Bit Wort
6	*wLength*	2	16 Bit Wort

Tabelle 2.3: Allgemeine Codierung der USB-Device-Requests

Alle hier erläuterten Requests werden über den Control-Endpoint EP0 abgewickelt. Anhand der im ersten Byte enthaltenen Bitmap kann entschieden werden, um welchen Request es sich handelt:

Bit-Position	Wert	Bedeutung
Bit 7	0 = vom Host zum Gerät 1 = vom Gerät zum Host	Datentransfer-Richtung während der optionalen Daten-Phase
Bit 6,5	00 = Standard-Request 01 = Klassenspezifischer Request 10 = Vendorspezifischer Request 11 = Reserviert	Request-Typ
Bit 4 .. 0	0000 = Device 0001 = Interface 0010 = Endpoint 0011 = Andere	Empfänger des Request (alle anderen Belegungen sind reserviert)

Tabelle 2.4: Codierung der Bitmap *bmRequestType*

Nun folgt eine Übersicht zu Standard-Device-Requests, die im Element *bRequest* übergeben werden:

bRequest	Codierung	Unterstützung
GET_STATUS	0x00	immer
CLEAR_FEATURE	0x01	immer
SET_FEATURE	0x03	immer
SET_ADDRESS	0x05	immer
GET_DESCRIPTOR	0x06	immer
SET_DESCRIPTOR	0x07	optional
GET_CONFIGURATION	0x08	immer
SET_CONFIGURATION	0x09	immer
GET_INTERFACE	0x0A	immer
SET_INTERFACE	0x0B	immer
SYNCH_FRAME	0x0C	optional

Tabelle 2.5: Standard-Device-Requests

Die hier aufgezählten Standard-Device-Requests sind für alle USB-Geräte definiert. Zusätzlich können noch klassen- und vendor-spezifischen Requests definiert werden.

2.1.6 Enumeration

Unter Enumeration wird die Identifizierung und Konfigurierung eines USB-Geräts verstanden, welches neu an den USB angeschlossen wurde. Im folgenden wird der Ablauf der Enumeration grob beschrieben. Zuerst versucht der Host einen ersten Datentransfer mit der neu angesteckten Function aufzubauen. Der erste Kommunikationsversuch wird immer auf Defaultadresse 0 des Geräts durchgeführt. Dazu schickt der Host einen Standard-Device-Request GetDescriptor mit dem Parameter DEVICE_DESCRIPTOR und wartet auf eine Antwort. Werden die ersten 8 Byte des Deskriptors übertragen, bricht der Host den Transfer ab und vermittelt der Function eine neue Adresse durch das Senden des Standard-Device-Requests SetAddress. Nach erfolgreichem Abschluss des kompletten Transfers besitzt die Function eine neue eindeutige Adresse (1-127) und wird ab diesem Zeitpunkt unter dieser Adresse angesprochen. Nun fragt der Host den vollständigen Device-Deskriptor ab und benutzt die darin enthaltenen Informationen für die Auswahl des passenden Gerätetreibers. Insbesondere die Felder Vendor-ID und Product-ID des Device-Deskriptors dienen dazu.

Der Host durchsucht mit diesen Informationen seine Treiber-Datenbank und versucht den entsprechenden Eintrag zu finden. Falls die Suche mit negativem Ergebnis endet, werden die Felder Klassen-Code, Subklassen-Code und Protokoll-Code ausgewertet. Die darin enthaltenen Informationen werden benutzt, um einen generischen Treiber für diese Geräteklasse zu laden.

Durch die Abfrage des Device-Deskriptors „weiß“ der Host, wie viele Konfigurationen dieses Gerät annehmen kann. Als nächstes erfolgt deshalb die Abfrage des ersten Configuration-Deskriptors mit allen dazugehörenden Interface- und Endpoint-Deskriptoren. Falls das Gerät weitere Konfigurationen unterstützt, werden die entsprechenden Deskriptoren ebenfalls abgefragt. Nun kennt der Host alle möglichen Konfigurationen und Eigenschaften des neuen Geräts und kann anhand verschiedener Kriterien eine auswählen.

Die Abfrage des Device- und Configuration-Deskriptors kann mehrfach erfolgen. Dies ist für das dynamische Nachladen der Treiberschichten in dem Host erforderlich. Jede neu geladene Treiberschicht kann die Deskriptoren abfragen und für sie relevante Information aus diesen verarbeiten.

Wurden alle Treiber geladen, weist der Host dem Gerät eine Konfiguration zu. Dies geschieht durch das Senden des Standard-Device-Requests SetConfiguration mit dem Wert für Konfiguration als Parameter. Die Firmware der Geräts aktiviert nun alle zu dieser Konfiguration gehörenden Endpoints. Besitzt eine Konfiguration ein oder mehrere Interfaces, werden diese anschließend durch den Request SetInterface aktiviert. Die Enumeration für das neu angeschlossene Gerät ist jetzt im Wesentlichen abgeschlossen. Der Gerätetreiber sorgt nun für die zyklische Abfrage der Interrupt-Endpoints und weitere Transfer- und Steuerungsaufgaben.

2.2 CAN-Bus Grundlagen

2.2.1 Wichtige Eigenschaften von CAN-Bus

Topologie

Der CAN-Bus besitzt eine linienförmige Topologie (s. Abb. 4). Die Anzahl der Bus-Teilnehmer ist nicht durch die Spezifikation beschränkt, sondern von der Leistungsfähigkeit der verwendeten Bustreiberbausteine abhängig. Bei zunehmender Busausdehnung nimmt die Datenübertragungsrate ab. Bei einer Buslänge von bis zu 40 m beträgt die Datenrate 1 MBit/s. Die Netzausdehnung von 1000 m beschränkt die Datenrate auf 80 KBit/s (vgl. 1.6.5 in [2]).

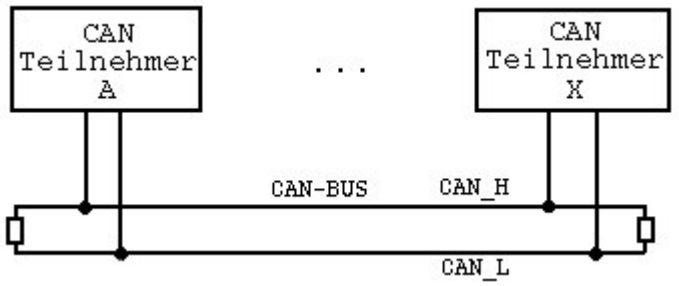

Abbildung 4: CAN-Bus Topologie

Nachrichtenorientiertes Protokoll

Der Datenaustausch erfolgt mittels kurzen Nachrichten (bis zu 8 Byte), welche zusätzlich eine Nachrichtenkennung (Identifier) beinhalten. Eine Nachricht kann gleichzeitig von allen aktiven Busteilnehmern empfangen werden (Broadcast-Betrieb). Jeder Teilnehmer prüft anhand des Identifiers, ob die empfangene Nachricht für ihn relevant ist.

Nachrichtenpriorisierung

Der Identifier einer Nachricht bestimmt gleichzeitig deren Priorität bezüglich des Buszugriffs. Somit ist der Buszugriff für besonders wichtige Nachrichten innerhalb einer kurzen Latenzzeit unabhängig von Busbelastung garantiert.

Multimaster-Fähigkeit und verlustlose, bitweise Busarbitrierung

Die Vergabe des Buszugriffsrechts erfolgt nicht zentralisiert. Sobald der Bus frei ist, kann jeder Busteilnehmer die Übertragung seiner Nachricht starten. Obwohl mehrere

Teilnehmer den Bus gleichzeitig belegen können, kommt es nicht zu einem Verlust der aufgeschalteten höchstpriorisierten Nachricht. Bei einem gleichzeitigen Buszugriff setzt sich die höchstpriorisierte Nachricht durch. Die parallel sendenden niederpriorisierten Knoten schalten sofort auf Empfang um und versuchen zu einem späteren Zeitpunkt ihre Nachrichten abzuschicken.

Geringe Latenzzeit

Bedingt durch die geringe maximale Nachrichtenlänge und die Priorisierung der Nachrichten ist der Buszugang mit sehr kurzer Latenzzeit möglich. Für die höchstpriorisierte Nachricht liegt diese bei maximal 130 (Standard Frame Format) bzw. 154 Bitzeiten (Extended Frame Format).

Sicherheit und netzweite Datenkonsistenz

Mehrere Fehlererkennungsmechanismen ermöglichen die Erkennung von verfälschten Nachrichten mit sehr hoher Wahrscheinlichkeit. Detektierte Fehler führen zu einer automatischen Wiederholung der Nachrichtensendung. Ein wesentlicher Aspekt der Datenintegrität eines verteilten Systems ist dessen systemweite Datenkonsistenz. Bei Prozeßsteuerungen müssen häufig mehrere beteiligte Knoten synchronisiert oder deren Aktionen auf gemeinsame Datensätze abgestimmt werden. Dies ist nur erreichbar, wenn alle beteiligten Knoten die Daten- und Synchronisationsnachrichten gleichzeitig und korrekt empfangen. Lokal gestörte Nachrichten müssen deshalb für alle Systemknoten als fehlerhaft bekannt gemacht werden. Im CAN-Protokoll ist hierfür ein Fehlersignalisierungsmechanismus definiert. Außerdem wird die Funktionalität von Netzknoten überwacht. Bei Überschreiten einer festgelegten mittleren Fehlerrate wird das Einwirken eines betroffenen Knotens auf das Netzwerk eingeschränkt oder der Knoten vom Netzwerk abgekoppelt (vgl. 1.6.5 in [2]).

2.2.2 CAN-Nachrichten

Jede CAN-Nachricht besteht aus bestimmten Bitfeldern. Dabei handelt es sich unter anderem um Arbitrationsfeld (Identifier), Datenfeld, CRC-Feld, und End Of Frame. Im Arbitrationsfeld ist die Priorität untergebracht, welche gleichzeitig die logische Adresse der Nachricht angibt. Bei Nachrichten im Standard Frame Format gemäß der Spezifikation 2.0A ist das Arbitrationsfeld 11-Bit lang. Nachrichten gemäß Spezifikation 2.0B (mit Extended Frame Format) besitzen ein 29-Bit langes Arbitrationsfeld (Abb. 5).

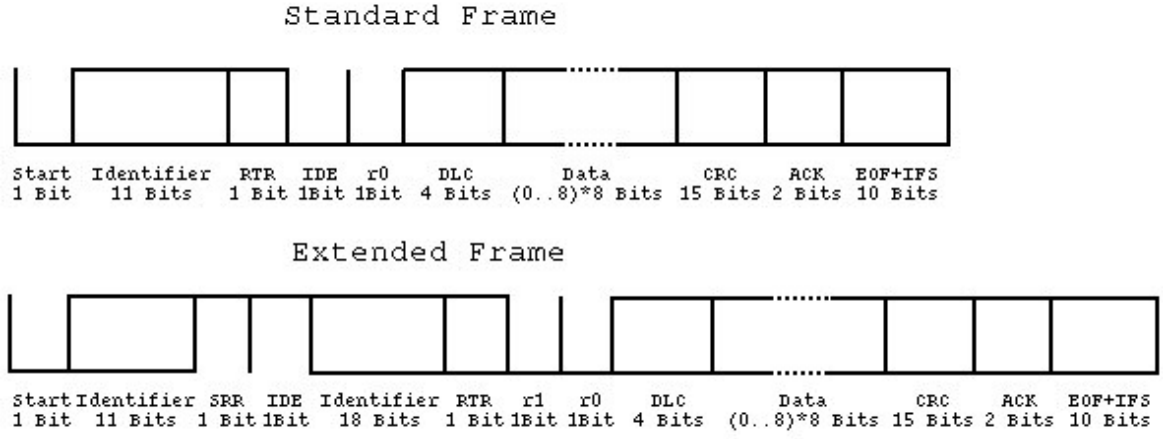

Abbildung 5: Aufbau von CAN-Nachrichten (Bildquelle [3])

Die einzelnen Bitfelder bedeuten:

- Startbit (1 Bit): Immer low (dominant) und kennzeichnet den Anfang einer Nachricht. Nach einer Idle-Zeit auf dem Bus dient die fallende Flanke zur Synchronisation der einzelnen Busknoten
- Identifier (11 Bit): Name und Priorität der Nachricht; kleinere Wert bedeutet höhere Priorität
- RTR (Remote Transmission Request, 1 Bit): Beim dominanten RTR handelt es sich um eine gewöhnliche Datennachricht, beim rezessiven RTR (high) - um ein Remote-Frame. Remote-Frame ist eine Nachricht, die selbst keine Daten enthält, sondern nur den Sender für diese Daten auffordert, eine Nachricht mit den gerade aktuellen Daten auszusenden.
- Control (6 Bit): Das erste Bit in diesem Feld ist das IDE (Identifier Extension) Bit. Bei Standard-Frames ist es immer dominant, d. h. es kommt nachfolgend keine Identifier-Erweiterung. Das Bit r0 ist reserviert, die weiteren 4 Bit codieren die Länge des nachfolgenden Datenfelds.
- Data (0 bis 8 Byte): Die Daten der Nachricht.
- CRC (15 Bit + 1 Bit CRC-Delimiter (high)): Enthält die Prüfsumme über alle vorangegangenen Stellen. Diese dient nur zur Fehlererkennung.
- ACK (2 Bit, high): Alle aktiven Busteilnehmer quittieren eine korrekt empfangene Nachricht mit einem dominanten Pegel in diesem ACK-Slot. Nur vom Sender wird ein rezessiver Pegel gesendet. Das Ausbleiben des dominanten Pegels wird vom Sender als Fehler interpretiert.
- EOF (7 Bit, high): End Of Frame kennzeichnet das Ende einer Nachricht.
- IFS (3 Bit, high): Inter Frame Space kennzeichnet den Zeitraum für das Übertragen einer korrekt empfangenen Nachricht in den Empfangspuffer des Controllers.
- Idle (>= 0 Bit, high): Der Bus ist unbenutzt, jeder Teilnehmer darf senden.

Extended Frames unterscheiden sich von Standard Frames in folgenden Positionen:

- SRR (1 Bit, high): Substitute Remote Request ersetzt das RTR-Bit, das bei Standard Frames an dieser Stelle steht und hat keine weitere Bedeutung.
- IDE (1 Bit, high): Identifier Extension zeigt an, dass noch weiteres 18-Bit Identifier-Feld folgt. Anschließendes RTR-Bit hat dieselbe Bedeutung wie bei Standard Frames.
- Control (6 Bit): r0 und r1 sind reserviert, weitere 4 Bit codieren die Länge des nachfolgenden Datenfelds.

Neben den Datennachrichten existieren noch die speziellen Nachrichten zur Fehlersignalisierung: Active Error Frame, Passive Error Frame und Overload Frame (vgl. 4.2.3 in [3]).

2.2.3 Arbitrierung beim Bus-Zugriff

Erkennt ein sendender Teilnehmer, dass der Bus bereits belegt ist, wird sein Sendewunsch bis zum Ende der aktuellen Übertragung verzögert. Beim gleichzeitigen Mehrfachzugriff von Teilnehmern auf den Bus wird automatisch der wichtigste (höher priore) Konkurrent ausgewählt. Die Arbitrierung erfolgt bitweise und ist verlustfrei, d. h., der Gewinner der Arbitrierung muss nicht seine Nachricht erneut von vorn senden. Dies wird auf folgende Weise realisiert: der logische Pegel Null muss auf dem Bus dominant und der logische Pegel Eins rezessiv sein. Eine von einem Teilnehmer gesendete Eins soll während der Arbitrierung von einer gesendeten Null eines anderen Teilnehmers überschrieben werden

können. Jeder Teilnehmer liest während des gesamten Sendevorgangs den logischen Wert, der sich auf dem Bus einstellt, zurück und vergleicht ihn mit dem gesendeten logischen Wert. Solange die beiden Werte übereinstimmen, wird weiter gesendet. Falls ein Teilnehmer den rezessiven Pegel sendet aber den dominanten Pegel auf dem Bus feststellt, erkennt er die verlorene Arbitrierung und unterbricht seinen Sendevorgang. Weiterhin schaltet er auf Empfang um, da die vom Gewinner gesendete Nachricht für ihn bestimmt sein könnte (vgl. 4.2.2 in [3]).

2.2.4 Ausnahmebehandlung

Das CAN-Protokoll garantiert einen sehr hohen Grad der Fehlererkennung. Die Mechanismen zur Fehlererkennung, Fehlerbehandlung und Fehlereingrenzung sind standardisiert und in jedem CAN-Controller implementiert. Jeder durch einen Netzteilnehmer erkannte Fehler wird sofort allen anderen Netzknoten durch einen Error Frame mitgeteilt. Nach der Fehlermeldung verwerfen diese die bis dahin empfangene Nachricht. Durch automatische Sendewiederholung wird versucht, den Fehler zu korrigieren. Bei einer anhaltenden lokalen Störung zieht sich der gestörte Teilnehmer schrittweise vom Busgeschehen zurück, damit die restlichen Busteilnehmer nicht durch ständige Error Frames gehindert werden.

2.3 Linux USB Subsystem und API für USB-Gerätetreiber

Die Grundlage aller USB-Treiber unter Linux ist die Kombination des USB-Subsystems (usbcore) mit einem der Hostcontrollertreiber (usb-uhci, usb-ohci). Neben dem integrierten Hub-Treiber enthält das USB-Subsystem eine Schnittstelle, über die Usermode Treiber auf USB zugreifen können. Das USB-Subsystem abstrahiert von hardwarespezifischen Details und erleichtert erheblich die Entwicklung von USB-Gerätetreibern. Mit einem sogenannten Upper-API stellt es gemeinsame Routinen für alle USB-Gerätetreiber zur Verfügung. Das Lower-API enthält gemeinsame Funktionen für alle Hostcontroller-Treiber (s. Abb. 6).

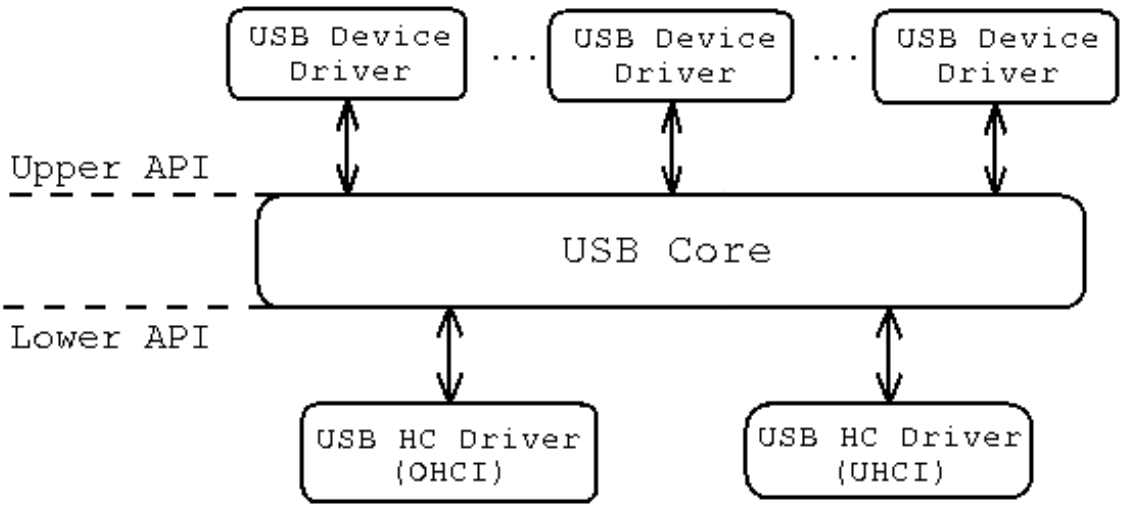

Abbildung 6: API-Schichten des USB-Subsystems von Linux (Bildquelle [4])

Die USB-Treiber werden beim Subsystem angemeldet und abgemeldet. Bei der Anmeldung werden neben dem Treibernamen zwei Eintrittspunkte registriert. Bei dem ersten Eintrittspunkt handelt es sich um eine Funktion *probe*, die beim Anschließen des

Geräts vom Subsystem aufgerufen wird und unter anderem die Initialisierung von Treiberstrukturen durchführt. Der zweite Eintrittspunkt ist eine Funktion *disconnect*, die nach dem Abstecken für das Aufräumen sorgt. Weiterhin können Funktionen für verschiedene Gerätedatei-Operationen und eine Minor-Nummer registriert werden. In diesem Fall werden die nachfolgenden 15 Minor-Nummern für den Treiber reserviert. Somit kann ein Treiber bis zu 16 ähnliche USB-Geräte bedienen.

Die Funktion *probe* erhält als Parameter einen Zeiger auf das Gerätekontext mit Zeigern auf alle Gerätedeskriptoren. Ein weiterer Parameter enthält die Interface-Nummer. *probe* kann anhand der Information aus den Deskriptoren entscheiden, ob das Gerät und Interface vom Treiber unterstützt wird. Für ein unterstütztes Gerät alloziert *probe* eine Treiberkontext-Struktur und liefert einen Zeiger auf diese als Rückgabewert.

Die Funktion *disconnect* erhält als Parameter einen Zeiger auf Gerätekontext und den von *probe* gelieferten Zeiger auf den Treiberkontext. Somit kann *disconnect* die Treiberkontext-Struktur freigeben.

Für den Datentransfer verwendet das USB-Subsystem eine sog. USB Request Block Struktur (URB). Eine URB-Struktur enthält alle notwendigen Parameter für eine USB Transaktion und wird an das USB-Subsystem übergeben. Das Ende des Transfers wird asynchron durch Aufrufen einer callback-Funktion signalisiert, welche im URB angegeben ist. Diese Funktion sollte möglichst schnell sein, da sie aus dem Interrupthandler des Hostcontrollers angesprungenen wird. Das USB-Subsystem stellt alle für URB-Operationen notwendige Routinen zur Verfügung (vgl. [4]).

3 Entwicklungssystem

3.1 Entwicklungsrechner und Softwareausstattung

Hardware

Als Entwicklungssystem wurde ein PC mit folgender HW-Ausstattung eingesetzt:

CPU:	Pentium II, 266 MHz
RAM:	192 MB
Chipset:	Intel i440LX AGPset
USB Controller:	82371AB PIIX4 USB rev. 1 (UHCI) OPTi 82C861 PCI to USB (OHCI)
CAN-PC Interface:	CPC-PCI / SJA1000D

Software

Betriebssystem:	Debian GNU/Linux 3.0 r1 "Woody"
Linux kernel:	2.4.18, 2.4.25
gcc:	2.95.4 20011002 (Debian prerelease)
avr-gcc:	3.0.3
avrdude:	4.2.0
sdcc:	2.3.0
asx8051:	1.70
Assembler as31:	2.0b3 (beta)
can4linux:	3.1

3.2 Aufbau des CAN-Bus Testsystems

3.2.1 CAN-PC Interface CPC-PCI und Bus-Kabel

Zum Testen der entwickelten Programme wurde ein einfaches CAN-Netzwerk mit zwei Teilnehmer-Knoten aufgebaut. Bei dem ersten Knoten handelt es sich um die USBCAN-Prototypen-Platine an dem USB-Bus des Entwicklungshosts, bei dem zweiten - um den Entwicklungshost selbst. Als CAN-Bus Schnittstelle wurde im Entwicklungshost eine PCI-Karte CPC-PCI verwendet (Abb. 7).

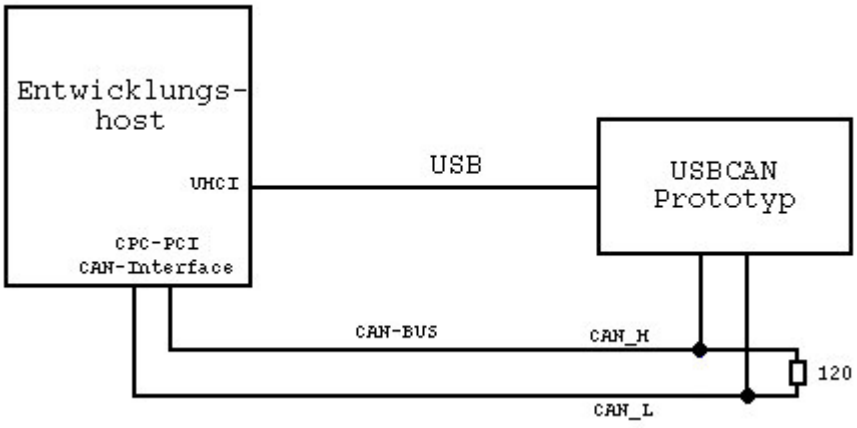

Abbildung 7: Aufbau des CAN-Bus Testsystems

Als Bus-Kabel wurde ein gewöhnliches Sub-D-Verbindungskabel (9-polig) eingesetzt. Die Belegung der Stecker ist wie folgt definiert:

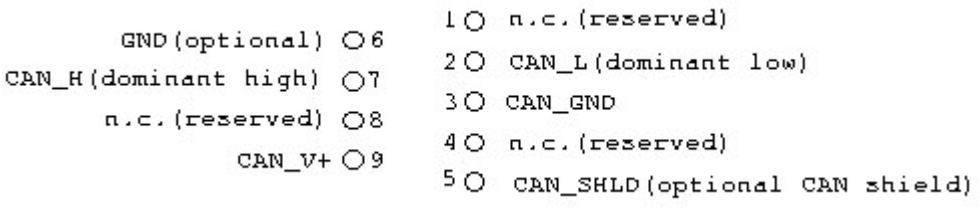

Abbildung 8: CAN Sub-D Steckerbelegung

3.2.2 Linux Treiber can4linux

CAN-PC Interface CPC-PCI wird unter Linux mit dem CAN-Bus Treiber can4linux unterstützt. Im folgenden wird kurz auf die Installation und Konfiguration von can4linux eingegangen (s. auch [13]).

Vor dem Entpacken sollte man ein Verzeichnis für Treiber-Code anlegen:

mkdir can4linux && cd can4linux
tar xzvf ../can4linux.3.1.tgz

Übersetzt wird der Treiber für jede vorgesehene Hardware getrennt. Um den Treiber für CPC-PCI zu übersetzten, wird *make* mit spezifiziertem *TARGET* aufgerufen:

make TARGET=CPC_PCI

Weiter müssen die Gerätedateien angelegt werden. Dies geschieht mit:

make inodes

Vor dem Laden des Treibers muss eine Konfigurationsdatei erstellt werden. Im Verzeichnis *etc* findet man Beispiele für Konfigurationsdateien. In unserem Fall dient die Datei *1-cpcpci.conf* als Basis. Wir legen nun unsere eigene Konfigurationsdatei an. Diese muss nach dem Rechnernamen benannt sein:

cd etc
cp 1-cpcpci.cong `uname -n`.conf
cd ..

Schließlich wird der Treibermodul geladen:

make load

Nachdem Treiber geladen wurde, wird er mit Werten aus Konfigurationsdatei konfiguriert. Der Treiber verwendet als Basisadresse und Interruptnummer für PCI-Karte die vom BIOS zugewiesenen Werte. Deswegen kann er die entsprechenden Werte aus Konfigurationsdatei nicht übernehmen. Wir ignorieren einfach die Fehlermeldungen bezüglich Base und IRQ beim Laden der Konfiguration.

In dem *examples* Verzeichnis findet man verschiedene Test-Applikationen. In unserem Fall wurden modifizierte *transmit.c* und *receive.c* für Testzwecke eingesetzt.

3.3 Entwicklung von zusätzlichen Hilfswerkzeugen

3.3.1 AN2131CTL für AN2131 Zugriff

Wie bereits im Abschnitt 1.3.3 erwähnt, wäre ein Werkzeug zum Debuggen von AN2131-Programmen hilfreich. *an2131ctl* wurde speziell für diese Zwecke entwickelt. Dieses Tool benutzt *libusb*-Funktionen zum Zugriff auf den USB. Neben dem lesenden und schreibenden Zugriff auf internen AN2131-Speicher und Endpoit-FIFOs können auch USB-Register des AN2131 ausgelesen und USB-Device-Requests formuliert werden. *an2131ctl* kann auch als Firmware-Downloader verwendet werden. Die Aktionen und Parameter werden über Kommandozeilenoptionen angegeben. Eine Übersicht zu Kommandozeilenoptionen befindet sich im Anhang E. Außerdem ist eine Man-Page zu *an2131ctl* nach der Installation verfügbar (*man an2131ctl*).

3.3.2 EZUSB_LDR Skript für das Linux-Hotplug-System

Eine der gestellten Anforderungen an Hostsoftware ist das Laden des angeschlossenen Geräts (USB-CAN, USB-AVR) mit entsprechender Firmware. Das Linux-Hotplug-System stellt bereits alle hierzu notwendigen Basis-Mechanismen wie das Laden des zugehörigen Gerätetreibers oder Ausführen von gerätespezifischen Setup-Skripts. Die Voraussetzung hierzu ist die Übersetzung des Kernels mit aktiviertem CONFIG_HOTPLUG.

Beim Anschließen oder Abziehen eines USB-Geräts wird vom Kernel-Hub-Demon *khubd* das Skript */sbin/hotplug* mit dem Parameter *usb* und einigen Umgebungsvariablen gestartet. Dieser sorgt für die Ausführung des */etc/hotplug/usb.agent*-Skripts, welches die Ausführung von benutzerdefinierten Skripten veranlassen kann. Die Umgebungsvariablen spezifizieren die auszuführende Aktion:

Variable	Werte
ACTION	"add", "remove"
PRODUCT	USB Vendor- und ProductID und Version (hexadezimal)
TYPE	Geräte-Klasse (dezimal)
INTERFACE	Klassen-Code für Interface 0

Falls Schnittstelle für User-Mode USB-Treiber "usbdevfs" vom Kernel unterstützt wird und entsprechendes Dateisystem eingebunden ist, werden zusätzlich Variablen DEVICE und DEVFS gesetzt. DEVICE enthält den Pfad zu der Gerätedatei, über die das angeschlossene Gerät angesprochen werden kann. Die Information in PRODUCT und DEVICE ist ausreichend um das Laden des Geräts mit Firmware durchzuführen. Das benutzerdefinierte Skript *ezusb_ldr* erfühlt diese Aufgabe (hier stark gekürzt, nur Prinzip):

```
case "$ACTION" in
    add)
      case $PRODUCT in        # pre-renumeration device IDs
            547/80/*)         #  EZ-USB development board IDs
                  FIRMWARE=/usr/share/usb/uci.ihx
                  FLAGS="-q -l"
            ;;
            547/2131/*)       # EZ-USB default device IDs
                  FIRMWARE=/usr/share/usb/avrpipe.hex
                  FLAGS="-q -l"
            ;;
            esac
            /usr/local/bin/an2131ctl  -D$DEVICE  $FLAGS  $FIRMWARE
      ;;
esac
```

Anhand der IDs in PRODUCT wird die Firmware ausgewählt. Der Pfad zu der Gerätedatei wird dem Firmware-Downloader mit -D$DEVICE bekannt gegeben.

Zusätzlich sind noch zwei Einträge in der */etc/hotplug/usb.usermap* Datei notwendig, damit *usb.agent* das benutzerdefinierte Skript *ezusb_ldr* starten kann:

ezusb_ldr 0x0003 0x0547 0x2131 0x0000 0x0000 0x00 0x00 0x00 0x00 0x00 0x00 0x00000000
ezusb_ldr 0x0003 0x0547 0x0080 0x0000 0x0000 0x00 0x00 0x00 0x00 0x00 0x00 0x00000000

Die geladene Firmware kann nun ein Abziehen (Disconnect) und anschließendes Anstecken simulieren und mit eigenen Vendor- und Product-IDs erneut enumerieren. Das Hotplug-System sorgt dann für das Laden des entsprechenden Gerätetreibers. Dieser Schritt ist aber nicht unbedingt notwendig. Ein User-Mode Treiber kann das Gerät auch gleich über dieselbe Datei wie in $DEVICE ansprechen.

4 USB-AVR Interface AVRPIPE und API

4.1 Aufbau

Die Abbildung unten enthält wesentliche Details zur Anbindung des ATmega8L Microcontrollers an AN2131:

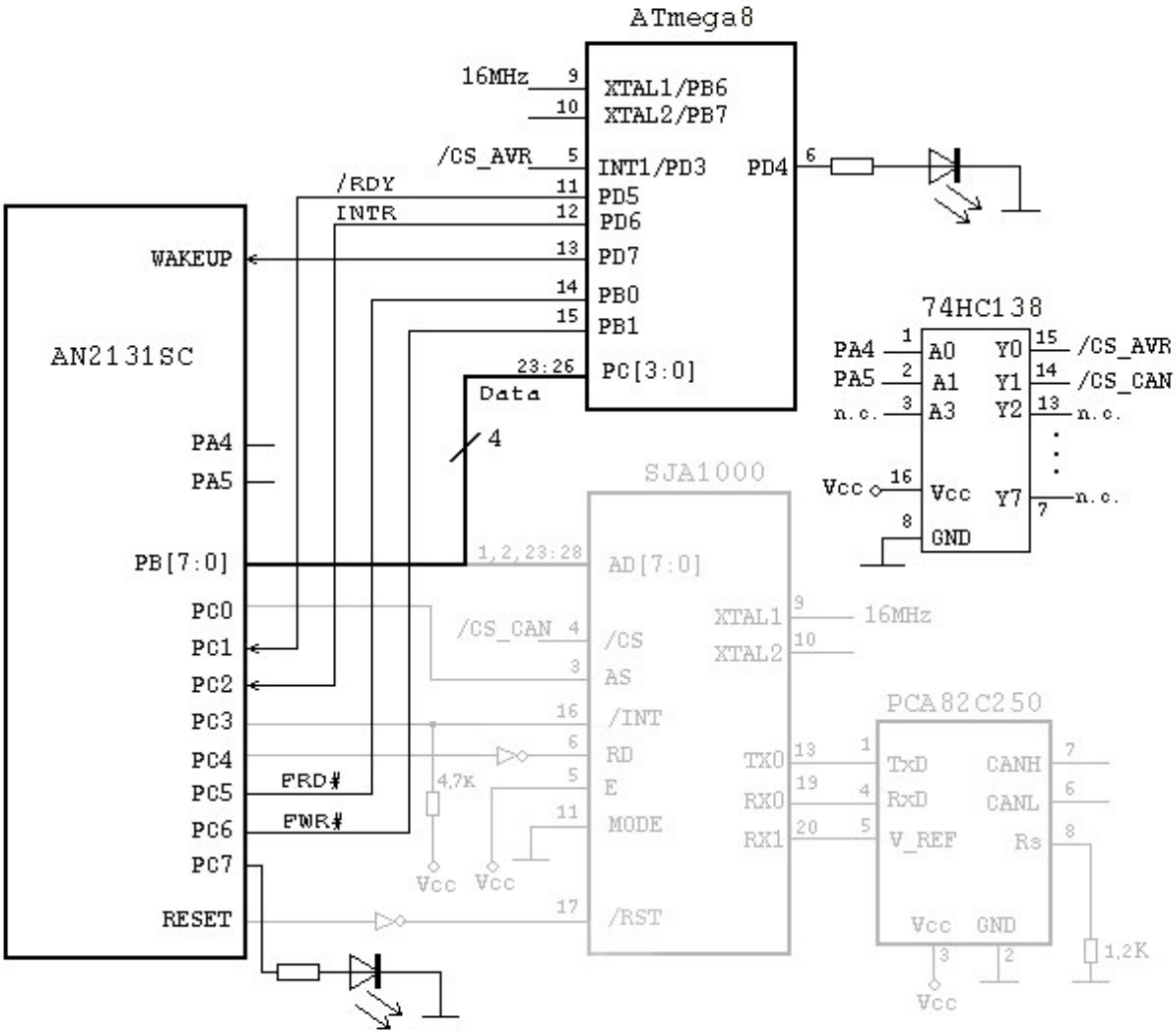

Abbildung 9: Aufbau der Prototypen-Platine

Zur Übertragung der Daten wird hier ein 4-Bit breiter Datenbus verwendet. Folgende Tabelle fasst die Pin-Anbindung zusammen:

AN2131 Pin	AVR Pin	Bedeutung
PA4, PA5 (AVR_CS)	PD3 (INT1)	AVR Chip Select
PB0 - PB3	PC0 - PC3	Data bus
PC1 (/RDY)	PD5	AVR not ready
PC2 (INTR)	PD6	
PC5	PB0	FRD#
PC6	PB1	FWR#
WAKEUP	PD7	

Tabelle 4.1: AN2131-AVR Pin-Mapping

Außerdem sind folgende Vorgaben definiert: mit dem aktiven /RDY (low) signalisiert AVR die Bereitschaft zur Kommunikation. Falls /RDY deaktiviert (high) ist, kann keine

Übertragung stattfinden. Der Initiator von Datentransfers ist immer AN2131. Er startet die Kommunikation indem er AVR_CS aktiviert, was zu einem Interrupt beim AVR führt. AVR kann anhand der FRD# und FWR# Signale an PB0 und PB1 feststellen, welche Transferart erwünscht ist und startet entsprechende Operation:

FRD#	FWR#	Operation
1	1	beide inaktiv, keine Operation
0	1	2131 liest AVR-Puffer
1	0	2131 schreibt AVR-Puffer
0	0	Kommunikation beendet

Die Vorgaben für Hostsoftware lauten:

AN2131 zu AVR

Für Datentransfer vom Host zum AVR wird folgende C-Funktion aufgerufen:

*int send_to_avr(uchar *buffer, int len)*

Der Parameter *buffer* zeigt auf einen Puffer mit Daten, welche zum AVR übertragen werden sollen. Mit dem Parameter *len* wird die Anzahl der Datenbytes in dem Puffer angegeben. Die Anzahl Datenbytes darf die Kapazität des korrespondierenden AVR-Puffers (64 oder 128 Byte) nicht überschreiten. Die Daten werden sofort zum AVR übertragen, indem ein Interrupt beim AVR ausgelöst wird. Nach dem Transfer setzt AVR ein Flag zur Signalisierung von gültigen Daten in dem AVR-RX-Puffer. *send_to_avr* soll nach einem Puffertransfer gleich zur Übertragung des nächsten Puffers eingesetzt werden können, selbst wenn die gültigen Daten im AVR-Puffer überschrieben werden würden.

Die Rückgabewerte sollten mögliche Fehler reflektieren:
-1: *len* ist zu groß
-2: kein Zugriff auf AVR möglich
-3: AVR /RDY nicht aktiv
Bei einem erfolgreichen Transfer wird die Anzahl der übertragenen Bytes (= l*en*) zurückgegeben.

AVR zu AN2131

Zum Lesen des AVR-TX-Puffers wird folgende C-Funktion aufgerufen:

*int read_from_avr(uchar *buffer)*

Der Parameter *buffer* zeigt auf ein Puffer, welcher groß genug ist um Inhalt des AVR-TX-Puffers aufzunehmen (z.B. 64 oder 128 Byte).

Falls AVR-TX-Puffer leer ist, wird 0 zurückgegeben. Nach erfolgreichem Lesen wird die Anzahl der gelesenen Bytes als Rückgabewert geliefert. Negative Werte signalisieren aufgetretene Fehler:
-1 : irgendein Fehler passiert
-3 : AVR /RDY nicht aktiv
Die Funktion soll nie blockieren.

Um die Anzahl der Datenbytes im AVR-TX-Puffer zu bestimmen wird folgende Funktion aufgerufen:

int avr_data_available(void)

Diese Funktion liest keine Pufferdaten (außer Statusbytes) aus. Der Rückgabewert reflektiert die Anzahl Bytes in dem Puffer oder aufgetretenen Fehler (ähnlich wie *read_from_avr()*).
Nach erfolgreichem Auslesen der Daten wird der Puffer im AVR als leer markiert, sodass hintereinander folgende Leseoperationen 0 zurück liefern.

4.2 Framework auf AVR-Seite

In dem AVR-RAM werden zwei Puffer angelegt. Bei dem ersten handelt es sich um Empfangspuffer (AVR-RX-Puffer), in dem die vom Host gesendeten Daten abgelegt werden. Der zweite Puffer (AVR-TX-Puffer) enthält die Daten, welche an den Host weitergeleitet werden. Die ersten zwei Bytes des AVR-TX-Puffers enthalten Statusinformationen (Anzahl der gültigen Bytes) und werden nach jedem Auslesen auf Null gesetzt. Alle Puffertransfers und die Übertragung von Statusinformationen (auf Anfrage von *avr_data_available()*) werden vom INT1-Interrupthandler synchron durchgeführt. Zusätzlich wurde in dem Interrupthandler eine Möglichkeit vorgesehen, kompletten AVR-RAM an AN2131 zu transferieren. Die Art der an den Interrupthandler delegierten Aufgabe wird mit verschiedenen Codes auf PortB- und PortC-Pins spezifiziert. Insgesamt stehen zwei Pins von PortB (PB0, PB1) und vier Pins von PortC (PC0, PC1, PC2, PC3) zur Verfügung. Die Codes an PB0, PB1 und PC0 - PC3 wurden wie folgt vergeben:

PB1	PB0	Bedeutung
0	0	keine Operation, Interrupthandler kehrt zurück
0	1	Handler verzweigt zur Empfangsroutine (*int1_write_rxbuf*)
1	0	Handler verzweigt zur Senderoutine (*int1_send_txbuf*)
1	1	Operation ist in PortC[0:3] codiert

PC3	PC2	PC1	PC0	Bedeutung
0	0	0	0	Handler verzweigt zur *int1_data_available* Routine
0	0	0	1	Handler verzweigt zur *int1_read_sram* Routine

Weiter sind noch 14 zusätzliche Verzweigungen realisierbar (2^4 - 2 obere PortC Varianten).

Die Verzweigungen wurden mit Sprungtabellen realisiert. Dies ist vorteilhaft, da hier die gleiche Zeit für jede Verzweigung auf einer Ebene notwendig ist. Dies erleichtert erheblich die Synchronisation zwischen Code in AVR und AN2131.
Der INT1-Interrupthandler sichert zunächst die verwendeten Register auf dem Stack und liest dann den Code vom PortB. Der Code wird zur Adresse der Sprungtabelle erster Ebene addiert und danach wird an diese neu berechnete Adresse gesprungen. Falls PB0 und PB1 gesetzt sind, wird die Routine angesprungen, welche entsprechend dem Code an PortC[0:3] zu einer Routine der zweiten Ebene (*int1_data_available*, *int1_read_sram*) verzweigt.

Alle Routinen, welche derart aus dem INT1-Handler angesprungen werden, erledigen ihre Aufgaben und springen zurück zum INT1-Handler, welcher nach dem Restaurieren der Register zurückkehrt.

Die Puffergröße wird beim Übersetzen der Programme festgelegt. Es sind Puffergrößen von 64, 128 und 256 Byte möglich.

4.3 Implementierung auf AN2131-Seite (AVRPIPE)

In dem internen Programm-/Datenspeicher von AN2131 werden zwei Transferpuffer angelegt. Der erste Transferpuffer (AN2131-RX) enthält das Abbild des AVR-TX-Puffers und wird beim Auslesen des AVR-TX-Puffers gefüllt. Der zweite (AN2131-TX) dient zum Zwischenspeichern der Daten, welche vom Host an AVR geschickt werden sollen. In einem weiteren Puffer wird der gesamte AVR-SRAM abgebildet (s. Abbildung 10).

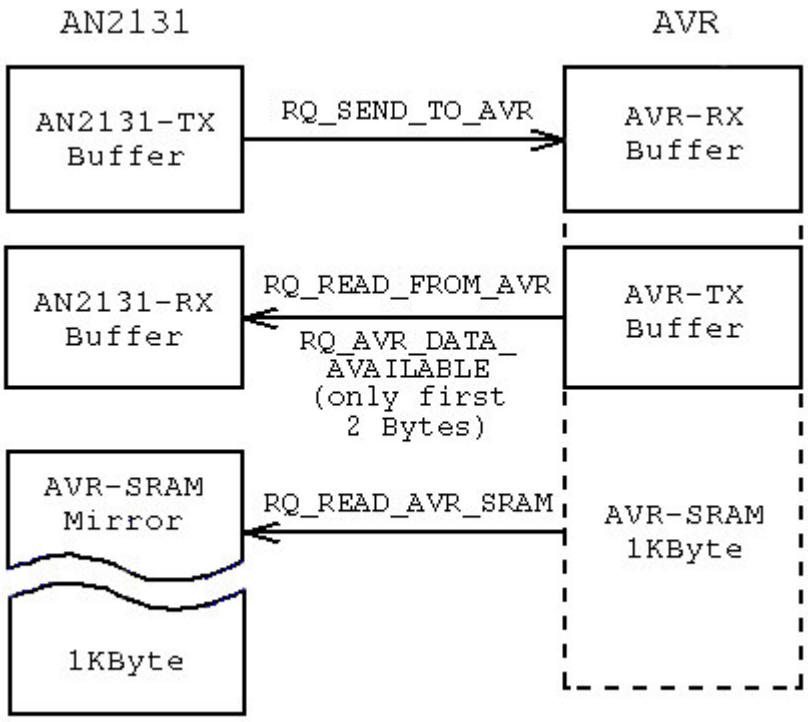

Abbildung 10: AN2131-AVR Transfers

Die Datenübertragung vom Host zum AN2131-TX-Puffer und vom AN2131-RX-Puffer zum Host geschieht ohne Beteiligung des Programms im AN2131. AN2131-Core erfüllt diese Aufgaben auf Anforderung des Hosts (FIRMWARE-DOWNLOAD/UPLOAD Request).

Alle Transferoperationen vom bzw. zum AVR werden vom Host mit einem USB-Device-Request eingeleitet. Dabei empfängt AN2131 8 Byte langes SETUP-Data-Paket, welches den Request beschreibt. Beim Empfangen eines solchen Requests wird beim AN2131 ein sog. Setup-Data-Available Interrupt (SUDAV) ausgelöst. Die entsprechende Behandlungsroutine liest das *bRequest* Byte des SETUP-Data-Pakets. Falls im *bRequest* angegebene Request unterstützt wird, verzweigt die Behandlungsroutine zu dem passenden Handler. Das Programm im AN2131 unterstützt folgende Requests: RQ_AVR_DATA_AVAILABLE, RQ_SEND_TO_AVR, RQ_READ_FROM_AVR, RQ_READ_AVR_SRAM. Die Codierung und weitere Parameter dieser Requests können dem Anhang D entnommen werden.

4.4 Libavrp als API zur Anwendungsentwicklung

Auf der Host-Seite wurde eine kleine Bibliothek *libavrp* mit 7 Funktionen zur Verfügung gestellt: *usb_avr_pipe_init(), open_avr(), close_avr(), sent_to_avr(), read_from_avr(), avr_data_available(), read_avr_sram().* Eine Beschreibung der einzelnen Prototypen (Argumente, Rückgabewerte) befindet sich im Anhang A. Alle Funktionen verwenden *libusb*-Routinen zum Zugriff auf ein USB-AVR-Gerät. Außer *usb_avr_pipe_init()* sind alle anderen Funktionen zur Anwendungsentwicklung bestimmt. *usb_avr_pipe_init()* wird automatisch nach dem Laden der Bibliothek bei jedem Programmstart aufgerufen. Eine einfache Applikation (einmaliges Auslesen des AVR-TX Buffers) könnte wie folgt aussehen:

```
/* simple.c */
#include "libavrp.h"

int main(void)
{
        int i, ret, devh;
        char buf[BUF_SIZE];

        devh = open_avr(0); // open device 0 (first connected device)
        if (devh < 0)
                exit(1);
        ret = read_from_avr(devh, buf);
        if (ret < 0)
                exit(1);
        if (ret)
           for(i = 0; i < ret; i++)
                printf("%#x ", buf[i]);
        else
           printf("no avr data available\n");
        close_avr(devh);
        return 0;
}
```

Listing 1: Anwendung von AVRPIPE API-Funktionen

Übersetzt wird dieses Programm wie folgt:

gcc simple.c -I <Pfad zum Verzeichnis mit libavrp.h> -L/usr/local/lib -lavrp -lusb

5 USB-CAN Interface UCI

5.1 Überlegungen zum Aufbau und Festlegungen

Aufgaben

Die Aufgaben des USB-CAN Schnittstellenprogramms für AN2131-Kern sind:

- Konfiguration und Steuerung des CAN-Controllers
- Übertragen von CAN-Nachrichten, welche über den USB ankommen, zum CAN-Controller und Starten des CAN-Transfers; auch Transferstatus mitteilen
- Zwischenspeichern der vom CAN-Controller empfangenen Nachrichten und deren Weiterleitung über den USB
- Melden von aufgetretenen CAN-Bus-Fehlern
- Beantworten von Anfragen (USB-Device-Requests) des Hosts zur USB-CAN Geräteidentifikation und Steuerung

Endpoint-Layout

Die Einrichtung der sog. Default-Pipe (Control-Pipe) zur Abwicklung von USB-Device-Requests und zur Gerätesteuerung ist bei jedem USB-Gerät erforderlich. Eine Control-Pipe ist immer mit dem Endpoint 0 assoziiert. Weiterhin wird eine Interrupt-Pipe zur Übermittlung von Gerätestatus-Informationen und Fehlern (und somit ein IN-Interrupt-Endpoint) benötigt. Eine FIFO-Größe von 16 Byte sollte für IN-Interrupt-Endpoint ausreichend sein.

Für die Kommunikation von CAN-Nachrichten vom bzw. zum USB-CAN Gerät eignen sich Bulk-Transfers am besten. Diese sind schnell und unterliegen der integrierten Fehlerbehandlung des USB-Protokolls. Somit ist noch die Einrichtung jeweils einer Bulk-Pipe pro Transferrichtung erforderlich. Die maximale FIFO-Größe eines Bulk-Endpoints beträgt 64 Byte. Alle oben genannten Endpoints werden im Interface 0 zusammengefasst (s. Abb. 11).

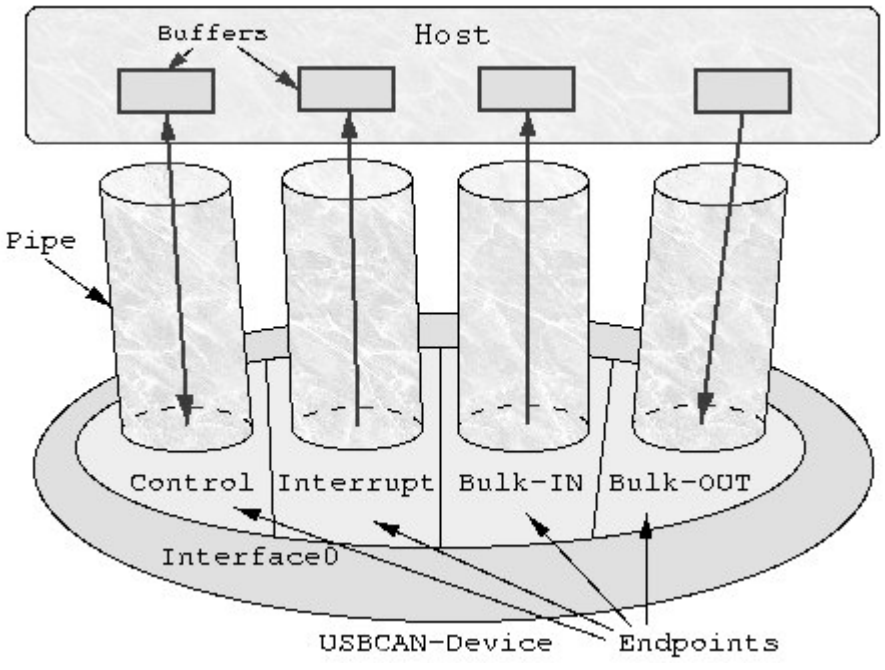

Abbildung 11: USB-CAN-Endpoints und Pipes

AN2131 besitzt 16 Endpoints für Control-, Bulk- und Interrupt-Transfers mit jeweils 64 Byte großem Endpoint-FIFO. Für ein USB-CAN-Gerät werden nur die vier oben genannten Endpoints benötigt, alle anderen werden deshalb deaktiviert. Folgende Tabelle erläutert das Endpoint-Layout von USB-CAN:

Endpoint	Endpoint-Adresse	Transferart	Richtung	Art der übertragenen Daten	FIFO-Tiefe, Byte
EP0	0x00	Control	bidirektional	Control, Kommandos	64
EP1	0x02	Bulk	OUT	gesendete CAN-Nachrichten	64
EP1	0x82	Bulk	IN	empfangene CAN-Nachrichten	64
EP2	0x84	Interrupt	IN	Status, Fehler	16

Tabelle 5.1: Endpoint-Layout des USB-CAN-Geräts

Internes Nachrichtenformat und Layout von Bulk-Endpoint-FIFOs

Der Transmit-Buffer des CAN-Controllers SJA1000 besitzt folgendes Layout:

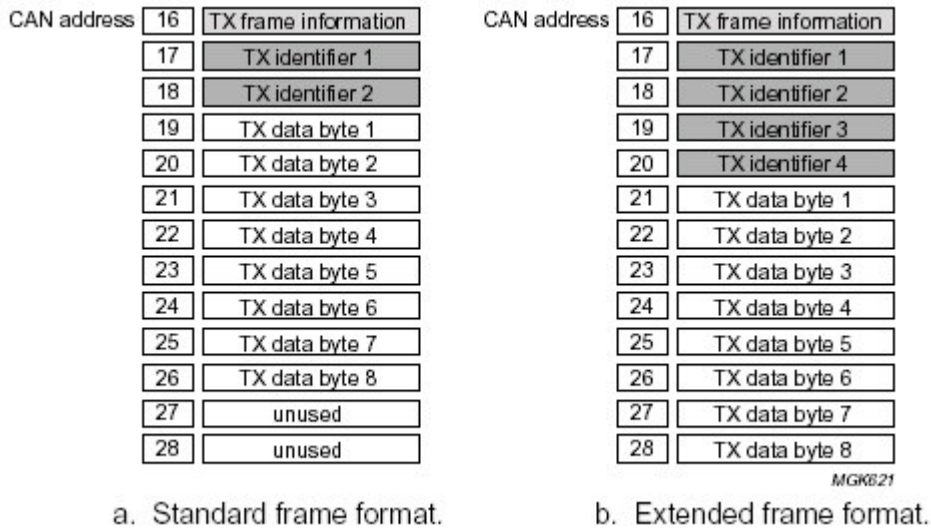

Abbildung 12: Layout des Transmit-Puffers von SJA1000 (Bildquelle [9])

Im ersten Byte wird die Information zum Frame-Format (Standard/Extended, Remote) und die Länge des Datenfelds codiert. Die nachfolgenden 2 (Standard Frame) bzw. 4 (Extended Frame) Byte enthalten den Identifier. Anschließend kommen die eigentlichen Daten einer Nachricht. Die Größe des Transmit-Buffers beträgt somit 13 Byte. Der Receive-Buffer des CAN-Controllers besitzt das gleiche Layout. Das Programm im AN2131 soll die Nachrichten in dem vom CAN-Controller vorgegebenen Format (Layout der beiden Puffer) erhalten und diese einfach in den Transmit-Buffer des Controllers übertragen. Die Formatierung übernimmt der Treiber. Beim Empfangen werden alle relevanten Bytes aus dem Puffer gelesen und unverändert an den Host weitergeleitet. Die Umwandlung in die von einer Anwendung erwartete Struktur wird ebenfalls vom Treiber durchgeführt.

Das Layout von Bulk-Endpoint-FIFOs wurde wie folgt festgelegt:

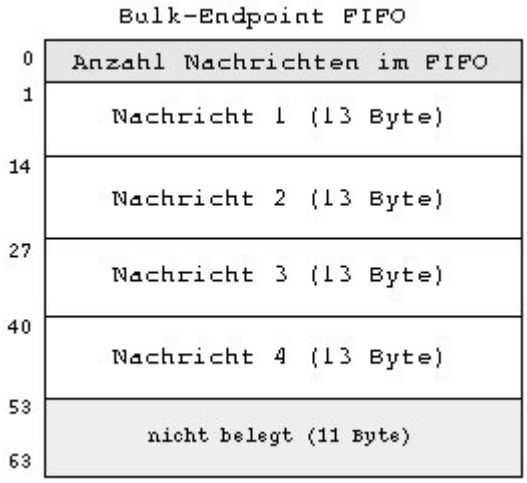

Abbildung 13: Bulk-Endpoint FIFO Layout

Somit werden bei 4 Nachrichten insgesamt 53 Byte zum bzw. vom Endpoint übertragen. Bei Nachrichtengrößen kleiner als 13 Byte (Standard Frame Nachrichten, Nachrichten mit kürzerem Datenfeld) könnte man mehr Nachrichten in einem FIFO unterbringen. Um das Programm im AN2131 einfacher zu gestalten, wurde darauf verzichtet. Eine zukünftige Erweiterung sollte aber kompaktere Variante implementieren.

Interrupt-Endpoint

Die über IN-Interrupt-Endpoint (IN-Interrupt-FIFO) gesendeten Daten reflektieren Status einer CAN-Sendeoperation (Anzahl der erfolgreich übertragenen Nachrichten) oder verschiedene CAN-Controller Fehler. Die Tabelle 5.2 beschreibt das Layout der gesendeten Information.

Offset	Name	Bedeutung
0	*st_code*	Status-Code; Bitmap, Bedeutung unten
1	*errors*	Anzahl aufgetretener Fehler
2	*overruns*	Anzahl RX-FIFO Overruns
3	*irq*	letztes IRQ des CAN-Controllers
4	*status*	Status Register des CAN-Controllers
5	*err_code_cap*	Error Code Capture Register des CAN uC
6	*err_warn_lim*	Error Warning Limit Register
7	*rx_err_cnt*	Receive error Counter Register
8	*tx_err_cnt*	Transmit Error Counter Register
9	*fw_fifo_ovr*	Anzahl der Firmware-RXFIFO Overruns
10	*mode*	Mode Register des CAN uC

Tabelle 5.2: Interrupt-IN Endpoint Daten

Das Element *st_code* enthält Information über ein aufgetretenes Ereignis und wird vom Treiber ausgewertet. Die Bedeutung einzelner Bits der Bitmap *st_code* ist in der Tabelle 5.3 beschrieben.

Bitposition	Wert	Bedeutung
7 bis 4	0 bis 4	Anzahl der erfolgreich gesendeten Nachrichten
3	1	Transmit-Fehler aufgetreten
	0	kein Transmit-Fehler
2	1	Firmware RX-Buffer Overrun
	0	kein Overrun
1	1	CAN Error Interrupt, nachfolgende 10 Byte enthalten gültige Information
	0	kein Error Interrupt
0	1	CAN RX-FIFO Overrun, nachfolgende 4 Byte enthalten gültige Info
	0	kein CAN-Controller RX-FIFO Overrun

Tabelle 5.3: Codierung der Statusbitmap *st_code*

5.2 Definition von Deskriptoren

Nachdem Endpoint-Layout festgelegt wurde, können die einzelnen Deskriptoren definiert werden. Auf oberster Ebene ist der Device-Deskriptor angesiedelt. Die Tabelle unten erläutert die Belegung der zugehörigen Felder.

Feldbezeichnung	Belegung	Beschreibung
bLength	0x12	Deskriptor-Größe in Byte
bDescriptorType	0x01	Device-Descriptor
bcdUSB	0x10, 0x01	USB-Spezifikation, BCD-Format
bDeviceClass	0xff	Herstellerspezifische Klasse
bDeviceSubClass	0x00	Subklasse
bDeviceProtocoll	0xff	Herstellerspezifisches Protokoll
bMaxPacketSize0	0x40	FIFO-Tiefe von Endpoint 0
idVendor (low)	UCI_VENDOR_ID, low	Hersteller-ID
idVendor (high)	UCI_VENDOR_ID, high	
idProduct (low)	UCI_PRODUCT_ID, low	Produkt-ID
idProduct (high)	UCI_PRODUCT_ID, high	
bcdDevice (low)	UCI_DEV_RELEASE, low	Seriennummer
bcdDevice (high)	UCI_DEV_RELEASE, high	
iManufacturer	0x01	String-Index für Hersteller
iProduct	0x02	String-Index für Produkt
iSerialNumber	0x03	String-Index für Seriennummer
bNumConfigurations	0x01	Anzahl unterstützen Konfigurationen

Tabelle 5.4: Device-Descriptor

Das USB-CAN-Gerät besitzt nur eine Konfiguration und somit einen Configuration-Deskriptor, dem ein Interface-Deskriptor untergeordnet ist. Die nachfolgenden Tabellen beschreiben jeweils die Deskriptoren.

Feldbezeichnung	Belegung	Beschreibung
bLength	0x09	Länge dieses Deskriptors in Byte
bDescriptorType	0x02	Configuration-Descriptor
wTotalLength (low)	0x30	Länge aller zu dieser Konfiguration gehörenden Deskriptoren (Configuration, Interface, Endpoints)
wTotalLength (high)	0x00	
bNumInterfaces	0x01	Anzahl Interfaces in dieser Konfiguration
bConfigurationValue	0x01	Nummer dieser Konfiguration (Argument für setConfiguration)
iConfiguration	0x04	String-Index für diese Konfiguration
bmAttributes	0x80	Attribute der Konfiguration (Bus-Powered, kein Remote-Wakeup)
MaxPower	0x32	Stromaufnahme in dieser Konfiguration (in 2mA-Einheiten)

Tabelle 5.5: Configuration-Descriptor

Feldbezeichnung	Belegung	Beschreibung
bLength	0x09	Größe dieses Deskriptors in Byte
bDescriptorType	0x04	Interface-Descriptor
bInterfaceNum	0x00	Interface Nummer
bAlternateSetting	0x01	AlternateSetting für dieses Interface
bNumEndpoints	0x03	Anzahl der zu diesem Interface gehörenden Endpoints (ohne EP0)
bInterfaceClass	0xff	Klassen-Code (vendor-spezifisch)
bInterfaceSubClass	0x00	Subklassen-Code
bInterfaceProtocoll	0xff	Protokoll-Code (vendor-spezifisch)
iInterface	0x00	String-Index für dieses Interface

Tabelle 5.6: Interface-Descriptor

Interface 0 umfasst 3 Endpoints (ohne EP0). Endpoint 0 ist bereits im Device-Deskriptor beschrieben. Somit werden noch drei Endpoint-Deskriptoren benötigt:

Feldbezeichnung	Belegung	Beschreibung
bLength	0x07	Größe dieses Deskriptors in Byte
bDescriptorType	0x05	Endpoint-Descriptor
bEndpointAddress	0x02	Endpoint-Adresse (OUT2 Endpoint)
bmAttributes	0x02	Transferart = Bulk
wMaxPacketSize (low)	0x40	FIFO-Größe des Endpoints in Byte (hier 64)
wMaxPacketSize (high)	0x00	
bInterval	0x00	Polling-Intervall (hier 0)

Tabelle 5.7: Bulk-OUT Endpoint-Descriptor

Feldbezeichnung	Belegung	Beschreibung
bLength	0x07	Größe dieses Deskriptors in Byte
bDescriptorType	0x05	Endpoint-Descriptor
bEndpointAddress	0x82	Endpoint-Adresse (IN2 Endpoint)
bmAttributes	0x02	Transferart = Bulk
wMaxPacketSize (low)	0x40	FIFO-Größe des Endpoints in Byte (hier 64)
wMaxPacketSize (high)	0x00	
bInterval	0x00	Polling-Intervall (hier 0)

Tabelle 5.8: Bulk-IN Endpoint-Descriptor

Feldbezeichnung	Belegung	Beschreibung
bLength	0x07	Größe dieses Deskriptors in Byte
bDescriptorType	0x05	Endpoint-Descriptor
bEndpointAddress	0x84	Endpoint-Adresse (IN4 Endpoint)
bmAttributes	0x03	Transferart = Interrupt
wMaxPacketSize (low)	0x10	FIFO-Größe des Endpoints in Byte (hier 16)
wMaxPacketSize (high)	0x00	
bInterval	0x01	Polling-Intervall (hier 1 ms)

Tabelle 5.9: Interrupt-IN Endpoint-Descriptor

Da im Device-Deskriptor die Indizes für Hersteller, Produkt und Revision eingetragen wurden, müssen zusätzlich die entsprechenden String-Deskriptoren definiert werden. Diese werden hier nicht abgebildet (bei Interesse s. Definitionen in usbcan/uci/uci.h).

5.3 Behandlung von USB-Device-Requests

Eine der im Abschnitt 5.1 genannten Aufgaben des Programms im AN2131 ist die Bearbeitung von USB-Device-Requests. Darunter fallen alle Standard-Device-Requests, welche von jedem USB-Gerät unterstützt werden müssen, und herstellerspezifische USB-Device-Requests.

Dieses etwas umfangreichere Teil des Programms wurde in C implementiert (s. usbcan/uci/uci.c). Für jeden nicht optionalen Standard-Request (außer SET_ADDRESS) wurde eine Behandlungsroutine erstellt, welche die von USB-Spezifikation geforderten Aktionen durchführt. Die Behandlung von SET_ADDRESS wird vom AN2131-Kern automatisch abgewickelt. Zusätzlich wurde ein herstellerspezifischer Request UCI_CAN_CHIP_ACCESS (s. Anhang C) und zugehörige Bearbeitungsfunktion definiert um bestimmte Steuerungsaufgaben (CAN-Controller starten bzw. anhalten, Baudrate oder Filter konfigurieren, etc.) zu lösen. Angesprungen werden die Funktionen aus dem USB-Interrupthandler. Die Steuerung des CAN-Controllers erfolgt über Zugriffe auf seine Control-Register, worauf im nächsten Abschnitt näher eingegangen wird.

5.4 Zugriff auf die Register des CAN-Controllers

Die Abbildung 13 unten zeigt die Anbindung des SJA1000 CAN-Controllers an AN2131.

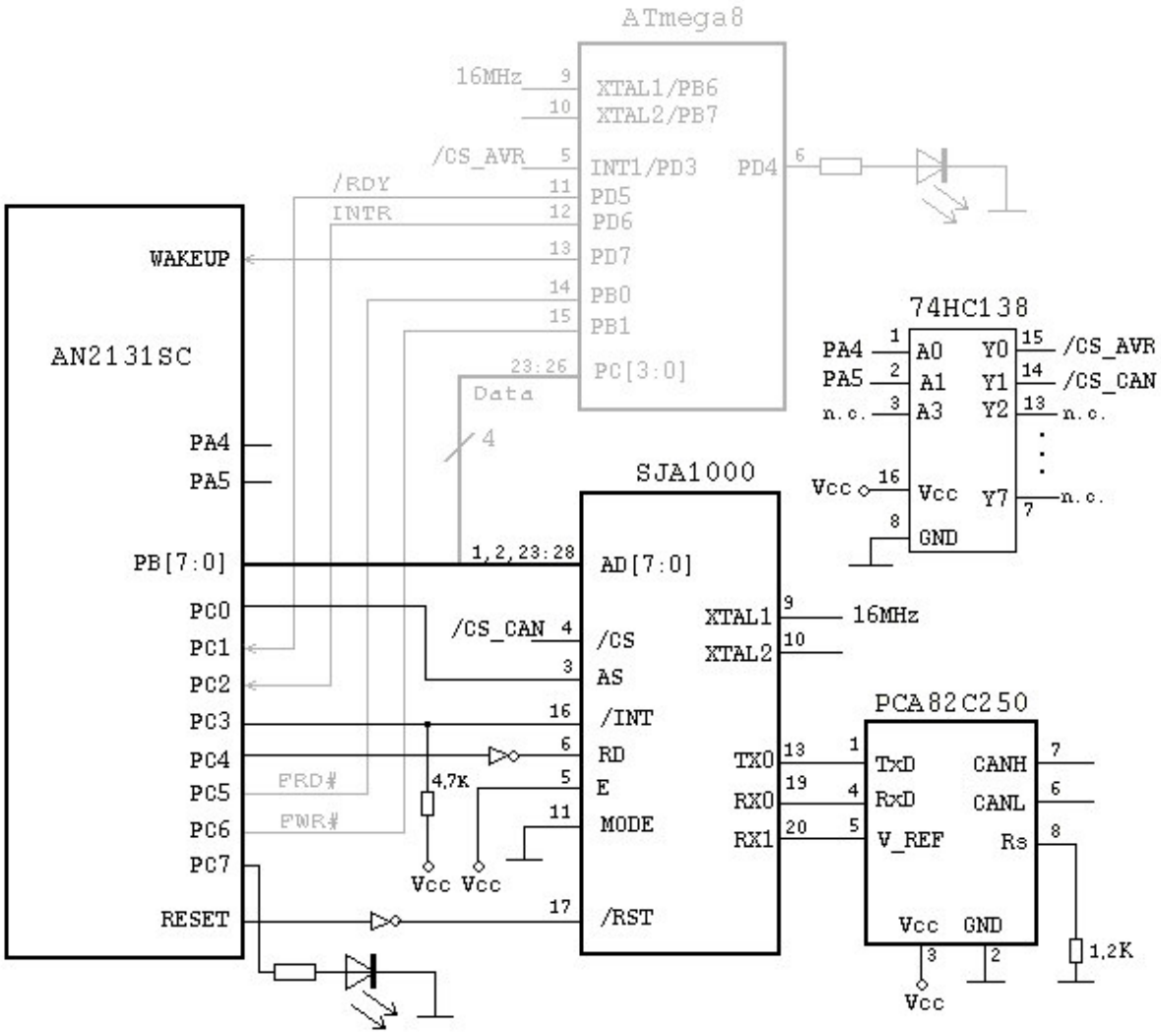

Abbildung 14: AN2131-SJA1000 Schnittstelle

Der Zugriff auf die Control-Register und Puffer des CAN-Controllers erfolgt über gemultiplexten Adress-/Datenbus (PB[7:0]). Da der MODE-Pin (11) des SJA1000 auf GND liegt, wird der Controller im sog. Motorola-Modus angesprochen. Die zugehörigen Signalpegel bei einem Lese- und einem Schreib-Zyklus sind den Abbildungen 14 und 15 zu entnehmen. Die einzelnen Signale AS, RD/#WR und /CS müssen vom Programm erzeugt werden.

Für den Registerzugriff wurden die Routinen *can_rd* und *can_wr* definiert. Anders als bei Standard-8051 liegen die Ports beim AN2131 im externen RAM und müssen entsprechend adressiert werden (*movx @Ri,a* oder *movx @dptr,a* Instruktion des AN2131). Zuerst wird ein Pointer-Register mit Adresse des Ports geladen und erst dann auf den Port registerindirekt zugegriffen. Zusätzlich sind die Ports nicht bidirektional, d.h., dass für das Lesen und Schreiben eines Ports verschiedene Adressen verwendet werden, was ein Umladen der Pointer-Register nach sich zieht. Außerdem muss die Datenrichtung über die OE-Register (Output Enable) festgelegt werden. Deshalb fallen die Routinen etwas "aufgebläht" aus (s. usbcan/uci/can-proto.asm).

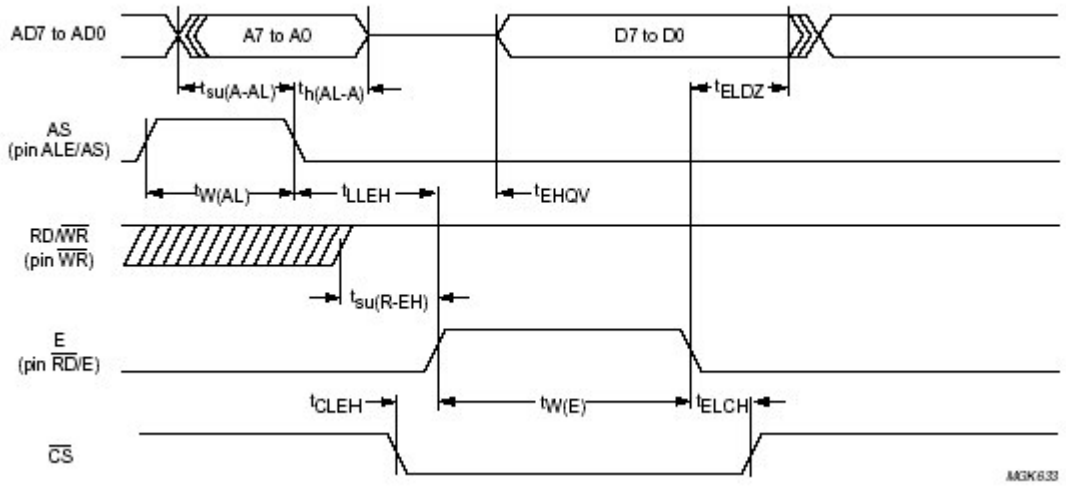

Abbildung 15: Zeitliche Lage der Signale bei einem Lesezyklus (Bildquelle [9])

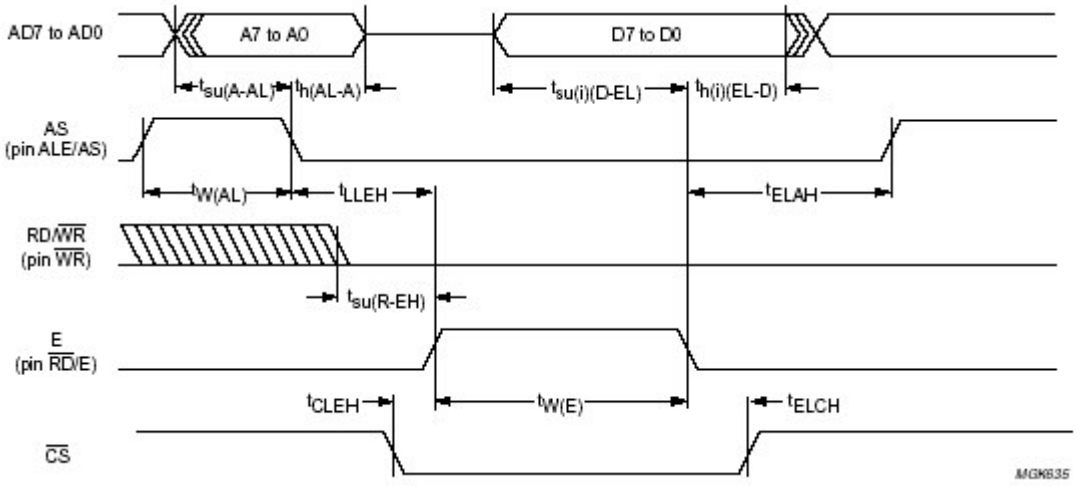

Abbildung 16: Zeitliche Lage der Signale bei einem Schreibzyklus (Bildquelle [9])

Die Routine *can_rd* wird hier näher betrachtet (C-Prototyp *unsigned char can_rd(unsigned char addr) using 1*). Als Argument erhält sie die Adresse des zu lesenden CAN-Controller-Registers. Nach der Aufrufe-Konvention des *sdcc*-Compilers wird der erste Argument in dem Low-Byte *dpl* des *dptr*-Registers übergeben. Nach dem Laden der Pointer-Register *r0, r1* mit Portadressen (Low Byte!) wird CAN-Registeradresse auf den Adress-/Datenbus (Port B) gelegt:

```
mov    a,dpl          ; get address argument
mov    r0,#0x98       ; low(OUTC), high is in MPAGE
mov    r1,#0x97       ; low(OUTB)
movx   @r1,a          ; put address to bus
```

Nun muss das Address Strobe (AS) Signal für eine kleine Zeit (min 8 ns) aktiviert werden. Parallel erfolgt die Aktivierung von RD (wobei RD negierte PC4-Wert ist):

```
mov    a,#0x01        ; address strobe (AS in bit 0, RD in bit 4)
movx   @r0,a          ; write AS and RD to Port C
clr    a
movx   @r0,a          ; disable AS, keeping RD active
```

Nach dem Schalten des Ports B in Leserichtung wird Chip-Select Signal aktiviert. CAN-Controller legt daraufhin den Registerwert auf den Bus:

```
mov    r1,#0x9D      ; low(OEB)
movx   @r1,a         ; zero OEB (turns off output buffers)
mov    r1,#0x96      ; low(OUTA)
mov    a,#0x10       ; /CS_CAN
movx   @r1,a         ; select CAN controller
```

Der Registerwert wird vom Bus gelesen und in *dpl* abgelegt (da nach Konvention des *sdcc* der Rückgabewert (*char*) einer Funktion über *dpl* an Aufrufer übergeben wird):

```
mov    r0,#0x9A      ; low(PINSB), high is in MPAGE
movx   a,@r0         ; read register from bus (PINSB)
mov    dpl,a         ; return it in dpl
```

Schließlich erfolgt die Deaktivierung des Chip-Select Signals und Umschaltung des Ports B auf Output:

```
clr    a
movx   @r1,a         ; disable /CS_CAN
mov    r1,#0x9D      ; low(OEB)
cpl    a             ; a = 0xff (1 cycle instead of 2 cycles mov a,#0xff)
movx   @r1,a         ; turn on output buffers
ret
```

Ähnlich ist die Routine *can_wr* aufgebaut. Diese ist etwas kurzer, da hier die Umschaltung des Ports B auf Lesen nicht notwendig ist.

5.5 CAN-Controller Initialisierung

Nach dem Laden des Programms in den AN2131 erfolgt die Initialisierung des CAN-Controllers. Dabei werden folgende Schritte unternommen:

- Umschalten in den PeliCAN Modus
- Aktivieren der Transmit-, Receive-, Overrun- und Error-Interrupts
- Konfiguration des SJA1000 für Zusammenarbeit mit dem Transceiver 82C250 (Output Control Register mit 0xDA belegen)
- Konfiguration auf 125 KBit/s (Bus Timing Register 0 und 1)
- Deaktivieren der Filter, Empfang aller Nachrichten ermöglichen (Acceptance-Code, Acceptance Mask)

Nach der Initialisierung bleibt der CAN-Controller zunächst im Reset-Modus. Das Umschalten in den Operation-Modus und Aktivieren des CAN-Controller Interrupts INT1 erfolgt erst später auf Anforderung einer Host-Anwendung (Starten des Controllers durch den UCI_CAN_CHIP_ACCESS-Request mit dem Parameter UCI_CAN_START).

5.6 CAN-Interrupt Behandlung

Mit einem Low-Pegel auf /INT (low active) signalisiert der CAN-Controller folgende Ereignisse: Empfangen einer Nachricht, erfolgreiches Absenden einer Nachricht, Überlauf des Empfangs-Puffers, Fehler. Die genaue Interrupt-Quelle erfährt der behandelnde

Microcontroller, indem er den Interrupt-Request-Register des CAN-Controllers ausliest und die einzelnen Bits auswertet. Um bei einem Receive-Interrupt den Empfangspuffer möglichst schnell für neu ankommende Nachrichten freizugeben, erfolgt die Interruptbehandlung in einer Schleife. Als Abbruchkriterium dient der Status des CAN-Empfangspuffers. Zusätzliche Abbruchbedingung ist der Überlauf des Zwischenpuffers im AN2131. In der Schleife wird Interrupt-Request-Register bei jedem Durchlauf ausgelesen um eventuell neu aufgetretene Interrupts nicht zu übersehen.

Bei einem Receive-Interrupt wird die empfangene Nachricht ausgelesen und in das Bulk-IN Endpoint-FIFO übertragen. Falls aber Bulk-IN Endpoint-FIFO nicht frei ist oder in dem Zwischenpuffer Nachrichten vorhanden sind, wandert die gerade empfangene Nachricht in den Zwischenpuffer um die Empfangsreihenfolge beizubehalten. Das Leeren des Zwischenpuffers (Übertragung in das Bulk-IN Endpoint-FIFO und Starten des USB-Transfers) wird vom USB-Interrupt-Handler durchgeführt.

Beim Transmit-Interrupt-Request wird geprüft, ob in dem Bulk-OUT Endpoint-FIFO die Nachrichten zum Absenden vorhanden sind. Falls noch Nachrichten in diesem FIFO vorliegen, wird die nächste Nachricht in der Reihenfolge in den Transmit-Buffer des CAN-Controllers übertragen und das Absenden veranlasst. Falls Bulk-OUT FIFO leer ist, wird es freigegeben um weitere Nachrichten vom USB empfangen zu können. Zusätzlich wird im diesem Fall dem Host die Übertragung des FIFO-Inhalts bestätigt (über IN-Interrupt Endpoint).

Im Falle eines Error- oder Overrun-Interrupts erfolgt die Übermittlung der Statusinformation (einige CAN-Controller Register und interne Fehlerzähler) über IN-Interrupt Endpoint-FIFO.

5.7 USB-Interrupt Behandlung

Die Behandlungsroutine für USB-Interrupts übernimmt:

- die Bearbeitung von USB-Device-Requests (Standard- und Vendor-Requests)
- Starten des CAN-Transfers beim Empfang der Nachrichten vom USB
- Übertragen von Nachrichten aus dem Zwischenpuffer zum Bulk-IN Endpoint-FIFO und Freigeben dieses FIFOs für USB-Transfer zum Host

Beim Ankommen eines USB-Device-Requests wird im AN2131 ein Setup-Data-Available Interrupt (SUDAV) ausgelöst. Anhand der Setup-Daten wird die entsprechende Behandlungsroutine bestimmt und dann aufgerufen. Dies gilt sowohl für Standard- als auch für Vendor-Requests.

Falls die für das Bulk-OUT Endpoint bestimmten Daten vom USB ankommen und in das Endpoint-FIFO übernommen werden, kommt es zu einem EP2-OUT Interrupt. Erste Nachricht aus dem OUT-FIFO wird zum CAN-Controller übertragen und das Absenden gestartet. Weitere Nachrichten werden dem FIFO vom CAN-Interrupthandler entnommen (nach erfolgreichem ersten Transfer wird ja ein CAN-Transmit Interrupt ausgelöst).

Zu einem EP2-IN Interrupt kommt es, wenn die Daten vom Bulk-IN Endpoint-FIFO erfolgreich an den Host übermittelt wurden. Dieses FIFO kann nun mit weiteren bis dahin vom CAN angekommenen Nachrichten geladen und für den USB-Transfer freigegeben werden. Diese Arbeit übernimmt der entsprechende Zweig des USB-Interrupthandlers.

Ein Manko bei der Realisierung des USB-Interrupt-Handlers ist die Art, wie Interrupt-Quelle festgestellt wird. Um die Interrupt-Quelle in Erfahrung zu bringen werden

nacheinander die IRQ-Register (USBIRQ, IN07IRQ, OUT07IRQ) gelesen und ausgewertet. Erst dann wird entsprechend verzweigt. AN2131 unterstützt einen sog. Autovectoring-Mechanismus (s. „9.10 USB Autovectors“ in [8]) um direkt zu dem richtigen Handler zu springen ohne IRQ-Register auswerten zu müssen. Eine zukünftige Implementierung sollte Autovectoring verwenden.

5.8 UCI Anpassungen an USB-Tiny-CAN

Das aus dem USB-CAN hervorgegangene USB-Tiny-CAN Gerät basiert ebenfalls auf AN2131SC und SJA1000. Die veränderte Belegung der Steuersignale (/RD, /WR, /CS, /INT, MODE) erfordert jedoch eine Anpassung des UCI-Programms. Der Schaltplan von USB-Tiny-CAN befindet sich im Anhang F.

Die einzelnen Anpassungen sind:

- Neue String-Deskriptoren für Hersteller, Produkt und Revision
- veränderte Felder des Device-Deskriptors: ProductID und Revision. (als VendorID wurde temporär 0x1234 gewählt. Der Hersteller von USB-Tiny-CAN sollte eine VendorID vom USB-Implementer's-Forum erwerben)
- veränderte Port-Konfiguration
- Verlegen des CAN-Interrupt Vektors von INT1 auf INT0 (inklusive Interrupt-Konfiguration)
- Umschreiben der beiden Routinen zum CAN-Registerzugriff *can_rd* und *can_wr* (s. auch usbcan/uci/can.asm)

Im Configuration-Deskriptor (s. Tabelle 5.5) wurde das Feld für Stromaufnahme *MaxPower* mit 0x32 belegt (entspricht 100mA). Die wirkliche Stromaufnahme sollte noch gemessen und im Deskriptor eingetragen werden.

6 USBCAN Treiber und API

6.1 USBCAN Treiber Implementierung

6.1.1 Überblick zu Aufgaben

Die Aufgaben des USBCAN Treibers lassen sich wie folgt zusammenfassen:

- Allozieren einer Treiberkontext-Struktur (Sende- und Empfangs-Puffer, URB-Strukturen für Geräte-Endpoints, etc.) für ein neu angeschlossenes Gerät und deren Freigabe beim Abstecken des Geräts
- Entgegennahme von Nachrichten einer Applikation und deren Zwischenspeicherung
- Umwandlung der Nachrichten in das vom UCI erwartete Format und absenden über den USB
- Analyse der von UCI gelieferten Statusinformation über ein Sendevorgang und entsprechende Benachrichtigung der Anwendung
- Entgegennahme der von UCI empfangenen Nachrichten, deren Zwischenspeichern und Umwandlung in die von einer Anwendung erwartete Struktur und Signalisierung des Empfangs an die Anwendung
- Bereitstellung von Mechanismen, welche einer Anwendung die Steuerung und Konfiguration des USB-CAN-Geräts ermöglichen (über USB-Device-Requests). Hierzu gehört z.B. Starten, Anhalten, Konfiguration des Akzeptanz-Filters und Baudrate, Lese- und Schreibzugriff auf die Register des CAN-Controllers.
- Meldung von Fehlern an eine Anwendung (CAN-Controller Fehler, Überläufe, auch Abstecken des Geräts vom USB)

Zudem soll der Treiber mehrere (bis zu 16) gleichartige Geräte verwalten können.

6.1.2 Treiberkontext

Die Treiberkontext-Struktur ist eines der zentralen Elemente bei der Lösung von oben genannten Aufgaben. Das Listing unten zeigt die einzelnen Felder:

```
struct usb_usbcan {
    struct usb_device *udev;          /* save off the usb device pointer */
    struct usb_interface *interface;  /* the interface for this device */
    unsigned char minor;              /* the starting minor number for this device */
    int open_count;                   /* number of times this port has been opened */

    unsigned char *bulk_in_buffers[NQ];    /* the buffers for received data */
    int bulk_in_size;                      /* the size of the receive buffer */
    __u8 bulk_in_endpointAddr;             /* the address of the bulk in endpoint */

    unsigned char *bulk_out_buffer;        /* send data buffer */
    int bulk_out_size;                     /* the size of the send buffer */
    __u8 bulk_out_endpointAddr;            /* the address of the bulk out endpoint */

    unsigned char *status_buffer;          /* the buffer to receive status data */
    int status_buf_size;                   /* the size of the status buffer */
    __u8 int_endpointAddr;                 /* the address of the interrupt in endpoint */
    int int_endpointInterval;
    struct urb *read_urb[NQ];       /* the urbs for reading data from device */
    struct urb *write_urb;          /* the urb for sending data to device */
```

```
    struct urb *status_urb;          /* the urb for status data from device */

    struct task_struct *task;        /* process using this device */
    struct siginfo sinfo;            /* sinfo send on events */
    struct tx_request tx_req;
    struct uci_err_status status;    /* the status of device */
    struct canmsg_fifo txfifo;       /* transmit fifo */
    struct canmsg_fifo rxfifo;       /* fifo containing received messages */

    struct semaphore sem;            /* locks this structure */
    spinlock_t lock;                 /* locks while operating on fifo ptrs, etc. */
    int queueing;
    wait_queue_head_t inq;           /* reader process waiting  */
    wait_queue_head_t outq;          /* writer process waiting */
};
```

Listing 2: USBCAN Treiberkontext-Struktur

Das Feld *udev* ist ein Zeiger auf Gerätekontext, das vom USB-Subsystem verwaltet wird. Viele USB-Subsystemfunktionen erwarten diesen Zeiger als Parameter. *interface* ist ein Verweis auf eine Struktur, welche ein Interface beschreibt, und wird verwendet um auf die Endpoint-Deskriptoren und somit auf die Endpoint-Information zu gelangen. *minor* enthält die Gerätenummer, *open_count* ist als Referenzzähler gedacht, jedoch unterstützt der Treiber nur eine Anwendung pro Gerät. *bulk_in_buffers* repräsentiert ein Array mit Pointern auf die Bulk-IN Endpoint Puffer, *bulk_in_size* - die Größe eines Bulk-IN Endpoint-Puffers. *bulk_in_endpointAddr* enthält die Adresse des Bulk-IN Endpoints. Die Adresse wird aus dem Endpoint-Deskriptor gelesen und in diesem Element gemerkt, da sie später mehrmals benutzt wird. Die nachfolgenden Elemente enthalten gleichartige Information zu Bulk-OUT und Interrupt-IN Endpoints mit dem Unterschied, dass bei diesen Endpoints je ein Puffer zum Einsatz kommt (daher kein Array mit Pufferpointern sondern je ein Pufferpointer). Beim Interrupt-IN Endpoint wird zusätzlich Abfrage-Interval gespeichert (*int_endpointInterval*).

Für die Kommunikation mit Endpoints verwendet der Treiber USB-Request-Block Strukturen (URBs) des USB-Subsystems. Die Zeiger *read_urb[], write_urb* und *status_urb* verweisen auf die entsprechenden URBs. Die Elemente *task* und *sinfo* sind für Signalisierung verschiedener Ereignisse an die Anwendung notwendig. Die Struktur *tx_req* verwendet der Treiber zur Abwicklung einer CAN-Sendeaufforderung. In *status* wird der Fehlerkontext des CAN-Controllers zwischengespeichert um die Fehleranalyse zu ermöglichen (zum Fehlerkontext s. Abschnitt 5.1 unter Interrupt-Endpoint). Die Elemente *rxfifo* und *txfifo* repräsentieren Empfangs- und Sendepuffer für CAN-Nachrichten. Beide arbeiten nach dem FIFO Prinzip und sind als Ringpuffer realisiert.

Weitere Elemente (*sem, lock, inq, outq)* sind für die Realisierung des wechselseitigen Ausschlusses und Blockierung des Benutzerprozesses vorgesehen.

6.1.3 Interne Abläufe

Als Schnittstelle zu einer Anwendung dient die Gerätedatei */dev/usb/usbcanN*, auf die Standard-Operationen *open*, *close*, *read*, *write* und *ioctl* angewendet werden. *N* steht für die Gerätenummer. Über verschiedene Ereignisse wird die Anwendung mit sog. Real-

Time Signalen informiert. Anders als bei gewöhnlichen Signalen ist bei Real-Time Signalen eine Möglichkeit vorgesehen, diese in eine Signal-Warteschlange der betroffenen Task einzureihen, sodass sie nicht verloren gehen (s. „Chapter 9 Signals“ in [6]). Der Signalhandler einer Anwendung erhält neben der Signalnummer zusätzliche Information über die Art des aufgetretenen Ereignisses (z.B. Bestätigung der Übertragung, Signalisierung der Anzahl von empfangenen Nachrichten, Fehlercodes, Disconnect-Event, etc.). Diese Information ist mit einem Gerät assoziiert, es wird also auch die Gerätenummer geliefert. Anhand dieser Information kann der Signal-Handler notwendige Behandlung veranlassen, z.B. eine passende Callback-Funktion aufrufen.

Senden von Nachrichten

Eine Anwendung übergibt die zu sendenden Nachrichten (vom Typ *struct canmsg_t*) mit einem *write*-Systemaufruf an den Treiber. Dieser speichert sie zunächst in seinem Sende-FIFO ab und formuliert dann einen ersten Sendeauftrag an UCI-Programm im AN2131. Dabei werden bis zu vier Nachrichten in das von UCI erwartete Format umgewandelt und in den Bulk-OUT Endpoint-Puffer eingetragen. Dieser Puffer ist in der *write_urb*-Struktur vermerkt. Nach dem Eintragen der für eine USB-Transaktion notwendigen Parameter (u. a. die Länge der gültigen Daten im Bulk-OUT Puffer) in der *write_urb*-Struktur wird diese an das USB-Subsystem übergeben. Das Subsystem sorgt für die Übermittlung von Bulk-OUT-Pufferdaten zu dem OUT-Endpoint-FIFO des USB-CAN-Geräts. Hat das Gerät die Daten übernommen, gilt die Transaktion als beendet. USB-Subsystem ruft in diesem Fall die in *write_urb* spezifizierte Completion-Funktion auf. Auch im Falle eines Fehlers wird diese Funktion aufgerufen. Die Aufgaben der Completion-Funktion sind die Formulierung des nächsten Sendeauftrags, dessen Übergabe an das USB-Subsystem und Signalisierung von Fehlern.

Ein Sendeauftrag einer Anwendung gilt nur dann als erfolgreich durchgeführt, wenn alle Nachrichten des Auftrags über den CAN-Bus übertragen wurden. Dies ist der Fall, wenn der Empfang von mindestens einem Busteilnehmer bestätigt wurde. Das UCI-Programm bearbeitet Teilaufträge mit jeweils bis zu vier Nachrichten (Kapazität des Bulk-OUT Endpoint-FIFOs) und bestätigt dem Treiber die Ausführung eines jeden Teilauftrag über den Interrupt-IN Endpoint. Erst wenn der komplette Auftrag einer Anwendung bearbeitet wurde, wird die Anwendung über ein Signal benachrichtigt. Im Falle eines Fehlers wird der Anwendung die Anzahl der bis zum Auftreten des Fehlers erfolgreich gesendeten Nachrichten mitgeteilt.

Nachrichten-Empfang

Wie bereits erwähnt ist der USB ein Single-Master System, bei dem nur der Host die Master-Funktion besitzt und somit USB-Transaktionen initiieren darf. Die Functions dürfen erst auf Aufforderung des Hosts ihre Daten schicken. Deswegen muss der Treiber ein Polling-Mechanismus zur periodischen Abfrage des Bulk-IN Endpoints implementieren.

Beim Starten des Geräts auf Anforderung einer Applikation wird neben dem Versetzen des CAN-Controllers in den Betriebszustand das Polling des Bulk-IN Endpoints aktiviert. Dies wird durch Übergabe von zwei Lese-URBs an das USB-Subsystem erreicht. Jedes URB beschreibt eine USB-Transaktion in Upstream-Richtung (zum Host) und enthält einen Pointer zum Bulk-IN Endpoint-Puffer, in dem die gelesenen Daten abgelegt werden. Nachdem das Subsystem die Lesetransaktion gestartet hat, schickt der USB-Host-Controller im Millisekunden-Takt einen sog. IN-Token zum Bulk-IN Endpoint des Geräts.

Ein IN-Token signalisiert dem Gerät die Bereitschaft des Hosts, die Daten zu lesen. Enthält das Bulk-IN Endpoint-FIFO im Gerät die Daten und ist dieses FIFO für den USB-Transfer freigegeben, erfolgt als Reaktion auf IN-Token sofortige Datenübermittlung. Andernfalls antwortet das Gerät mit einem sog. Not-Acknowledge Handshake. Dies bedeutet für den Host, dass die Function noch nicht bereit ist, neue Daten zu senden. In diesem Fall wird der Host einen IN-Token später (im nächsten Takt) erneut senden.

Beim Beenden einer Lese-Transaktion wird ein im Lese-URB angegebener Completion-Handler aufgerufen. Dieser transferiert die empfangenen Nachrichten in das entsprechende FIFO und informiert die Anwendung mit einem Signal über den Empfang. Zusätzlich wird Lese-URB erneut an das Subsystem übergeben um das Polling aufrechtzuerhalten. Die Anwendung kommt an die empfangenen Nachrichten mit einem *read*-Systemaufruf.

Der Entwickler einer CAN-Anwendung wird sich weniger für alle diese Details interessieren. Vielmehr sucht er eine einfache Möglichkeit der Kommunikation über Nachrichtenaustausch. Deshalb wird die Treiberschnittstelle von einer kleinen Bibliothek *libusbcan* gekapselt, auf die im nächsten Abschnitt eingegangen wird.

6.2 Libusbcan als API zur Anwendungsentwicklung

Für die Entwicklung von CAN-Anwendungen auf der Basis von USB-CAN wurde eine kleine Bibliothek zur Verfügung gestellt. Eine Übersicht über die einzelnen Funktionen befindet sich im Anhang B.

Die Transferfunktionen verwenden eine Struktur vom Typ *canmsg_t,* welche die CAN-Nachricht enthält:

```
typedef struct {
    unsigned long       id;
    unsigned int        flags; // std/ext, rtr, dlc
    unsigned char       msgdata[CAN_MSG_LENGTH];
    struct timeval      ts; // time stamp for received msg
} canmsg_t;
```

Listing 3: CAN-Message Struktur *canmsg_t*

Im *id* Feld ist der Identifier der Nachricht untergebracht. Das Element *flags* spezifiziert den Typ (Standard, Extended, Remote) und codiert im unteren Nibble die Anzahl von gültigen Datenbytes (Länge der Nachricht, auch Data Length Code oder DLC genannt) im Daten-Array *msgdata*. Das Feld *ts* ist nur bei empfangenen Nachrichten gültig und enthält einen Zeitstempel (Mikrosekunden-Auflösung).

Folgendes Listing zeigt eine einfache Anwendung einiger Bibliotheksfunktionen:

```
#include <stdio.h>
#include <stdlib.h>
#include <libusbcan.h>

void tx_callback (unsigned long cnt)
{
    printf("transmited messages: %lu\n", cnt);
}
```

```
void error_callback (unsigned long err)
{
      can_print_err(err);
}

void rx_callback (unsigned long cnt)
{
      canmsg_t msg[4];
      int i,j, len;

      i = can_receive_msg_buf(devh, msg, (cnt > 4 ? 4 : cnt));
      if(i<0)
         fprintf(stderr, "read returns %d,  errno %d, %s\n", i, errno, strerror(errno));

      for (i = 0; i < cnt; i++) {
            printf("%d: ts %12lu.%05lu, id %lu,  %s, data: ",
                  i, msg[i].ts.tv_sec, msg[i].ts.tv_usec,
                  msg[i].id, (msg[i].flags & CAN_EFF ? "ext. frame":"std. frame"));
            len = (msg[i].flags & CAN_DLC)%9;
            for (j = 0; j < len; j++)
                  printf("%#x ", msg[i].msgdata[j]);
            printf("\n");
      }
}

int main(void)
{
         int devh, ret;
         canmsg_t msg;
         char *msg_to_send = "c_msg0";

         memset(msg, 0, sizeof(msg));
         msg.id = 17;
         sprintf(msg.msgdata, msg_to_send);
         msg.flags |= strlen(msg_to_send);

         can_init();

         devh = can_open(0, O_NOBLOCK);
         if (devh < 0) {
                  perror("unable to open device 0");
                  exit(1);
         }
         can_set_callback(devh, CAN_ERR_CB, error_callback);
         can_set_callback(devh, CAN_TX_CB, tx_callback);
         can_set_callback(devh, CAN_RX_CB, rx_callback);
         ret = can_set_baud(devh, CAN_BAUD_1000K);
         if (ret < 0) {
                  perror("unable to set baudrate");
                  exit(1);
         }
```

```
        ret = can_start(devh);
        if (ret < 0) {
                fprintf(stderr,"unable to start CAN-Controller: %d\n", ret);
                exit(1);
        }
        ret = can_send_msg(devh, &msg);
        if (ret <= 0) {
                fprintf(stderr,"unable to send message: %d\n", ret);
                exit(1);
        }
        can_stop(devh);
        can_close(devh);
        return 0;
}
```

Listing 4: Anwendung von Libusbcan API-Funktionen

Das obige Programm kann als ein einfachstes Bus-Monitor dienen. Hierzu ist eine kleine Erweiterung vorzunehmen: vor dem *can_stop()*-Aufruf wird eine Endlosschleife eingefügt, in der das Programm schlafen gelegt wird. Beim Empfangen wird automatisch die *rx_callback()* Funktion aufgerufen, welche die Nachrichten vom Treiberpuffer ausliest und auf Standardausgabe ausgibt. Ein etwas umfangreicheres Beispiel ist die Anwendung usbcan/lib/testlibusbcan.c, welche zum Testen des Treibers und der Bibliotheksfunktionen geschrieben wurde.

7 Zusammenfassung

Ziel der Diplomarbeit war die Entwicklung von Software für eine Prototypen-Platine, welche zwei USB-Geräte realisiert. Bei dem ersten USB-Gerät handelte es sich um eine USB zu Microcontroller Schnittstelle, bei dem zweiten - um einen USB zu CAN-Bus Adapter. Zusätzlich musste auf der Host-Seite eine Schnittstelle zur Entwicklung von entsprechenden Linux-Applikationen bereitgestellt werden.

Alle in der Problemstellung genannten Teilaufgaben und Anforderungen konnten gelöst werden. Basierend auf dem für das USB-AVR-Gerät vom Aufgabensteller vorgeschlagenen Puffertransfer-Mechanismus können nun anwendungsspezifische Kommunikationsprotokolle definiert werden. Der zugrunde liegende Usermode Treiber (als Bibliothek libavrp) sollte auch mit neueren Versionen des Linux-Kernels (2.6.x) funktionieren.

Begründet mit der Komplexität der Treiberaufgaben wurde für das USB-CAN-Gerät ein Kernelmode Treiber vorgeschlagen. Der entwickelte Treiber *usbcan* kann mit Linux-Kernel 2.4.x verwendet werden und unterstützt bis zu 16 gleichartige USB-CAN-Geräte, alle CAN-Bitraten, asynchronen Nachrichtenempfang mit Pufferung und Signalisierung und benachrichtigt eine Anwendung über CAN-Controller Fehler. Bei einem Umstieg auf 2.6.x Kernel muss der Treiber neu geschrieben werden, da die Programmierschnittstelle des USB-Subsystems bei neueren Kernelversionen geändert wurde.

Die Anwendung-Treiber-Schnittstelle wurde in beiden Fällen von Bibliotheken gekapselt um verhältnismäßig einfachere Anwendungsentwicklung zu ermöglichen. Die entwickelte Software wurde auch auf einem Host mit USB-Controller der OHCI-Familie erfolgreich getestet.

Als Hauptziel wird nun die Benutzung der entwickelten Software angestrebt. Alle Programme stehen unter GNU General Public License (GPL, [7]) zur Verfügung. Dies soll sicherstellen, dass die Software für alle Benutzer frei ist und somit weitergegeben werden kann. Darüber hinaus besteht durch die GPL die Möglichkeit, dass Anwender die Software entsprechend eigenen Vorstellungen und Bedürfnissen anpassen.

8 Literatur

[1] USB 2.0, Hans Joachim Kelm (Hrsg.)
Franzis' Verlag GmbH, 85586 Poing, 2001
ISBN 3-7723-7965-6

[2] Controller-Area-Network: CAN; Grundlagen, Protokolle, Bausteine, Anwendungen
Konrad Etschberger
München, Wien: Hanser, 1994
ISBN 3-446-17596-2

[3] CAN Controller Area Network; Grundlagen und Praxis
Wolfhard Lawrenz (Hrsg.)
Heidelberg: Hüthig, 1994
ISBN 3-7785-2263-7

[4] Programming Guide for Linux USB Device Drivers
Detlef Fliegl, 2000
http://usb.cs.tum.edu/usbdoc

[5] Linux Device Drivers, 2st Edition
Alessandro Rubini, Jonathan Corbet
O'Reilly & Associates, Inc.

[6] Understanding the Linux Kernel
Daniel P. Bovet & Marco Cesati
O'Reilly & Associates, Inc., 2001
ISBN 0-596-00002-2

[7] GNU General Public License
Homepage des GNU Projekts `http://www.gnu.org/licenses/gpl.html`

Außerdem wurden folgende Datenblätter verwendet:

[8] EZ-USB Technical Reference Manual, Version 1.10, Cypress Semiconductor

[9] SJA1000 Stand-alone CAN controller, Philips Semiconductors, 2000

[10] SJA1000 Stand-alone CAN controller APPLICATION NOTE AN97076, Philips Semiconductors, 1997

[11] Atmel ATmega8(L) Datasheet, Rev. 2486L-AVR-10/03, Atmel Corporation, 2003

[12] AVR Instruction Set, Rev. 0856D-AVR-08/02, Atmel Corporation, 2002

[13] Dokumentation zum can4linux Treiber, Version 3.1 (aus can4linux.tgz)

[14] USB-Tiny-CAN Schaltplan, Version 1.0, MHS-Elektronik, 2004

Anhang

A Libavrp API

int open_avr (int dev_num);

Öffnet einen Kommunikationskanal zum USB-AVR-Gerät.

Parameter:

dev_num - die Nummer des Geräts. Das erste angeschlossene Gerät besitzt als Nummer 0, das zweite - 1, usw.

Rückgabewert:

-1: falls irgendein Fehler aufgetreten ist.

Falls erfolgreich, ein Handle für den geöffneten Kommunikationskanal. Dieses Handle wird als Argument für Transferfunktionen verwendet.

int close_avr(int dev_num);

Schließt einen geöffneten Kommunikationskanal.

Parameter:

dev_num - Handle für einen geöffneten Kommunikationskanal

Rückgabewert:

0: falls erfolgreich geschlossen
negativer Wert im Falle eines Fehlers

int send_to_avr (int dev_num, unsigned char *buffer, unsigned short len);

Sendet Daten zum AVR-RX-Puffer.

Parameter:

dev_num - Handle für einen geöffneten Kommunikationskanal
buffer - Puffer mit Daten, die gesendet werden
len - Anzahl Datenbytes im Puffer **buffer**

Rückgabewert:

-1: **len** ist zu groß, Handle ungültig oder **buffer** gleich 0
-2: kein Zugriff auf AVR möglich
-3: AVR /RDY Signal nicht aktiv
Bei einem erfolgreichen Transfer wird die Anzahl der übertragenen Bytes (= **len**) zurückgegeben.

int read_from_avr (int dev_num, unsigned char *buffer);

Liest Daten vom AVR-TX-Puffer.

Parameter:

dev_num - Handle für einen geöffneten Kommunikationskanal
buffer - Puffer, in dem gelesenen Daten abgelegt werden

Rückgabewert:

-1: Handle ungültig, **buffer** gleich 0 oder Fehler
-3: AVR /RDY Signal nicht aktiv
0: AVR-TX-Puffer leer
Bei einem erfolgreichen Transfer wird die Anzahl der gelesenen Bytes geliefert.

int avr_data_available (int dev_num);

Liefert die Anzahl von gültigen Bytes im AVR-TX-Puffer.

Parameter:

dev_num - Handle für einen geöffneten Kommunikationskanal

Rückgabewert:

-1: Handle ungültig oder Fehler
-3: AVR /RDY Signal nicht aktiv
Bei einem erfolgreichen Transfer wird die Anzahl der gültigen Bytes im AVR-TX-Puffer geliefert.

int read_avr_sram (int dev_num, unsigned char *buffer);

Liest kompletten AVR-SRAM (1024 Byte) aus.

Parameter:

dev_num - Handle für einen geöffneten Kommunikationskanal
buffer - Puffer, in dem gelesenen Daten abgelegt werden (Puffergröße >= 1024 Byte!)

Rückgabewert:

-1: Handle ungültig oder Fehler
-3: AVR /RDY Signal nicht aktiv
Bei einem erfolgreichen Transfer wird 1024 geliefert.

B Libusbcan API

int can_init (void);

Initialisiert interne Strukturen. Diese Funktion muss am Anfang des Programms aufgerufen werden!

Rückgabewert:
0: falls Initialisierung erfolgreich
negativer Wert im Falle eines Fehlers

int can_open (int dev_num, unsigned char flags);

Öffnet ein USB-CAN-Gerät.

Parameter:
dev_num - Gerätenummer. Das erste angeschlossene Gerät besitzt die Nummer 0, zweite - 1, usw.

flags - gibt an, ob Sende- und Empfangsoperationen blockieren

O_BLOCK: - falls keine Nachrichten im RX-FIFO, blockiert die Empfangsfunktion, bis Nachricht empfangen wird
- falls TX-FIFO voll, blockiert die Sendeoperation, bis im TX-FIFO wieder Platz ist.

O_NOBLOCK: kein Blockieren der Sende- und Empfangsoperationen

Rückgabewert:
Handle für das Gerät (wird als Argument bei allen anderen Funktionen benutzt)
negativer Wert im Falle eines Fehlers

int can_close (int dev_num);

Schließt ein geöffnetes Gerät.
(evtl. installierte Callbacks werden deinstalliert).

Parameter:
dev_num - Handle für Gerät

Rückgabewert:
Referenzzähler für dieses Gerät
negativer Wert im Falle eines Fehlers

int can_start (int dev_num);

Startet den CAN-Controller.

Parameter:
dev_num - Handle für Gerät

Rückgabewert:
0: Controller bereits gestartet
Mode-Register des CAN-Controllers (falls Bit 0 gesetzt,
konnte nicht gestartet werden)
negativer Wert im Falle eines Fehlers

int can_stop (int dev_num);

Versetzt CAN-Controller in den Reset-Modus.

Parameter:
dev_num - Handle für Gerät

Rückgabewert:
0: Operation erfolgreich
negativer Wert im Falle eines Fehlers

int can_set_callback (int dev_num, unsigned char type, void(*callback)(unsigned long));

Installiert Callback-Funktionen für verschiedene Ereignisse
(Nachricht(en) erfolgreich abgesendet, Nachrichtenempfang, Fehler).

Parameter:
dev_num - Handle für Gerät
type - Typ der Callback-Funktion
(CAN_RX_CB, CAN_TX_CB, CAN_ERR_CB)
callback - Zeiger auf Callback-Funktion

Argument der Callback-Funktion:
bei CAN_TX_CB - Anzahl erfolgreich gesendeten Nachrichten
bei CAN_RX_CB - Anzahl empfangener Nachrichten
bei CAN_ERR_CB - Fehlercode (wird an **can_print_err** übergeben)

Rückgabewert:
0: Operation erfolgreich
negativer Wert im Falle eines Fehlers

int can_set_baud (int dev_num, unsigned char baud);

Konfiguriert CAN-Controller für angegebene Bitrate.

Parameter:

dev_num - Handle für Gerät

baud - Makro für Bitrate: CAN_BAUD_10K, CAN_BAUD_20K, CAN_BAUD_50K, CAN_BAUD_100K, CAN_BAUD_125K, CAN_BAUD_250K, CAN_BAUD_500K, CAN_BAUD_800K, CAN_BAUD_1000K

Rückgabewert:

0: Operation erfolgreich

negativer Wert, falls Fehler aufgetreten

int can_set_baud_custom (int dev_num, unsigned char btr0, unsigned char btr1);

Setzt Bus-Timing Register 0 und 1 auf angegebene Werte.
Wird verwendet um andere Bitraten (als vordefinierte) zu ermöglichen.

Parameter:

dev_num - Handle für Gerät

btr0 - Wert für BTR0 Register des CAN-Controllers

btr1 - Wert für BTR1 Register des CAN-Controllers

Rückgabewert:

0: Operation erfolgreich

negativer Wert, falls Fehler aufgetreten

int can_set_acc_mask (int dev_num, unsigned long mask);

Setzt Acceptance Mask Register 0 - 3.

Parameter:

dev_num - Handle für Gerät

mask - Wert für Acceptance Mask Register 0 - 3

Rückgabewert:

> 0: Operation erfolgreich

negativer Wert, falls Fehler aufgetreten

int can_set_acc_code (int dev_num, unsigned long code);

Setzt Acceptance Code Register 0 - 3

Parameter:

dev_num - Handle für Gerät

code - Wert für Acceptance Code Register 0 - 3

Rückgabewert:

> 0: Operation erfolgreich

negativer Wert, falls Fehler aufgetreten

int can_send_msg (int dev_num, canmsg_t *msg);

Sendet eine CAN-Nachricht. Diese Funktion
darf nicht aus einem Callback-Handler aufgerufen werden!

Parameter:

dev_num - Handle für Gerät
msg - CAN-Nachricht

Rückgabewert:

1: Nachricht erfolgreich gesendet
0: Transfer wegen CAN-Fehler abgebrochen
negativer Wert im Falle eines Fehlers

int can_send_msg_buf (int dev_num, canmsg_t *msg_buf, int size);

Sendet mehrere CAN-Nachrichten. Diese Funktion
darf nicht aus einem Callback-Handler aufgerufen werden!

Parameter:

dev_num - Handle für Gerät
msg_buf - CAN-Nachrichten-Puffer
size - Anzahl Nachrichten im Puffer **msg_buf**

Rückgabewert:

0: Transfer wegen CAN-Fehler abgebrochen
negativer Wert im Falle eines Fehlers
Anzahl der erfolgreich gesendeten Nachrichten

int can_receive_msg (int dev_num, canmsg_t *msg);

Liest eine CAN-Nachricht aus dem RX-FIFO des Treibers.

Parameter:

dev_num - Handle für Gerät
msg - Puffer, in dem gelesene Nachricht abgelegt wird

Rückgabewert:

0: keine Nachrichten im RX-FIFO des Treibers
1: eine Nachricht erfolgreich gelesen
negativer Wert im Falle eines Fehlers

int can_receive_msg_buf (int dev_num, canmsg_t *msg_buf, int size);

Liest mehrere Nachrichten aus dem RX-FIFO des Treibers.

Parameter:

dev_num - Handle für Gerät
msg_buf - Puffer, in dem gelesene Nachrichten abgelegt werden
size - Anzahl der zu lesenden Nachrichten

Rückgabewert:

0: keine Nachrichten im RX-FIFO des Treibers
Anzahl der erfolgreich gelesenen Nachrichten
negativer Wert im Falle eines Fehlers

int can_reset (int dev_num);

Initialisiert CAN-Controller und versetzt ihn in den Reset-Modus
(125 Kbit/s, accept. code = 0x00000000, accept. mask = 0xffffffff)

Parameter:

dev_num - Handle für Gerät

Rückgabewert:

0: Operation erfolgreich
negativer Wert im Falle eines Fehlers

void can_print_err (unsigned int err);

Schreibt Error-String(s) auf Standard-Error.
Kann aus dem Error-Callback aufgerufen werden um
ein String zum Error-Code zu erhalten.

Parameter:

err - Fehler-Code

void can_debug_on(int level);

Setzt Debug-Level.

Parameter:

level - Level-Code (0 bis 3)

C UCI Vendor-Request und Parameter

UCI Firmware unterstützt neben Standard-Requests folgenden Vendor-Request:
UCI_CAN_CHIP_ACCESS (Code 0xB0)

Parameter von UCI_CAN_CHIP_ACCESS Request geordnet nach *wIndexL*:

Name	*bmRequest-Type*	*bRequest*	*wValue*	*wIndex*	*wLength*
UCI_READ_CAN_REGISTER	0xC0	0xB0	Reg. Adr. (Low)	0x01 (Low)	0x01
UCI_WRITE_CAN_REGISTER	0x40	0xB0	Reg. Adr. (Low) Reg. Value (High)	0x02 (Low)	0x00
UCI_GET_CAN_REGISTERS	0xC0	0xB0	Startadr. (Low)	0x03 (Low)	Anzahl Reg.
UCI_SET_BUS_TIMINGS	0x40	0xB0	BT Reg.0 (Low) BT Reg.1 (High)	0x04 (Low) CODE (High)	0x00
UCI_SET_ACCEPT_CODE	0x40	0xB0	0x00	0x05 (Low)	0x04
UCI_SET_ACCEPT_MASK	0x40	0xB0	0x00	0x06 (Low)	0x04
UCI_CAN_RESET	0x40	0xB0	0x00	0x07 (Low)	0x00
UCI_CAN_START	0xC0	0xB0	0x00	0x08 (Low)	0x01
UCI_CAN_STOP	0x40	0xB0	0x00	0x09 (Low)	0x00

Anmerkungen:

UCI_SET_BUS_TIMINGS: Setzt BTR0 und BTP1 abhängig von *wIndexH*. Falls CODE in *wIndexH* ungleich 0 ist, werden Bus-Timing Register abhängig von CODE mit vordefinierten Werten konfiguriert (s. Tabelle unten). Andernfalls werden BTR0 und BTR1 mit Werten aus *wValue* belegt.

CODE	Baudrate, Kbit/s	CODE	Baudrate, Kbit/s
0x00	in *wValue*	0x05	125
0x01	10	0x06	250
0x02	20	0x07	500
0x03	50	0x08	800
0x04	100	0x09	1000

UCI_CAN_RESET: Initialisiert den CAN-Chip und versetzt ihn in den Reset-Modus

UCI_CAN_START: Versetzt CAN-Chip in Operation-Modus (SingleAcceptanceFilterMode) (liefert ein Byte mit Status: falls Bit 0 im gelieferten Statusbyte gesetzt ist, konnte nicht in Operation-Modus gehen, evtl. Hardware Reset aktiv)

UCI_CAN_STOP: Versetzt CAN-Chip in Reset-Modus

D AVRPIPE Vendor-Requests, Parameter

RQ_AVR_DATA_AVAILABLE: Liefert Anzahl Bytes in dem AVR-TX-Buffer

RQ_SEND_TO_AVR: Startet Transfer zum AVR (Datentransfer zu AVR-RX-Buffer)

RQ_READ_FROM_AVR: Startet Transfer vom AVR (Datentransfer aus AVR-TX-Buffer)

RQ_READ_AVR_SRAM: Liest AVR-SRAM aus (1024 Byte)

Name	*bmRequest-Type*	*bRequest*	*wValue*	*wIndex*	*wLength*
RQ_AVR_DATA_AVAILABLE	0xC0	0xB0	0x00	0x00	0x04
RQ_SEND_TO_AVR	0xC0	0xB1	0x00	0x00	0x02
RQ_READ_FROM_AVR	0xC0	0xB2	0x00	0x00	0x02
RQ_READ_AVR_SRAM	0xC0	0xB3	0x00	0x00	0x02

Anmerkung:
Alle obigen Transfers übertragen Daten von bzw. zu internen Puffern des AN2131. Diese Puffer können dann einfach mit dem FIRMWARE-LOAD Request 0xA0 des AN2131 vom bzw. zum Host übertragen werden.

E an2131ctl Kommandozeilenoptionen

NAME
an2131ctl - manual page for an2131ctl - 0.1

SYNOPSIS
an2131ctl [OPTIONS]

DESCRIPTION
Controls EZ-USB device (Cypress' an2131) as specified by OPTIONS. Using this tool you can access to internal RAM of the EZ-USB device, write or read from its endpoints or send control requests to it. You can also use this program as a firmware downloader to the EZ-USB microcontroller. an2131ctl was written as part of the Linux USBCAN Project (mainly for debugging).

OPTIONS
-h, --help
display this help and exit.

-v, --version
display version information and exit.

-l FILE, --load=FILE
download the 8051-program from FILE (assumes that file is in intel hex format; for binary files use -b option).

-b, --binary
specifies downloading binary file.

-q, --quiet
no user interaction (useful for scripts).

-g, --go
run firmware program (load before running !!!).

-s, --stop
stop program (put cpu into reset state and hold it).

-d, --dumpreg
EZ-USB registers / bulk buffers dump.

-R, --Reset
reset (cause re-enumeration).

-S ADDR, --Spy=ADDR
spy memory region at ADDR

-D DEVPATH, --Device=DEVPATH
DEVPATH specifies "usbfs" device path to EZ-USB device (like /proc/bus/usb/001/003). This is optional and mainly for use in hotplug scripts.

-I ID, --Id=ID
ID specifies Vendor- and Product-ID of USB device in format VID/PID/*; default is 547/2131/*

-r, --read
read from endpoint if specified by -e option or from memory if –m option is used.

-w BYTE, --write=BYTE
write BYTE to endpoint if specified by -e option or to memory if -m option is used. BYTE will be written SIZE times (SIZE is given with -c option or 64 default).

-m ADDR, --memory=ADDR
operate on memory starting at address ADDR (read or write depends on -r / -w option).

-e EP, --endpoint=EP
endpoint EP to read from or to write to (read / write is specified by -r / -w option).

-c SIZE, --count=SIZE
specifies how many bytes to read/to write (see -r, -w, -e and -m options); SIZE is 64 default.

-C REQ, --Ctrl=REQ
REQ is USB-Device-Request (*bRequest*), hexadecimal! If this option is used, you also have to specify request type (-t), index with -m, value with -e and length with -c.

-t TYPE, --type=TYPE
TYPE is USB-Device-Request type (*bmRequestType*), hexadecimal!

F USB-Tiny-CAN Schaltplan

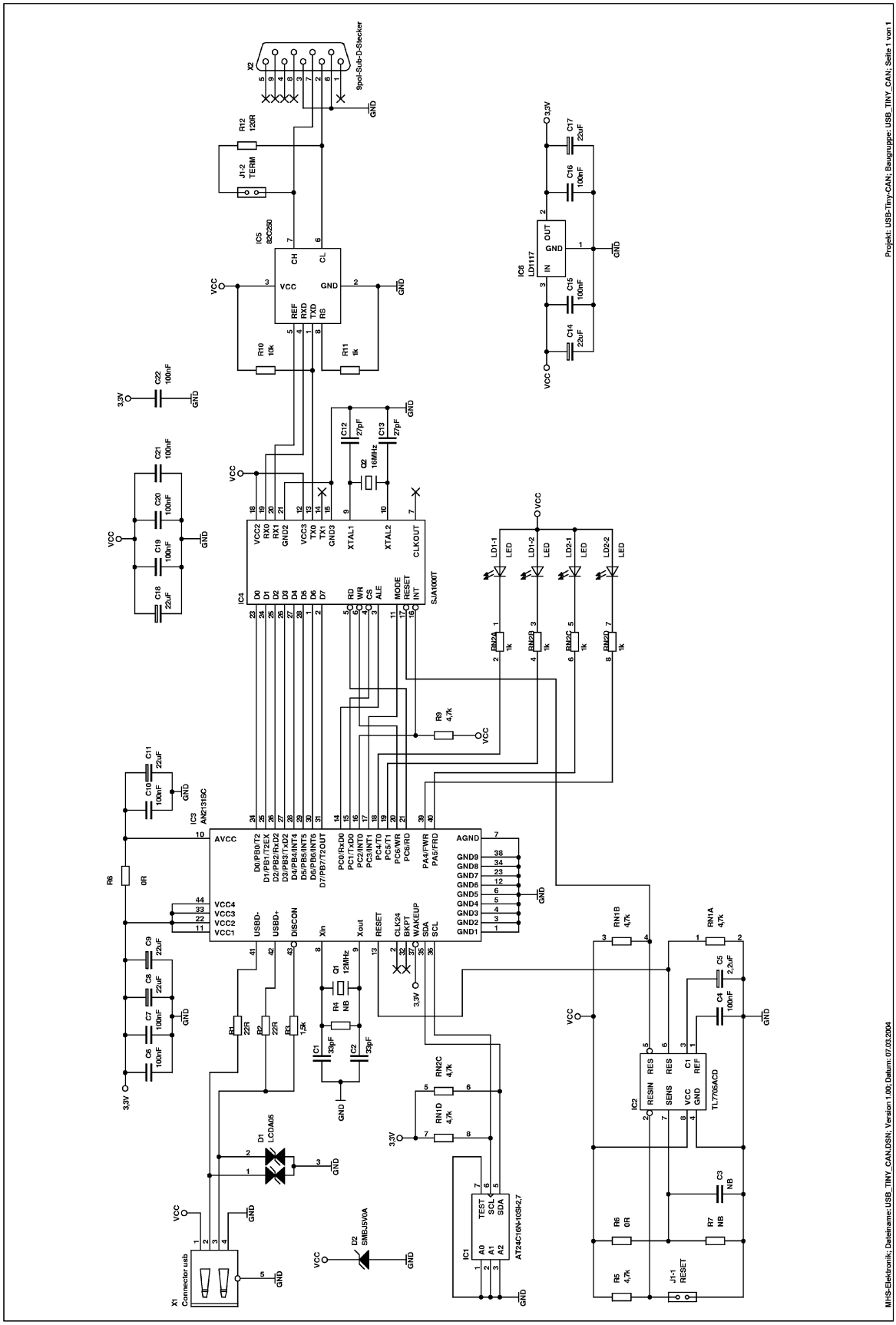

Abbildung 17: USB-Tiny-CAN Schaltplan

G Platinen-Fotos

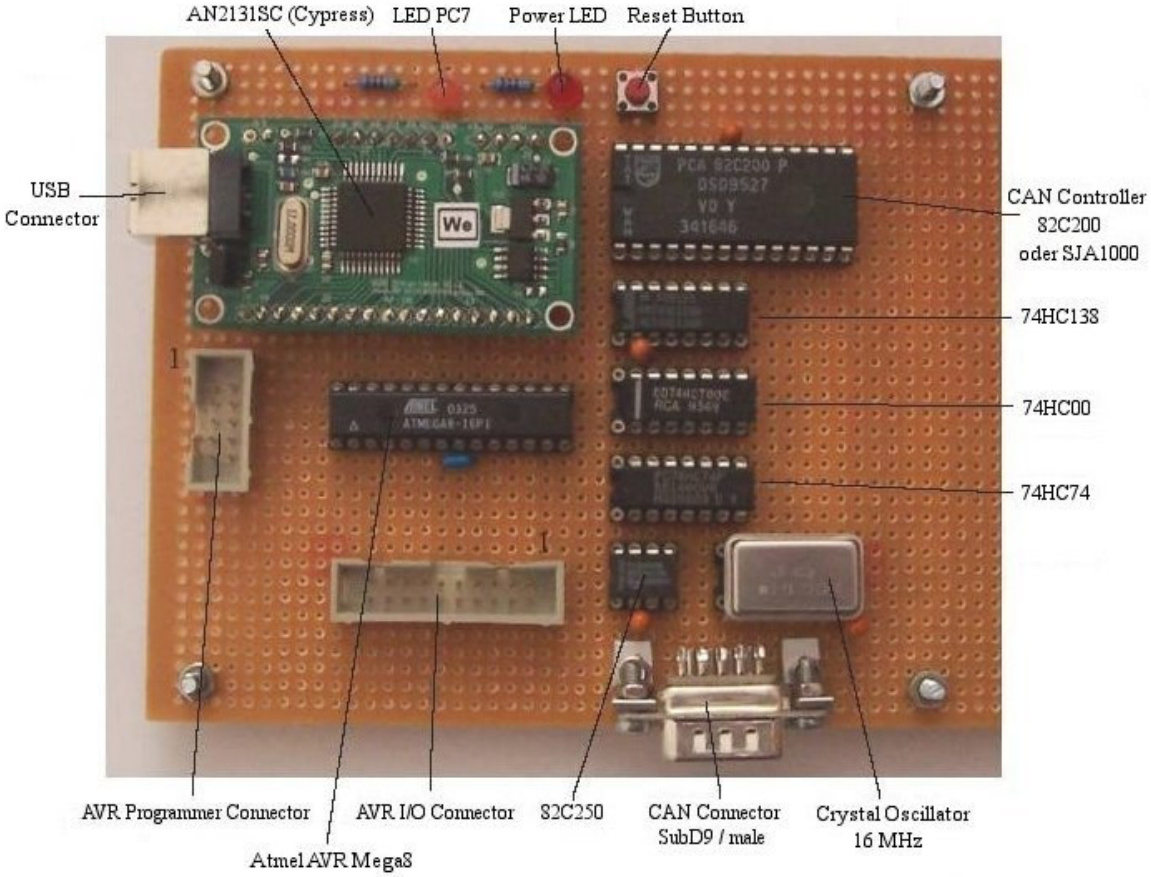

Abbildung 18: Prototypen-Platine

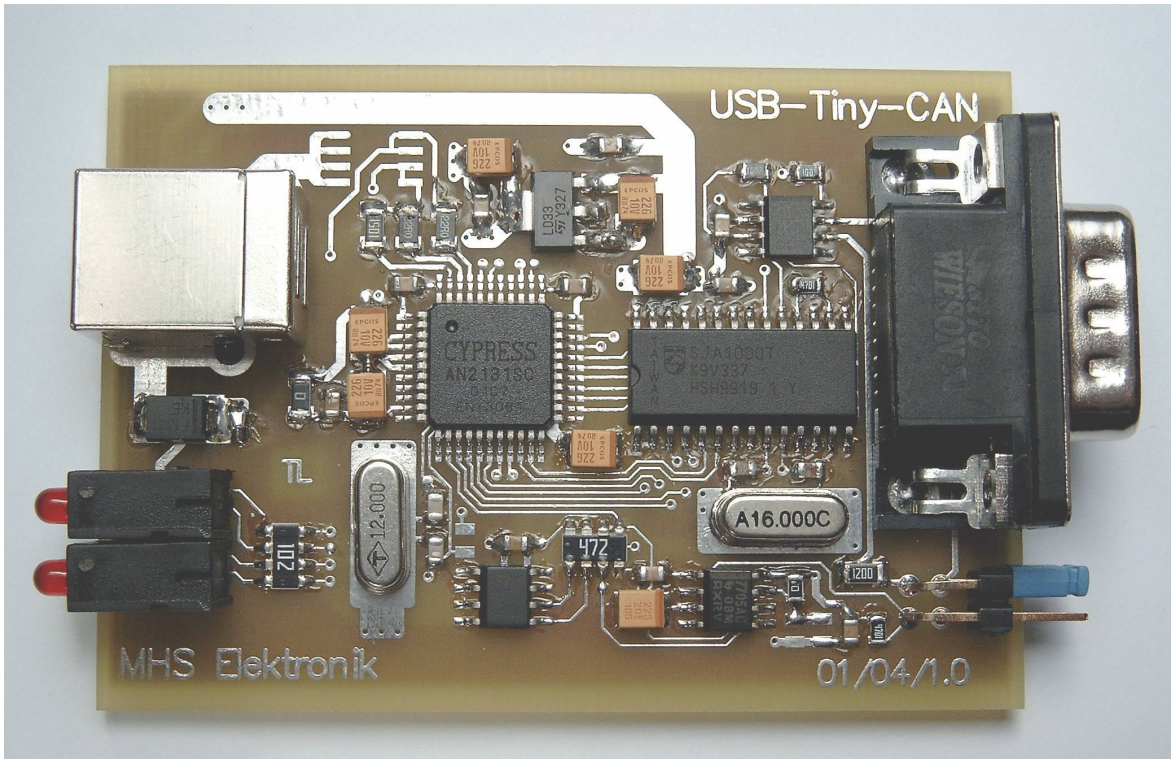

Abbildung 19: USB-Tiny-CAN