Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Oliver Schell

Spezifikation und Realisierung der Software eines USB-2-X Moduls

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Oliver Schell

Spezifikation und Realisierung der Software eines USB-2-X Moduls

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Ing. Franz Korf
Zweitgutachter : Prof. Dr. rer. nat. Stephan Pareigis

Abgegeben am 23. September 2008

Oliver Schell

Thema der Bachelorarbeit

Spezifikation und Realisierung der Software eines USB-2-X Moduls

Stichworte

Eingebettetes System, Mikroprozessor, ARM7, Schnittstellenkonverter, Gateway, USB, RS485, CAN, SPI

Kurzzusammenfassung

Ziel dieser Arbeit ist es, die Funktionalität eines Schnittstellenkonverters auf dem 32 Bit Prozessor AT91SAM7X256 mit ARM7-Kern von Atmel zu realisieren. Im ersten Ansatz soll dabei die Umsetzung von USB nach CAN, RS485 oder SPI implementiert werden. Für die Realisierung wird ein Entwicklungsboard für den AT91SAM7X256 Prozessor mit den notwendigen Schnittstellen genutzt.

Oliver Schell

Title of the paper

Software Specification and Realisation of an USB-2-X Module

Keywords

Embedded System, Microprocessor, ARM7, Interface Converter, Gateway, USB, RS485, CAN, SPI

Abstract

The goal of this final thesis is to implement the functionality of an interface converter on the famous AT91SAM7X256 32 bit processor with ARM7 core. The first approach ist to allow communication between USB and CAN, RS485 or SPI. The project will be realized on an evaluation board for the AT91SAM7X256 processor which provides the necessary interfaces.

Danksagung

Ein besonderer Dank geht an meine Eltern für die Unterstützung während meines Studiums. Für seine qualifizierten Ratschläge möchte ich mich bei meinem Betreuer Franz Korf bedanken. Für konstruktive Kritik möchte ich mich bei meiner Freundin, Göran Eggers und Olav Kahlbaum bedanken. Einen weiteren Dank möchte ich meinen Lektoren und Editoren aussprechen, die sich aufopferungsvoll durch die Seiten gekämpft haben. Für die Ausgabe des Themas danke ich der Firma TRINAMIC Motion GmbH.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

ACK	Acknowledged
AIC	Advanced Interrupt Controller
API	Application Programming Interface
BRGR	Baud Rate Generator Register
BRP	Baud Rate Prescaler
CAN	Controller Area Network
CD	Compact Disc
CDC	Communication Device Class
CDT	C/C++ Development Tools
COM	Communication
CRC	Cyclic Redundancy Check
DBGU	Debug Unit
DLC	Data Length Code
DLE	Data Link Escape
DLL	Dynamic Link Library
DPR	Dual Ported RAM
EEPROM	Electrically Erasable Programmable ROM
EHCI	Enhanced Host Controller Interface
EOP	End Of Packet
ETX	End Of Text
FIFO	First In First Out
GAP	'Get Axis Parameter'
GCC	GNU Compiler Collection
GND	Ground
GUID	Global Unique Identifier

ICs	Integrated Circuits
IDE	Identifier Extension
IFS	Inter Frame Space
IIC	Inter Integrated Circuit
IO	Input Output
IPT	Information Processing Time
JTAG	Joint Test Action Group
LIN	Local Interconnect Network
LSB	Last Significant Bit
MISO	Master In Slave Out
MOB	Message Object Buffer
MOSFET	Metal Oxide Semiconductor Field-Effect Transistor
MOSI	Master Out Slave In
MSB	Most Significant Bit
NAK	Not Acknowledged
NRZI	Non Return to Zero Invert
NYET	No Response Yet
OCD	On-Chip Debugger
OHCI	Open Host Controller Interface
PC	Personal Computer
PID	Packet Identifier
PIO	Programmable IO
RAM	Random Access Memory
RDR	Receive Data Register
RDRF	Receive Data Register Full
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
ROR	'Rotate Right'
RTOS	Real-Time Operating System
RTR	Remote Transmission Request
RTS	Request To Send

SJW	Synchronisation Jump Width
SOF	Start Of Frame
SPI	Serial Peripheral Interface
SRAM	Static RAM
SRR	Substitute Remote Request
STX	Start Of Text
TDR	Transmit Data Register
TXCOMP	Transfer Complete
TXPKTRDY	Transmit Packet Ready
UDP	USB Device Port
UHCI	Universal Host Controller Interface
USART	Universal Synchronous Asynchronous Receiver Transmitter
USB	Universal Serial Bus

1. Einleitung

1.1. Bezug zum Ist-Zustand

Die Firma TRINAMIC entwickelt und vertreibt Integrated Circuits (ICs)[1] und darauf aufbauende Module für die Ansteuerung von elektronisch kommutierten Motoren[2] im Sub-Kilowatt-Bereich. Sowohl die Module als auch die ICs sind hierbei mit einer Vielzahl unterschiedlicher Schnittstellen ausgestattet. Um diese an jeden handelsüblichen Personal Computer (PC) anschließen zu können, hat TRINAMIC einen universellen Schnittstellenkonverter entwickelt (USB-2-X). Dieser bietet eine Umsetzung von Universal Serial Bus (USB) nach Controller Area Network (CAN), Local Interconnect Network (LIN), RS485, Inter Integrated Circuit (IIC) und Serial Peripheral Interface (SPI)[3]. Das Design dieses Bausteins erreicht speziell bei der Kommunikation mit einer CAN Schnittstelle nicht mehr die gewünschte Performance und die im USB-2-X Modul verwendete Prozessorplattform (68HC08)[4] ist innerhalb der Firma TRINAMIC einzigartig und erschwert deshalb die Wartbarkeit sowie kundenspezifische Änderungen deutlich.

1.2. Ziel der Arbeit

Ziel dieser Arbeit ist es nun, die Funktionalität des Schnittstellenkonverters auf eine neue Prozessorplattform aufzusetzen. Hierfür wird der auch in einer Reihe von anderen Projekten bei TRINAMIC bereits erfolgreich eingesetzte 32 Bit Prozessor AT91SAM7X256 mit ARM7-Kern von Atmel zum Einsatz kommen. Durch die bereits gesammelten Erfahrungen mit dem AT91SAM7X256 sollen die Wartbarkeit und

[1] *ICs* sind auf kleinstem Raum, auf einem einzigen Stück Halbleitersubstrat, untergebrachte (integrierte) elektronische Schaltungen. Vgl. (Müller u. a., 2007, S. 8)

[2] *Elektronisch kommutierte Motoren* erzeugen das für den Motorenlauf notwendige Wechselfeld über eine elektronische Schaltung, dem so genannten "Frequenzumrichter". Vgl. Böbel (2005)

[3] *CAN, LIN, RS485, IIC und SPI* gehören zur Familie der seriellen Datenbusse.

[4] Der von der Firma Freescale hergestellte *68HC08* ist der Nachfolger des 68HC05, dem meistverkauften 8 Bit Controller weltweit.

kundenspezifische Änderungen erleichtert und letztendlich Kosten reduziert werden. Im ersten Ansatz soll dabei die Umsetzung von USB nach CAN, RS485 und SPI implementiert werden. Für die Realisierung steht ein Entwicklungsboard für den AT91SAM7X256 Prozessor mit den notwendigen Schnittstellen zur Verfügung.

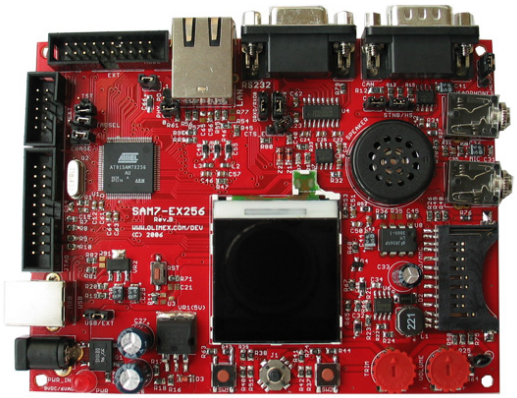

Abbildung 1.1.: Das Entwicklungsboard

Wie das System realisiert wurde, wird in den folgenden Kapiteln beschrieben. Begonnen wird in Kapitel 1 mit der Betrachtung der Aufgaben und dem Einsatzgebiet des USB-2-X Moduls. Anschließend werden in Kapitel 2 die Grundlagen der beteiligten Komponenten beschrieben. Darauf aufbauend werden in Kapitel 3 die Aufgaben der Komponenten genauer betrachtet und in Kapitel 4 das Design der Komponenten diskutiert. Die Implementierung der einzelnen Komponenten wird in Kapitel 5 beschrieben. Ebenso stellt Kapitel 5 kurz das entwickelte User Interface vor, um einen Eindruck von der Benutzung des USB-2-X Moduls zu verschaffen. Im Anschluss werden in Kapitel 6 die Testergebnisse des Gesamtsystems und der USB Komponente dargestellt, damit in Kapitel 7 ein Fazit über die getane Arbeit und ein Ausblick auf zukünftige Arbeiten folgen kann. Im Anhang befinden sich Informationen über die verwendete Soft- und Hardware sowie eine Installationsanleitung und das verwendete Protokoll (vgl. A) für die Kommunikation über USB.

1.3. Aufgaben des Systems

Die Aufgabe der Software des USB-2-X Moduls ist es, Daten über USB von einem User Interface auf dem Host PC zu empfangen, um sie dann über eine ausgewählte Schnittstelle an die Steuereinheit eines Motors zu versenden oder aber Daten von der Motorsteuerung über ein Interface zu empfangen, um sie dann über USB an das User Interface auf dem PC zu senden.

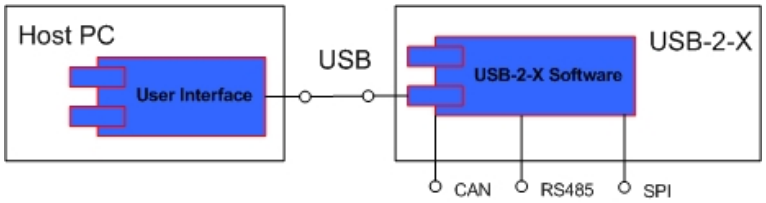

Abbildung 1.2.: Komponenten des Systems

Dabei muss das USB-2-X Modul die Daten, welche über USB verschickt werden, in ein für das jeweilige Interface geeignetes Format konvertieren, um sie dann an die Motorsteuerung zu senden, bzw. Daten, die von der Motorsteuerung empfangen werden, aufbereiten, um sie dann an das User Interface zu senden. Im Einzelnen fallen darunter folgende Funktionen:

- Initialisierung der verschiedenen Interfaces
- Verwaltung des Datenflusses
- Datenaustausch mit Peripheriegeräten über verschiedene Interfaces
- Fehlerüberprüfung

Das zum Testen des USB-2-X Moduls benötigte User Interface wird auch im Zuge dieser Arbeit entwickelt und hat die Aufgabe, Daten über USB an das USB-2-X Modul zu senden und vom USB-2-X Modul gesendete Daten über USB zu empfangen. Im Einzelnen fallen darunter folgende Funktionen:

- Zugriff auf den Gerätetreiber für das USB-2-X Modul
- Generierung von Nachrichten
- Datenaustausch mit dem USB-2-X Modul über USB

1.4. Einsatzgebiet

Die Firma TRINAMIC fertigt unterschiedliche Module mit unterschiedlichen Schnittstellen zum Ansteuern elektronisch kommutierter Motoren. Bevor diese Produkte die Firma verlassen, werden Tests durchgeführt, um ihre Funktionalität zu gewährleisten. Für die Funktionstests stehen Testprogramme zur Verfügung. Sie laufen auf einem PC und verschicken unterschiedliche Steuerbefehle an die Module über USB. Primäres

Einsatzgebiet des USB-2-X Moduls sind also Tests, welche i. A. auf den Laboreinsatz beschränkt sind, da keine galvanische Trennung[5] existiert.

Dabei wird zum Beispiel ein Modul mit CAN Interface an das USB-2-X Modul angeschlossen. Das USB-2-X Modul empfängt USB Nachrichten von einem User Interface und sendet sie über die CAN Schnittstelle zur Motorsteuerung. Die Nachricht selbst kann motorspezifische Steuerungsbefehle enthalten, wie zum Beispiel "Rotate Right" (ROR) oder auch "Get Axis Parameter" (GAP). Die Antwort der Motorsteuerung wird dann über das USB-2-X Modul an das User Interface auf dem PC gesendet, wo der Benutzer sie auswerten kann.

1.5. Marktanalyse

Recherchen haben ergeben, dass zwar viele Schnittstellenkonverter angeboten werden, sie jedoch entsprechend ihres Anwendungsbereiches eingeschränkt sind. In der Automobilindustrie wird zum Beispiel viel mit dem CAN Bus gearbeitet. Wird ein Fahrzeug gewartet, wird nun lediglich eine Umsetzung von CAN nach USB benötigt, um beispielsweise Fahrzeugdaten auf einem Laptop auszuwerten. Innerhalb der Firma TRINAMIC war die Entscheidung der Entwicklung des USB-2-X Moduls also sinnvoll, da die elektronischen Motoren über Module mit unterschiedlichen Interfaces angesprochen werden können. Kunden der Firma TRINAMIC haben somit die Möglichkeit, einen universellen Schnittstellenkonverter zu erwerben.

Die folgenden Links verschaffen einen kleinen Überblick über erhältliche Schnittstellenkonverter. Es wird schnell deutlich, dass universelle Schnittstellenkonverter eher eine Seltenheit sind.

- http://www.canusb.com/
- http://www.advantech.com/
- http://www.sysacom.com/
- http://www.ftdichip.com/
- http://www.vscom.de/

[5] Wenn es für Ladungsträger keinen Weg gibt, aus einem Stromkreis in einen anderen zu fließen, spricht man von *galvanischer Trennung* der beiden Stromkreise. Zwischen den Stromkreisen besteht dann keine elektrisch leitfähige Verbindung. Vgl. wikipedia.de (2008)

2. Grundlagen

In diesem Kapitel sollen die Grundlagen der beteiligten Komponenten des USB-2-X Moduls beschrieben werden, um dem Leser einen leichteren Einstieg in die weiterführenden Kapitel zu ermöglichen.

2.1. USB

Damit das Thema nicht den Rahmen dieser Arbeit überschreitet, beschränkt sich dieses Kapitel im Wesentlichen auf die Eigenschaften von USB, die das Entwicklungsboard auch anbietet und die in dieser Abschlussarbeit benötigt werden.

2.1.1. Einleitung

Um die Entwicklung von Treibern erheblich zu vereinfachen, wurde das USB Klassenkonzept[1] entwickelt. Es kategorisiert verschiedene USB Geräte ihren Eigenschaften nach in so genannte Klassen. Betriebssysteme können diese Geräte erkennen und automatisch Standardtreiber laden, die nur wissen, welche Form die Daten des USB Gerätes haben. In der Praxis bedeutet dies, dass man sich nur noch um die Firmware[2] des Gerätes kümmern muss, da die Entwicklung eines Treibers nicht notwendig ist. Im Wesentlichen unterscheidet man zwischen der Human Interface Device Class, Communication Device Class und Audio Device Class. Die Human Interface Device Class wird beispielsweise benutzt, um Tastaturen oder auch Mäuse anzusteuern, die Communication Device Class repräsentiert Daten von Kommunikationsschnittstellen und die Audio Device Class wird genutzt, um zum Beispiel Daten mit einer Soundkarte auszutauschen.

[1] Vgl. Sauter (2008)

[2] Als *Firmware* bezeichnet man Software, die auf einen Flash-Read Only Memory (ROM) oder Electrically Erasable Programmable ROM (EEPROM) eines Gerätes geschrieben wird, um dort elementare Aufgaben zu übernehmen.

Für den Datenaustausch unterstützt USB drei Geschwindigkeitsmodi:

1. High Speed - 480 Mbit/s
2. Full Speed - 12 Mbit/s
3. Low Speed - 1.5 Mbit/s

USB [3] ist ein Single Master Bus[4] und die Spezifikation selber beschreibt keine Eigenschaften für den Multimaster-Betrieb. Allerdings beschreibt die On-The-Go Spezifikation ein Protokoll, mit dem zwei Hosts die Rolle des Masters aushandeln können. Bei einer Datenübertragung ist der Master verantwortlich für die Verteilung der Bandbreite und die Koordination des Datenflusses über Token.
USB Geräte selbst sind miteinander über eine Sterntopologie verbunden, wobei ein einzelnes Gerät wiederum ein Hub sein kann und somit einen weiteren Stern bildet. An einen USB Port können bis zu 127 Geräte angeschlossen werden, so dass viele Hersteller bereits Hosts mit mehr als einem Port und Controller anbieten, damit selbst bei fünf und mehr Ports mehr Geräte ohne relevante Bandbreiteneinbußen angeschlossen werden können.
Die Host Controller selber haben ihre eigenen Spezifikationen. Für die USB 1.1 Spezifikation existieren zum einen das Universal Host Controller Interface (UHCI), welches von Intel entwickelt wurde und eher softwarelastig ist und somit kostengünstigere Controller erlaubt, zum anderen das Open Host Controller Interface (OHCI), welches von Compaq, Microsoft und National Semiconductor eher hardwarelastig entwickelt wurde und somit einfachere Software erlaubt. Mit der Entwicklung von USB 2.0 wurde das Enhanced Host Controller Interface (EHCI) von Intel, Compaq, NEC, Lucent und Microsoft eingeführt.
Eine letzte nennenswerte Eigenschaft von USB sind seine Transfermodi, auf die später (vgl. 2.1.10) genauer eingegangen wird.

- Control
- Interrupt
- Bulk
- Isochron

Jeder dieser Transfermodi bietet Eigenschaften für verschiedene Anwendungsfälle an.

[3] Vgl. Peacock (2007)
[4] *Single Master Bus* bedeutet, dass nur ein Master am Bus angeschlossen ist. Nur er kann jegliche Kommunikation initiieren. Kein Gerät kann selbstständig Daten übertragen. Der Master muss zyklisch alle Geräte nach neuen Daten abfragen. Vgl. Sauter (2008)

2.1.2. Hardware

USB benötigt zur physikalischen Datenübertragung vier Leitungen: D+, D-, VBus[5] und Ground (GND). Der Empfänger definiert eine binäre Eins, wenn D+ um 200 mV größer als D- ist und eine binäre Null, wenn D+ um 200 mV kleiner als D- ist. Eine Eins wird im Full Speed Modus sprachgebräuchlich dem Buchstaben J zugewiesen und eine Null dem Buchstaben K.

Tabelle 2.1.: USB - Buszustände

Zustand	D+	D-	Differentiell
J(idle)	HIGH	LOW	1
K(resume)	LOW	HIGH	0

2.1.3. Speed Identification

Bei Full Speed Geräten wird auf der Device Seite die D+ Leitung über einen Pull Up Widerstand[6] mit einer Spannungsquelle von 3.3 V verbunden. Der Pull Up Widerstand kann über einen Input Output (IO) Pin mittels eines p-Kanal Metal Oxide Semiconductor Field-Effect Transistor (MOSFET)[7] gesteuert werden. Damit das Device beim Anschluss als Full Speed Gerät erkannt werden kann, wird die D+ Leitung auf 3.3 V "gezogen", in dem der Pull Up Widerstand über den IO Pin aktiviert wird. Der Host erkennt das Device und provoziert nach 100 ms einen Busreset. Ist das Device nicht am Host angeschlossen, werden D+ und D- über zwei Pull Down Widerstände auf GND “gezogen“.

[5] Die *VBus Leitung* des USB Busses versorgt angeschlossene Geräte mit der benötigten Spannung.
[6] Ein *Pull Up Widerstand* wird verwendet, um eine Leitung auf einen definierten Wert zu “ziehen“. Vgl. roboternetz.de (2005)
[7] *p-Kanal MOSFETs* werden in elektronischen Schaltungen häufig als einfache Schalter verwendet.

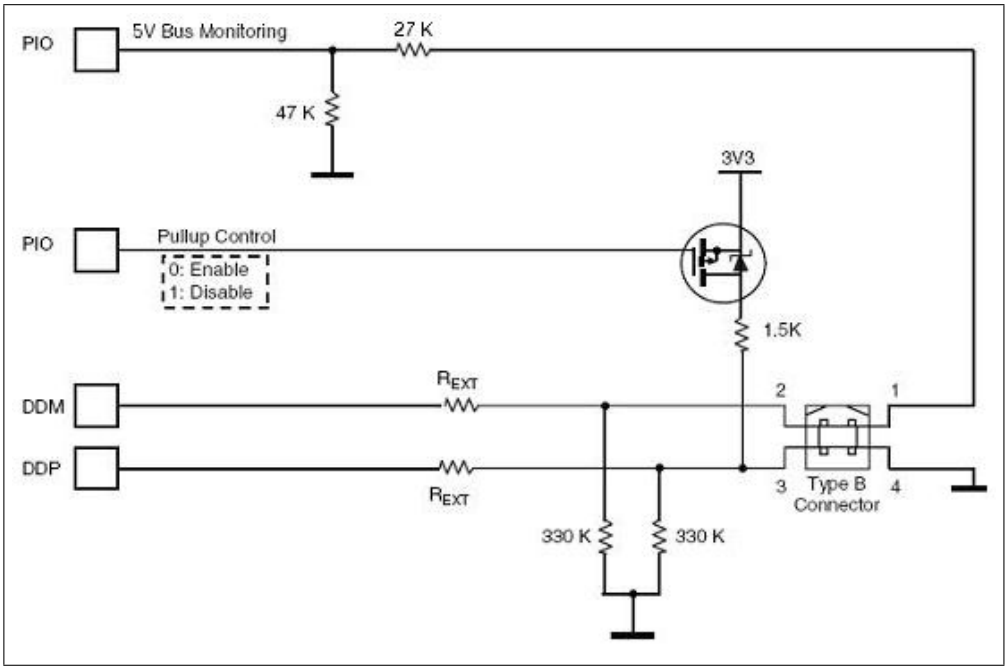

Abbildung 2.1.: USB - Geschwindigkeitserkennung

2.1.4. Datenkodierung

Um Daten möglichst fehlerfrei auf dem Bus zu übertragen, wird das Non Return to Zero Invert (NRZI) Verfahren verwendet. NRZI ist ein Verfahren, um Bitmuster auf einer physikalischen Leitung zu übertragen. Eine Null ist ein Wechsel zwischen J und K und eine Eins ein Ausbleiben dieses Wechsels.

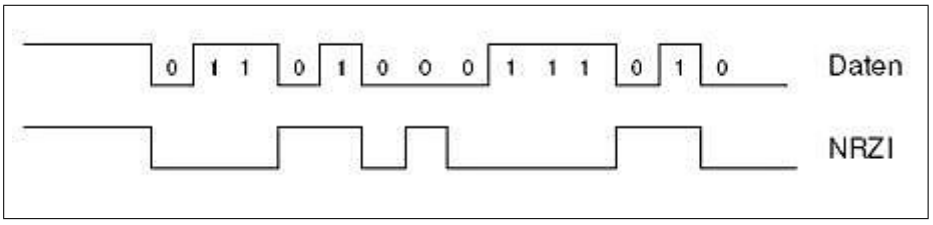

Abbildung 2.2.: USB - Datenkodierung

Jede Übertragung beginnt mit einem Sync Feld (vgl. 2.1.6), um Sender- und Empfängerclock zu synchronisieren. Da zu viele Einsen die richtige Dekodierung des differentiellen Signals verhindern, wird während der Übertragung sechs aufeinanderfolgenden Einsen eine Null zwischengefügt, um die Übertragung synchron zu halten. Dieses Verfahren nennt sich Bitstuffing.

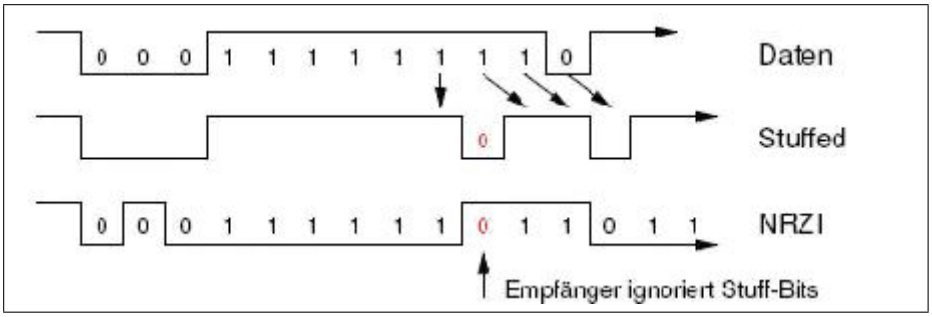

Abbildung 2.3.: USB - Bitstuffing

Auf der Gegenseite wird nun der Wechsel zwischen J und K wieder in Nullen und Einsen übersetzt und jede zusätzlich eingefügte Null ignoriert.

2.1.5. USB Protokoll

Jede USB Transaktion besteht aus folgenden Pakettypen:

1. Token Paket
2. Optionales Datenpaket
3. Statuspaket

Jede Transaktion wird dabei vom Host initiiert. Er bestimmt über das Token Paket was folgen wird und somit, ob es sich um eine Setuptransaktion handelt, ob es sich bei den Datenpaketen um eine Lese- oder Schreibtransaktion handelt und mit welchem Device und Endpunkt Daten ausgetauscht werden. Danach folgt ein Datenpaket und als letztes ein Statuspaket, welches signalisiert, ob das Token oder die Daten erfolgreich empfangen wurden oder ob der Endpunkt zum Zeitpunkt der Übertragung nicht verfügbar war.

2.1.6. USB Pakete

Auf dem USB Bus werden Daten mit dem Last Significant Bit (LSB) zuerst übermittelt. USB Pakete bestehen aus den folgenden Feldern:

- SYNC: Das Sync-Muster beinhaltet sieben Nullen gefolgt von einer Eins und wird bei der NRZI-Kodierung in KJKJKJKK (10101011) übersetzt. Es wird benutzt, um die Clock von Transmitter und Receiver zu synchronisieren.

- PID: Damit die Richtigkeit des Packet Identifiers überprüft werden kann, ist das PID Feld 8 Bit lang, wobei die letzten 4 Bit den ersten 4 Bit in invertierter Darstellung entsprechen. Der PID identifiziert den Pakettypen wie folgt:

Tabelle 2.2.: USB - PID Identification

Group	PID	Packet Identifier
Token	1	OUT Token
	1001	IN Token
	101	SOF Token
	1101	SETUP Token
Data	11	DATA0
	1011	DATA1
	111	DATA2
	1111	MDATA
Handshake	10	ACK Handshake
	1010	NAK Handshake
	1110	STALL Handshake
	110	No Response Yet (NYET)
Special	1100	PREamble
	1100	ERR
	1000	Split
	100	Ping

- ADDR: Ist die Geräteadresse eines USB Gerätes.
- ENDP: Ist die Endpunktadresse innerhalb eines Interfaces.
- CRC: Cyclic Redundancy Checks[8] werden bei Daten (16 Bit Prüfsumme) und den Token (5 Bit Prüfsumme) durchgeführt.
- EOP: End Of Packet ist das Ende eines Pakets.

[8] Die *zyklische Redundanzprüfung* ist eine Klasse von Verfahren zur Bestimmung eines Prüfwertes für Daten. Vgl. wikipedia.de (2008)

Das Tokenpaket

Das USB Protokoll unterscheidet zwischen den Tokenpakete IN, OUT, Start Of Frame (SOF) und SETUP. Sie haben bis auf das SOF Paket folgendes Format:

Tabelle 2.3.: USB - Form des Tokenpakets

SYNC	PID	ADDR	ENDP	CRC5	EOP

Das SOF Paket

Das SOF Paket wird bei Full Speed Geräten alle 1 ms $\pm$ 500 ns gesendet. Mit Hilfe des SOF Pakets unterteilt der Host Controller die verfügbare Zeit auf dem Bus in Frames. Innerhalb eines Frames kann er Pakete von jedem Typ verschicken. Zwischen zwei SOF Paketen können im Full Speed Modus 1023 Datenbyte übertragen werden. Da sich angeschlossene Geräte über das SOF Paket synchronisieren, müssen sie auch verschickt werden, wenn kein Datenverkehr stattfindet. Jedes SOF Token wird mit einer 11 Bit breiten Systemzeit (Time Stamp) markiert, um Zeitüberschreitungen (Timeouts) der angeschlossenen Geräte feststellen zu können.
SOF Pakete haben folgendes Format:

Tabelle 2.4.: USB - Die Form des SOF Pakets

SYNC	PID	Frame Number	CRC5	EOP

Die Datenpakete

Man unterscheidet im Full Speed Modus zwei Arten von Datenpaketen, DATA0 und DATA1, die beide bis zu 1023 Datenbyte übertragen können. Diese alternierenden Datenpakete unterstützen die Garantie der Datenreihenfolge.
Sie haben folgendes Format:

Tabelle 2.5.: USB - Die Form der Datenpakete

SYNC	PID	DATAX	CRC16	EOP

Die Handshakepakete

Es gibt drei verschiedene Arten von Handshakepaketen:

1. ACK signalisiert den erfolgreichen Empfang.
2. NAK signalisiert, dass vom Device momentan keine Daten empfangen oder gesendet werden können oder dass ein Fehler aufgetreten ist.
3. STALL signalisiert, dass das Device nur durch einen Eingriff vom Host in Betrieb genommen werden kann.

Die Handshakepakete haben folgendes Format:

Tabelle 2.6.: USB - Form der Handshakepakete

SYNC	PID	EOP

2.1.7. USB Hardware Funktionen

Die Datenübertragung mittels des USB Protokolls wird hardwareseitig wie folgt geregelt:

Für die Endpunkte gibt es Puffer, die typischerweise 8 Byte groß sind. Jeder dieser Puffer wird einem Endpunkt zugeordnet. Sendet der Host beispielsweise einen Device Descriptor Request (vgl. 2.1.12), stellt die USB Hardware über das SETUP Token fest, ob es an sich adressiert ist. Wenn dies der Fall ist, kopiert es den Inhalt des folgenden Datenpakets in den richtigen Endpunkt. Jede USB Device Port (UDP) Funktion wird also über die Geräteadresse und einen Endpunkt angesteuert. Im Anschluss sendet die UDP Funktion ein entsprechendes Statuspaket und generiert einen Interrupt für den entsprechenden Endpunkt. Die Software kann dann auf den Interrupt reagieren und den Request beantworten.

2.1.8. Endpunkte

Auf Softwareebene stellen die Endpunkte das Ende der Datenkanäle dar. Wenn der Host Daten sendet, landen sie im Endpunkt OUT Puffer des USB-2-X Moduls, nachdem er ein OUT Token gesendet hat. Dort können sie von der Software gelesen werden. Wenn das USB-2-X Modul Daten senden will, werden die Daten in den Endpunkt IN Puffer geschrieben. Schickt der Host nun ein IN Token zum betroffenen Endpunkt, werden die

Daten gesendet. Endpunkte können also als Interface zwischen Hardware und Software verstanden werden.

Alle Geräte besitzen den Endpunkt 0. Über ihn werden Control- und Statusrequests empfangen und beantwortet. Er ist der einzige Endpunkt, bei dem eine Datenübertragung in zwei Richtungen, also bidirektional, stattfinden kann.

2.1.9. Pipes

Während ein USB Gerät Daten über Endpunkte überträgt, sendet der Host sie über Pipes. Eine Pipe stellt eine logische Verbindung zwischen Endpunkt und Host dar.

USB definiert zwei Arten von Pipes:

1. Stream Pipes haben kein definiertes Format und eine vordefinierte Richtung. Sie unterstützen den Bulk-, Isochron- oder Interrupt Transfer und können vom Host oder Device kontrolliert werden.
2. Message Pipes haben ein vordefiniertes Format und werden vom Host kontrolliert. Sie dürfen in beide Richtungen Daten übertragen, unterstützen aber nur den Control Transfer über den Endpunkt 0.

2.1.10. Endpunkt- und Transferarten

Es gibt folgende Transferarten:

- Control Transfer
- Interrupt Transfer
- Isochron Transfer
- Bulk Transfer

Control Transfer

Der Control Transfer wird benutzt, um Device Informationen abzufragen und ist ein wesentlicher Bestandteil der USB Enumeration (vgl. 2.1.13). Die Paketlänge beträgt maximal 64 Byte. Ein Control Transfer besteht grundsätzlich aus drei Phasen, bei denen für die Datenübertragung eine Message Pipe verwendet wird.

1. Setup Phase
2. Optionale Datenphase
3. Statusphase

Setup Phase: In dieser Phase wird der Request gesendet. Das SETUP Token enthält die Geräteadresse und die Endpunkt Nummer. Als nächstes wird ein DATA0 Paket gesendet, welches den detaillierten Setup Request enthält. Das letzte Paket ist ein Quittungspaket. Wurden die Daten erfolgreich empfangen (Cyclic Redundancy Check (CRC) und Packet Identifier (PID) ok), wird ein Acknowledged (ACK) gesendet. Ist ein Fehler aufgetreten, werden die Daten ignoriert, weil die USB Funktionen nicht zwischen einem STALL oder Not Acknowledged (NAK) Paket als Antwort auf ein Setup Paket unterscheiden können.

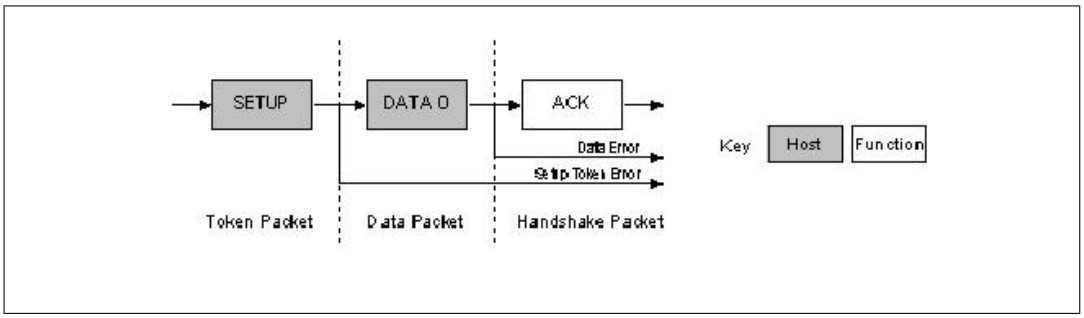

Abbildung 2.4.: USB - Setup Phase einer Setup Transaktion

Optionale Datenphase: Sie besteht aus einer oder mehreren IN oder OUT Transaktionen. Der Setup Request kennzeichnet in diesem Fall die Anzahl der Daten, die übertragen werden sollen. Wird die maximale Paketgröße überschritten, werden alle Pakete mit der maximalen Paketgröße übertragen.

IN: In diesem Fall erwartet der Host Control Daten vom Device und signalisiert dies durch ein IN Token. Tritt ein Fehler auf, wird das Paket vom Device ignoriert. War die Übertragung erfolgreich, kann das Device mit den angeforderten Control Daten antworten, ein STALL Paket senden, wenn der Endpunkt einen Fehler hat oder ein NAK Paket, wenn keine Daten vorhanden sind.

OUT: Wenn der Host Control Daten senden will, signalisiert er dies durch ein OUT Token, gefolgt von einem Datenpaket, welches die Control Daten enthält. Ist das OUT Token oder das Datenpaket fehlerhaft, wird es ignoriert. Wurde das Paket erfolgreich in den Puffer gelegt, folgt ein ACK Paket. Ist der Puffer voll, antwortet das Device mit einem NAK. Wenn der Endpunkt einen Fehler (z.B. halt bit set) aufweist, wird mit einem STALL Paket geantwortet.

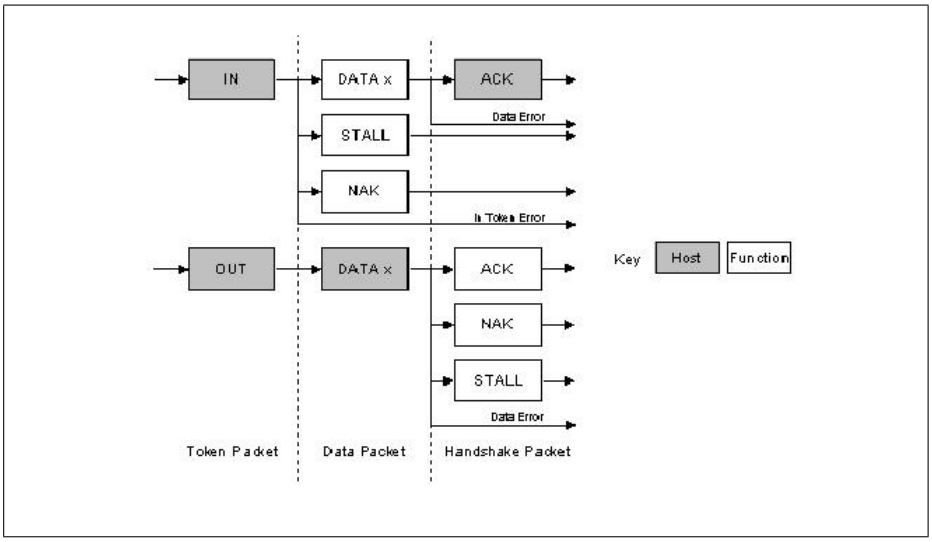

Abbildung 2.5.: USB - Data IN und OUT Transfer der Datenphase einer Setup Transaktion

Statusphase: Hier wird der Status der kompletten Übertragung gekennzeichnet. Hat der Host während der Datenphase ein IN Paket gesendet, um Daten zu empfangen, muss er den erfolgreichen Empfang quittieren. Dafür sendet er ein OUT Token gefolgt von einem Zero Length Paket. Das Device kann nun mit einem ACK die abgeschlossene Übertragung melden oder mit einem STALL kennzeichnen, dass ein Fehler aufgetreten ist. Ist der UDP des Devices gerade beschäftigt, signalisiert er dies mit einem NAK und fordert den Host auf den Vorgang zu wiederholen.

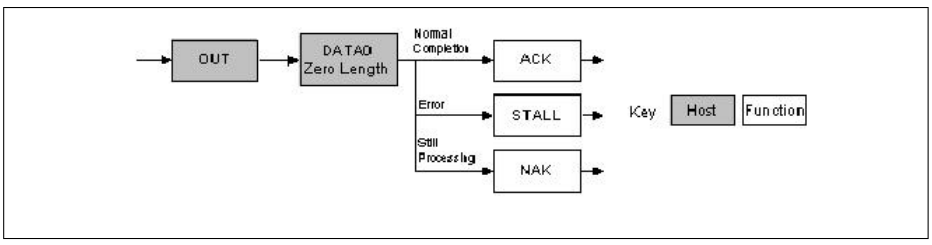

Abbildung 2.6.: USB - Data OUT Transfer der Statusphase einer Setup Transaktion

Hat der Host in der Datenphase ein OUT Token gesendet, kennzeichnet das USB Device den erfolgreichen Versand mit einem Zero Length Paket als Antwort auf ein IN Token. Ist ein Fehler aufgetreten, signalisiert das Device dies mit einem STALL. Ist das Device beschäftigt, wird ein NAK gesendet. Der Host wird damit aufgefordert, die Statusphase zu wiederholen.

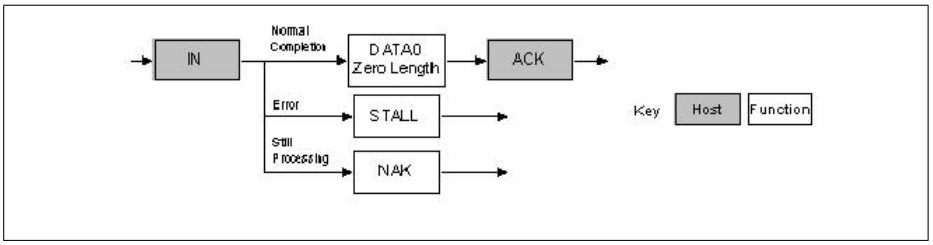

Abbildung 2.7.: USB - Data IN Transfer der Statusphase einer Setup Transaktion

Interrupt Transfer

Eigenschaften des Interrupt Transfers sind:

- Garantierte Latenzzeit
- Verbindung mit unidirektionaler Stream Pipe
- Die Daten stehen solange in der Warteschlange, bis der Host das USB Device pollt und nach Daten fragt.
- Fehlererkennung mit wiederholter Zusendung
- Interrupt Transfers haben bei Full Speed Geräten eine Paketlänge von maximal 64 Byte.

IN: Der Host fragt periodisch den Endpunkt ab. Die Pollingrate ist innerhalb des zugehörigen Endpunktdeskriptors festgelegt. Jede Abfrage beginnt mit einem IN Token. Ist das Token fehlerhaft, wird es vom Device ignoriert. Wenn Daten in der Warteschlange des Endpunktes liegen, wird als Antwort ein Datenpaket gesendet, woraufhin der Host den erfolgreichen Transfer mit einem ACK quittiert. Sind die Daten fehlerhaft, antwortet der Host nicht. Waren keine Daten in der Warteschlange als das IN Token gesendet wurde, antwortet das Device mit einem NAK. Tritt ein Fehler im Endpunkt auf, schickt das Device ein STALL Paket.

OUT: Will der Host dem Device Interrupt Daten senden, wird dies mit einem OUT Token, gefolgt von den Daten, signalisiert. Ist ein Teil dieser Daten fehlerhaft, werden beide Teile ignoriert. Wurden die Daten erfolgreich übertragen, antwortet das Device

mit einem ACK. Ist der Endpunkt Puffer noch nicht leer, antwortet das Device mit einem NAK. Tritt ein Fehler im Endpunkt auf, antwortet das Device mit einem STALL.

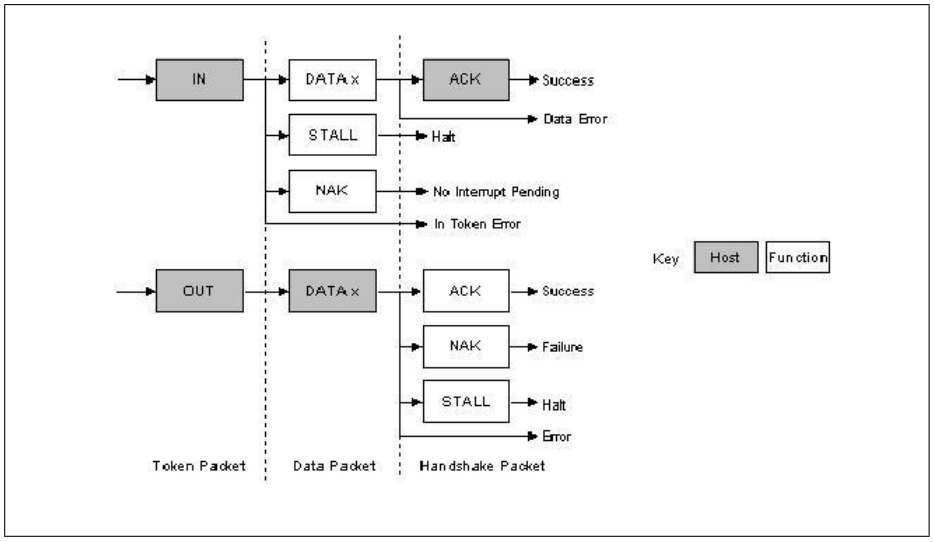

Abbildung 2.8.: USB - Data IN und Data OUT Transaktion des Interrupt Transfers

Isochron Transfer

Eigenschaften des Isochronen Transfers sind:

- Garantierte Bandbreite
- Begrenzte Latenzzeit
- Verbindung mit unidirektionaler Stream Pipe
- Fehlererkennung, aber keine wiederholte Zusendung
- Nur Full- und High Speed Modus

Die maximale Payloadlänge beträgt 1023 Byte für ein Full Speed Device, sollte aber nicht ausgenutzt werden, da dies die Bandbreite des Busses beeinträchtigen würde. Im Falle des verwendeten Entwicklungsboards beträgt die Payloadgröße 256 Byte. Das folgende Diagramm zeigt das Format von isochronen IN- und OUT Transaktionen.

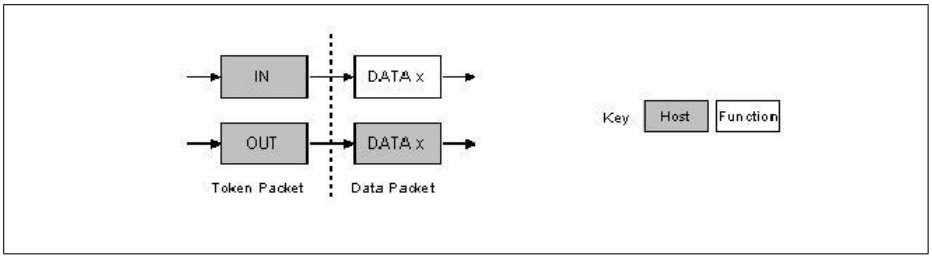

Abbildung 2.9.: USB - Data IN und Data OUT Transaktion des Isochronen Transfers

Bulk Transfer

Eigenschaften des Bulk Transfers sind:

- Nützlich zum Übertragen großer Datenmengen
- Fehlererkennung via CRC
- Garantierte Datenzustellung
- Keine Garantie der Bandbreite
- Verbindung mit unidirektionaler Stream Pipe
- Nur Full- und High Speed Modus

Für Full Speed Endpunkte ist die Paketgröße auf 8, 16, 32 oder 64 Byte beschränkt. Ein Bulk Transfer ist beendet, wenn er die angeforderte Datenmenge gesendet hat.

Das folgende Diagramm zeigt das Format von IN und OUT Transaktionen.

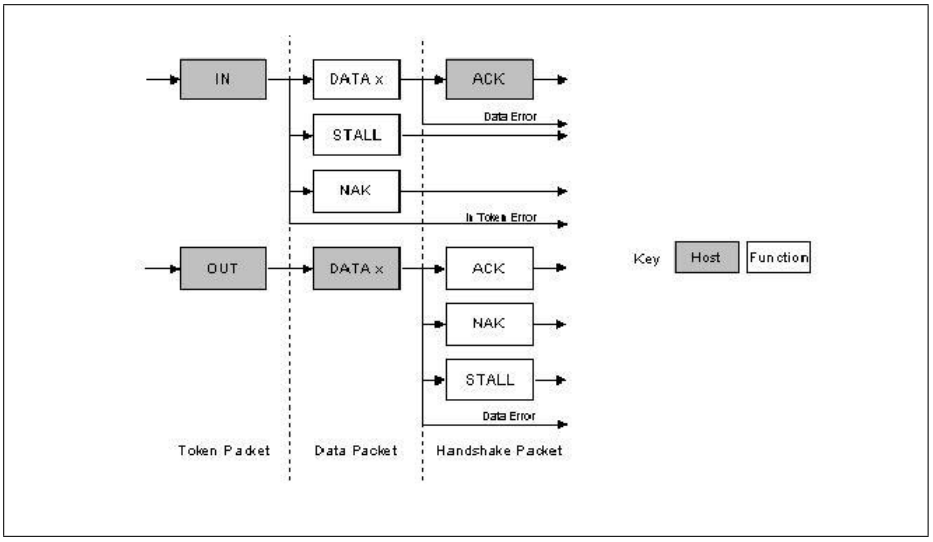

Abbildung 2.10.: USB - Data IN und Data OUT Transaktion des Bulk Transfers

2.1.11. Bandbreitenmanagement

Die Bandbreite des Busses wird wie folgt aufgeteilt: Bei Full Speed Geräten werden 10% der Bandbreite für den Control Transfer reserviert. Die anderen 90% stehen dem Isochron- und Interrupt Transfer zur Verfügung. Dem Bulk Transfer wird Bandbreite zugewiesen, wenn welche frei ist. Er hat die niedrigste Priorität beim Senden.

2.1.12. Deskriptoren

Die Informationen über das Device müssen in einem vordefinierten Format im Speicher des Devices stehen. Dieses Format nennt man Deskriptor. Im Wesentlichen unterscheidet man zwischen Gerätedeskriptoren, Konfigurationsdeskriptoren, Interface Deskriptoren, Endpunktdeskriptoren und String Deskriptoren.

Gerätedeskriptoren enthalten Informationen über das Gerät und Konfigurationsdeskriptoren über die verschiedenen Interfaces und Endpunkte. Der Gerätedeskriptor enthält beispielsweise die Parameter BDeviceClass, VendorID[9] und ProductID. Durch die VendorID weiß das Betriebssystem, welchen Treiber es laden soll. So ist echtes “Plug and

[9] *Die VendorID* wird vom USB Implementers Forum kostenpflichtig vergeben und identifiziert den Gerätetreiber auf der Host Seite eindeutig. Es gibt für Universitäten, Hobbyisten etc. kostenfreie Alternativen, allerdings nur für den nichtkommerziellen Einsatz.

Play“ möglich. Dieser Prozess auf der Seite des Host PCs nennt sich Enumeration und wird in Kapitel (2.1.13) genauer erläutert.

Der Konfigurationsdeskriptor enthält zum Beispiel Informationen zur Stromversorgung und die Anzahl der Interfaces. Wenn der Host bei der Enumeration den Gerätedeskriptor liest, kann er zwischen verschiedenen Konfigurationsdeskriptoren wählen. Schließt man zum Beispiel ein USB Gerät an einen Laptop an, wird die Konfiguration “Self-Powered“ gewählt. Letztendlich kann aber immer nur eine Konfiguration zur Zeit gewählt werden.

Die Interface Deskriptoren bündeln Endpunkte und fassen sie zu Funktionsgruppen zusammen. Besitzt man ein Multifunktionsgerät, wird jeweils ein Interface für die Faxfunktion, ein Interface für die Printerfunktion und ein Interface für die Scannerfunktion definiert. Im Gegensatz zum Konfigurationsdeskriptor ist die Anzahl der aktiven Interfaces nicht limitiert.

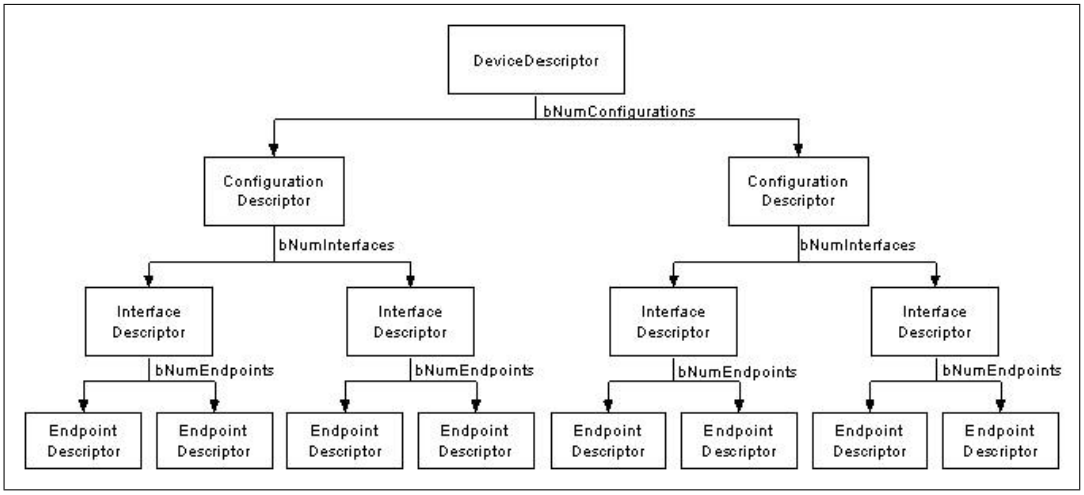

Abbildung 2.11.: USB - Deskriptor Hierarchie

Interface Deskriptoren bestehen unter anderem aus dem “bInterfaceNumber“ und dem “bAlternateSetting“ Feld. Diese beiden Felder erlauben die Auswahl verschiedener Interfaces zur Laufzeit. Durch einen “SetInterface“ Request kann von “Interface One“ auf “Interface One, bAlternateSetting = 1“ umgestiegen werden. Die Endpunktstruktur des alternativen Interfaces sollte genauso aussehen wie die des normalen Interfaces, mit dem einzigen Unterschied, dass als First In First Out (FIFO) Größe 0 Byte angegeben wird.
Findet nun gerade keine Kommunikation mit “Interface One“ statt, wählt der Host das alternative Interface aus und spart somit die Bandbreite, die für den Endpunkt innerhalb des normalen Interfaces reserveriert wurde. Vor dem nächsten Transfer wählt er nun

wieder “Interface One“ aus, um Bandbreite für den kommenden Transfer zu reservieren. So wird im Leerlauf die verfügbare Bandbreite für andere Geräte nicht eingeschränkt.

Endpunktdeskriptoren werden im Wesentlichen mit folgenden Parametern beschrieben:

- Die Endpunktadresse definiert die Adresse für den Endpunkt und enthält ebenfalls die Übertragungsrichtung für den Endpunkt. Befindet sich an Bit sieben eine Eins, so bedeutet dies, dass der Host von diesem Endpunkt lesen kann. Bei einer Null kann der Host in diesen Endpunkt schreiben.
- Die maximale Paketgröße wird durch die Tiefe des zugehörigen FIFO Speichers bestimmt.
- Die verwendete Transferart
- Das Polling Intervall bestimmt für die periodischen Transferarten, wie oft der Host Daten lesen oder senden muss.

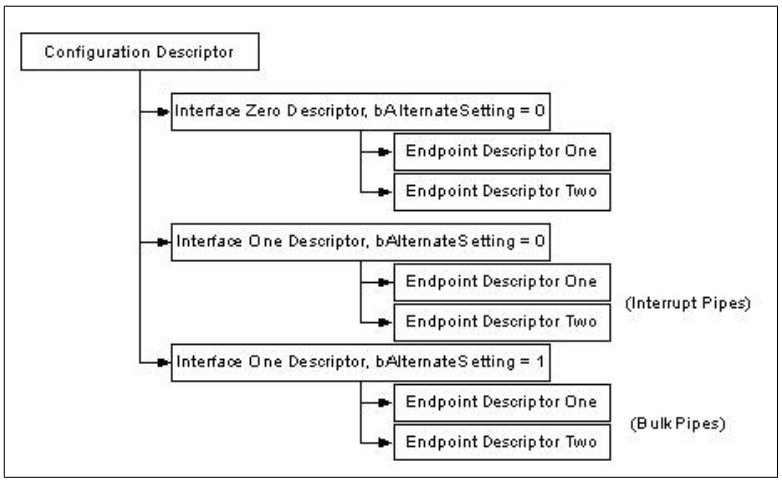

Abbildung 2.12.: USB - Interface Deskriptoren

2.1.13. Enumeration

Wenn ein Gerät angeschlossen wird, erkennt der Host dies bei Full Speed Geräten über die angelegte Spannung auf der D+ Leitung (vgl. 2.1.3). Dieser Zustand nennt sich “Attached“ und ist kein Enumerationszustand.

Es gibt vier Enumerationszustände:

1. Nach Anschluss eines Gerätes meldet der Root Hub das neu gefundene Gerät. Dieser Zustand nennt sich “Powered“.
2. Danach setzt der Host das Gerät zurück und weist ihm die Adresse 0 zu. Dieser Zustand nennt sich “Default“.
3. Über diese Adresse fragt er nach dem Gerätedeskriptor. Hat er die ersten 8 Byte empfangen, führt er einen Busreset durch. Anschließend weist er dem Gerät über ein “SetAddress“ Kommando eine feste Adresse zu. Danach fragt er über den Control Endpunkt nach dem kompletten Gerätedeskriptor. Im Anschluss fragt er nach dem Konfigurationsdeskriptor. Er befindet sich dann im Zustand “Addressed“.
4. Diese Informationen sind einem bestimmten Treiber zugewiesen, der dann geladen werden kann. Danach fragt der Host alle Deskriptoren noch einmal ab und sendet einen “SetConfiguration“ Request. Dieser Zustand nennt sich “Configured“.

Aus jedem dieser Zustände gelangt man in den Zustand “Suspended“, wenn der Bus inaktiv ist. Ist der Bus aktiv, gelangt man von dem Zustand “Suspended“ in den vorherigen Zustand. Nach einem Bus Reset befindet sich das Gerät im Zustand “Default“. Unterbricht man die Stromversorgung für einen kurzen Zeitraum, fällt man in den Zustand “Powered“ zurück.

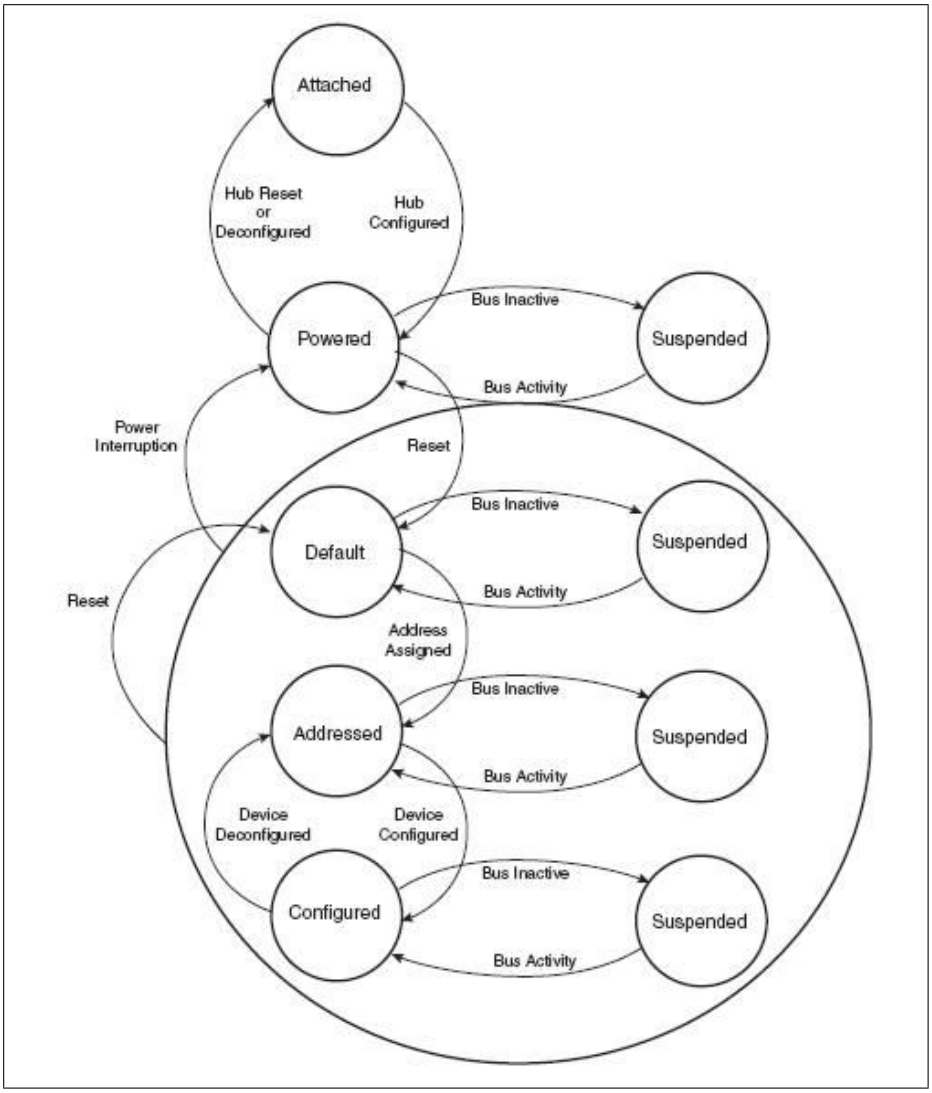

Abbildung 2.13.: USB - Enumeration

2.2. CAN

2.2.1. Einleitung

Der CAN Bus[10] wurde durch die Firma Bosch spezifiziert. Ursprünglich für die Anwendung in Kraftfahrzeugen entwickelt, hat er sich auch in Bereichen der Anlagenautomatisierung, zum Beispiel für die Kommunikation von Einheiten untereinander, durchgesetzt. Die CAN Spezifikation umfasst dabei nur die physikalische Ebene, die Bittransferkodierung sowie die Übermittlung von Telegrammen und die Fehlerbehandlung.

2.2.2. Eigenschaften

Der CAN Bus wurde entwickelt, um zeitkritische Informationen zu übertragen, d.h., er muss Nachrichten mit sehr geringen Latenz- und Zykluszeiten übermitteln. Das ist zum Beispiel in einem Klasse C Netzwerk erforderlich, um die Stabilität eines Motors zu kontrollieren. Eine weitere Anforderung an den CAN Bus, die sich aus Sicherheitsgründen ergibt, ist seine Multimaster-Fähigkeit. Jeder Knoten innerhalb des Netzes kann eine Datenübertragung einleiten, so dass bei einem Ausfall nicht die Funktion des Gesamtsystems gefährdet ist. Ein weiterer Vorteil ist, dass Daten mittels Broadcast[11] übertragen werden und so eine ereignisgesteuerte Kommunikation möglich ist. Für die Übertragung von Daten stehen Baudraten bis zu 1 Mbit zur Verfügung, woraus sich bei einer Auslastung von 50% und einem Protokolloverhead von 50% in Multimaster-Systemen eine Nutzdatenrate von ca. 25 kB/s ergibt. Um den Datenverkehr zu koordinieren, wird ein spezielles Zugriffsverfahren namens Carrier Sense, Multiple Access/Collision Detection + Collision Resolution (CSMA/CD + CR) verwendet. Eine Fehlererkennung existiert ebenfalls. Auf der Nachrichtenebene wurde eine 15 Bit CRC Prüfsumme implementiert. Auf physikalischer Ebene ist eine Fehlererkennung über Differenzsignale möglich.

2.2.3. Standard CAN Frame

Für den inhaltlichen Aufbau einer CAN Nachricht existieren momentan zwei Spezifikationen, die sich hauptsächlich in der Anzahl der Identifierbits unterscheiden, der

[10] Vgl. Wedemeyer (1999); Etschberger (1994)

[11] Bei einem *Broadcast* werden Datenpakete von einem Punkt aus an alle Teilnehmer eines Netzes übertragen.

Standard CAN Frame und der Extended CAN Frame. Beim Standard CAN Frame hat der Identifier[12] 11 Bit und beim Extended CAN Frame 29 Bit. Wählt man den Extended Identifier, kann man also 536.870.912 verschiedene Informationen filtern, beim Standard Format sind es nur 2048.

Tabelle 2.7.: CAN - Standard Frame

Start Bit	Identifier	RTR	IDE	R0	DLC	Data	CRC	ACK	EOP	IFS
1 Bit	11 Bit	1 Bit	1 Bit	1 Bit	4 Bit	0...8 Bit	15 Bit	2 Bit	7 Bit	3 Bit

Die übertragenen Bits auf dem CAN Bus können dominant (LOW Pegel) oder rezessiv (HIGH Pegel) sein. Der Identifier spiegelt gleichzeitig die Priorität der Nachricht wieder. Je kleiner der Identifier desto höher seine Priorität. Bei der Übertragung des Identifiers wird das Most Significant Bit (MSB) zuerst übertragen. Jede Übertragung einer Nachricht wird durch ein dominantes Startbit eingeleitet, durch dessen fallende Flanke sich die Teilnehmer synchronisieren können. Es folgt der 11 Bit Identifier. Anschließend wird das Remote Transmission Request (RTR) Bit übertragen. Ist es dominant, enthält die Nachricht keine Daten und die Antwort wird direkt im Anschluss übertragen, wenn vorher nicht eine Nachricht mit höherer Priorität übertragen wird. Es wird also genutzt, um ein Device anzustoßen, Informationen zu versenden.

Das folgende Kontrollfeld besteht aus 6 Bit. Im Falle des Standard Frames hat das erste Bit immer einen LOW Pegel, da mit dem Identifier Extension (IDE) Bit zwischen Standard- und Extended Identifier unterschieden wird. Das folgende Bit ist für zukünftige Erweiterungen reserviert. Im Anschluss folgen die 4 Bit des Data Length Code (DLC). Er gibt die Länge der zu übertragenen Datenbytes an. Maximal 8 Datenbyte können übertragen werden. Sie folgen direkt dem DLC Feld.

Damit Übertragungsfehler erkannt werden können, folgt eine 15 Bit lange Prüfsumme. Zur Bestätigung einer fehlerfreien Übertragung stehen 2 ACK Bit zur Verfügung. Damit dies funktioniert, legt der Sender einen rezessiven Pegel auf den Bus und erwartet, dass mindestens ein Teilnehmer ihn mit einem dominanten Pegel überschreibt. Beendet wird ein Frame durch 7 rezessive Bit, welche die Bitstuffingkodierung verletzen. Sie tragen den Namen End Of Packet (EOP). Als letztes folgen nochmal 3 rezessive Bit, mit denen sichergestellt werden soll, dass jeder Busteilnehmer genug Zeit hat die komplette Nachricht zu verarbeiten. Dieser Zeitraum nennt sich Inter Frame Space (IFS).

[12] Der *Identifier* eines CAN Telegramms wird benutzt, um den Adressaten einer Information festzustellen.

2.2.4. Unterschiede des Extended Frame

Tabelle 2.8.: Extended CAN Frame

Start Bit	Identifier	SRR	IDE	Identifier	RTR	R1	R0	DLC	...	IFS
1 Bit	11 Bit	1 Bit	1 Bit	18 Bit	1 Bit	1 Bit	1 Bit	4 Bit	...	3 Bit

Führt das IDE Bit einen rezessiven Pegel, hat das RTR Bit keine Bedeutung und bekommt den Namen Substitute Remote Request (SRR). Nach ihm folgt das IDE Bit und im Anschluss die restlichen 18 Bit des Extended CAN Identifiers, das RTR Bit und das 6 Bit lange Kontrollfeld. Die ersten beiden Bits des Kontrollfeldes sind für zukünftige Entwicklungen reserviert. Der Rest des Frames gleicht dem Standard Frame.

2.2.5. Arbitrierung

Die Arbitrierung[13] ist durch die Funktionalität der Bustreiber gegeben. Sie können einen rezessiven Pegel mit einem dominanten überschreiben, indem sie, wenn der Bus frei ist, die zu sendenden Bits, den Identifier zuerst, auf den Bus legen und im Anschluss den Buszustand wieder einlesen und die Werte vergleichen. Wird dabei festgestellt, dass ein rezessiver Pegel von einem dominanten Pegel überschrieben wurde, stellt der Busteilnehmer mit dem rezessiven Pegel die Übertragung sofort ein und wartet wieder bis der Bus frei ist. So ist sichergestellt, dass immer die Nachricht mit der höchsten Priorität gesendet werden kann. Da die Übertragung beim Überschreiben eines rezessiven Pegels mit einem dominanten Pegels sofort unterbrochen wird, muss der Busteilnehmer, der den dominanten Pegel gesendet hat, seine Daten nicht erneut senden. Das Verfahren erkennt Kollisionen auf dem Bus und ist dabei zerstörungsfrei.

2.3. SPI

2.3.1. Einleitung

SPI [14] beschreibt einen synchronen seriellen Bus. Dieser wurde von Motorola entwickelt, jedoch nie in einen vollständigen Standard oder eine Norm überführt. Außerdem wurde seitens Motorola keine Definition über ein Softwareprotokoll gemacht, sondern nur die

[13] *Arbitrierung* beschreibt den Vorgang der Buszuteilung.
[14] Vgl. Büch (2006); Pont (2001)

reine Hardwarefunktionsweise beschrieben. Auch wurde SPI nie mit Patenten belegt und ist somit lizenzfrei. Das hat dem Bussystem, neben der einfachen Implementierung, eine weite Verbreitung verschafft.

2.3.2. Eigenschaften

Technisch gesehen besteht die SPI Schnittstelle aus vier Leitungen (zwei Datenleitungen, einer Clockleitung und einer Chip Select Leitung) plus GND. Die Datenleitungen bezeichnet man als Master Out Slave In (MOSI) und Master In Slave Out (MISO) .

Schreibt man ein Datenwort in ein Register, wird automatisch die Datenübertragung eingeleitet. Der Master sendet Nachrichten an den Slave, welcher wiederum die empfangenen Daten simultan zurücksendet. Für das Senden und Empfangen wird jeweils ein 8 Bit Schieberegister genutzt, für die der Master den Takt generiert. Der Slave selber sendet also nichts, wenn er nichts empfängt. Theoretisch ist es auch möglich Daten an mehrere Slaves zu senden, indem man sie über ihre Chip Select Leitung selektiert. Auch dann kann der Master nur von einem Slave Daten gleichzeitig empfangen. Man spricht im Falle des SPIs also von einem Single Master-Multi Slave Interface.

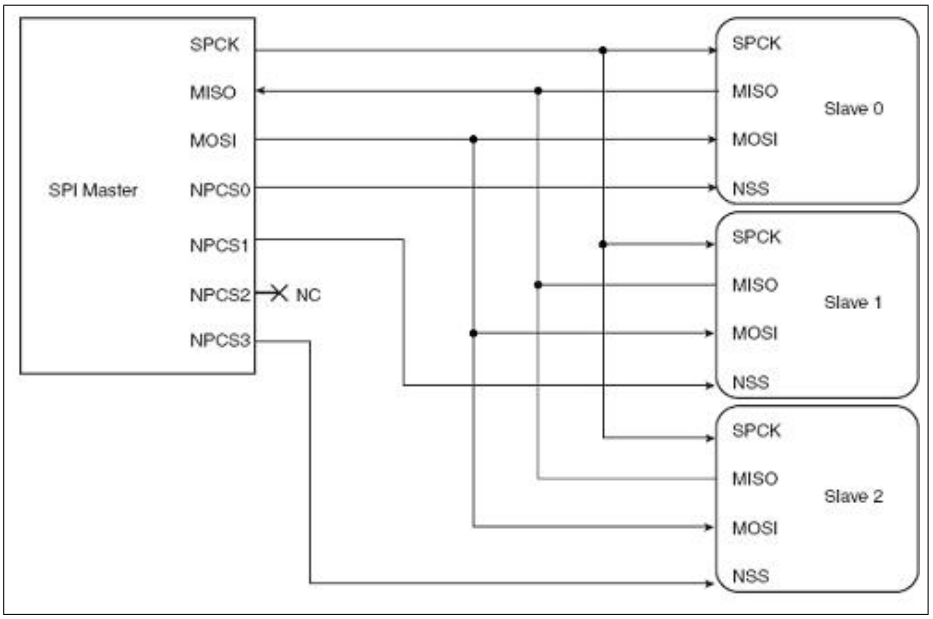

Abbildung 2.14.: SPI - Blockdiagramm Single Master-Multi Slave Implementation

2.4. RS485

2.4.1. Einleitung

Die RS485[15] Schnittstelle ist für die serielle Hochgeschwindigkeits-Datenübertragung über große Entfernungen entwickelt worden und findet im industriellen Bereich zunehmend Verwendung. Die RS485-Norm definiert lediglich die elektrischen Spezifikationen für Differenzempfänger und -sender in digitalen Bussystemen. Die ISO-Norm 8482 standardisiert darüber hinaus zusätzlich die Verkabelungstopologie mit einer max. Länge von 500 m.

2.4.2. Physikalisches Übertragungsverfahren

Die seriellen Daten werden, ohne Massebezug, als Spannungsdifferenz zwischen zwei korrespondierenden Leitungen übertragen. Für jedes zu übertragende Signal existiert ein Adernpaar, das aus einer invertierten und einer nicht invertierten Signalleitung besteht. Die invertierte Leitung wird durch den Index “A“ oder “+“ gekennzeichnet, die nicht invertierte Leitung mit “B“ oder “-“. Der Empfänger wertet nur die Differenz zwischen beiden Leitungen aus, so dass Störungen auf der Übertragungsleitung nicht zu einer Verfälschung des Signals führen. RS485-Sender stellen unter Last Ausgangspegel von ± 2 V zwischen den beiden Ausgängen zur Verfügung. Die Empfängerbausteine erkennen Pegel von ± 200 mV noch als gültiges Signal.

2.4.3. Eigenschaften

RS485 ist als bidirektionales Bussystem mit bis zu 32 Teilnehmern konzipiert. Ein RS485 Bus kann sowohl als 2-Draht- als auch als 4-Draht-System aufgebaut werden. Der Vorteil der 2-Draht-Technik liegt im Wesentlichen in der Multimaster-Fähigkeit, wobei jeder Teilnehmer prinzipiell mit jedem anderen Teilnehmer Daten austauschen kann. Da mehrere Sender auf einer gemeinsamen Leitung arbeiten, muss durch ein Protokoll sichergestellt werden, dass zu jedem Zeitpunkt maximal ein Datensender aktiv ist. Alle anderen Sender müssen sich zu dieser Zeit im inaktiven Zustand befinden.

[15] Vgl. Flik (2005)

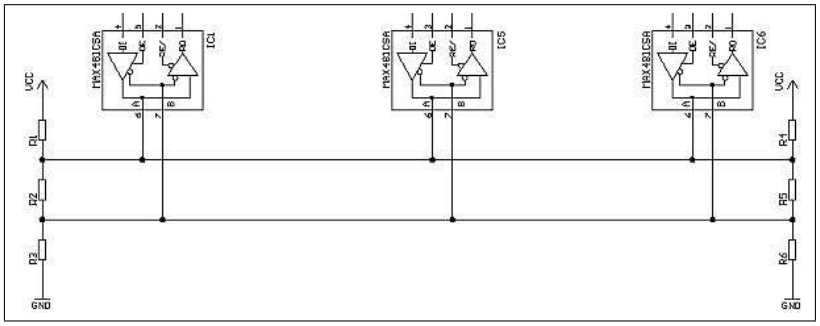

Abbildung 2.15.: RS485 - Schaltplan des 2-Draht-Busses

2.4.4. Terminierung

Ein Abschluss des Kabels mit Terminierungsnetzwerken[16] ist bei RS485-Verbindungen grundsätzlich erforderlich, um in den Zeiten, in denen kein Datensender aktiv ist, auf dem Bussystem einen definierten Ruhepegel zu erzwingen.

2.4.5. Protokoll

Wie bereits erwähnt ist bei RS485 kein festes Protokoll definiert, deshalb wird hier das am häufigsten verwendete erklärt. Das hier beschriebene Protokoll ist nur eine von vielen Möglichkeiten und kann den entsprechenden Bedürfnissen angepasst werden. Ein einfaches Protokoll für die Kommunikation enthält drei bis vier Komponenten und wird als Frame bezeichnet.

Tabelle 2.9.: RS485 - Protokoll

Adresse	Framelänge	Daten	Prüfsumme

Als Erstes wird eine Adresse übertragen. In kleinen Systemen ist diese meist 8 Bit lang und in der Regel ausreichend, da die meisten Treiber weniger als 256 Geräte treiben können. Darauf folgt die Länge des Frames, die entweder 8 oder 16 Bit beträgt. Werden für die Framelänge 8 Bit definiert, können 0-255 Datenbyte übertragen werden. Danach folgen die eigentlichen Daten. Es muss immer die Anzahl an Daten übertragen werden, die durch die Länge definiert wurde. Nach den Daten kann optional noch eine

[16] Um auf dem Bussystem den Ruhepegel zu erzwingen, kann man über ein *Terminierungsnetzwerk* eine Leitung über 1 kΩ auf Masse und die andere Leitung über 1 kΩ auf V_{cc} legen. Vgl. roboternetz.de (2005)

Prüfsumme folgen. Diese ermöglicht es, Übertragungsfehler festzustellen. Als Prüfsumme kann man z.B. 16 Bit CRC verwenden.

3. Analyse

Ziel der Analyse ist es, die Komponentenhierarchie auf die Software zu reduzieren und ihre Aufgaben theoretisch zu beschreiben, um anschließend ihr Design zu diskutieren und sie dann im Kapitel Implementierung in Quellcode zu übersetzen.

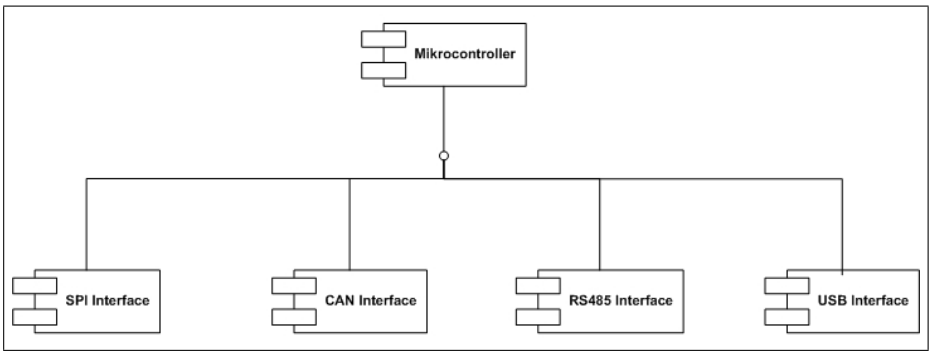

Abbildung 3.1.: Analyse - Beteiligte Komponenten

3.1. Funktionale Anforderungen

3.1.1. Hauptprogramm

Hardwarekonfiguration

Damit nach dem Systemstart Nachrichten über USB empfangen werden können, muss das Hauptprogramm dafür sorgen, dass die USB Schnittstelle zu diesem Zeitpunkt funktionstüchtig ist. Die dafür notwendige Funktion wird im Hauptprogramm aufgerufen und umfasst beispielsweise das Aktivieren des Pull Up Widerstandes, welcher für die Speed Identification (vgl. 2.1.3) benötigt wird oder aber die Aktivierung der Clock für den UDP. Um während der Entwicklungsphase den Programmablauf zu verfolgen, ohne das Programm zu unterbrechen, wird die Debug Unit (DBGU) des Entwicklungsboards konfiguriert. Sie ermöglicht Debug-Ausgaben über den Universal Synchronous Asynchronous Receiver Transmitter (USART). So ist es möglich, Statusmeldungen

über den USART auszugeben, ohne beispielsweise den USB Enumerationsprozess zu unterbrechen, wenn man den Debugger startet. Dies ist erforderlich, da Vorgänge wie die Enumeration strenge Timing Parameter einhalten müssen.

Datenflussverwaltung

Empfängt das USB-2-X Modul Nachrichten über USB, ist die Aufgabe des Hauptprogramms, die Nachricht gemäß USB-2-X Protokoll (vgl. A) auszuwerten und die weiterzuleitenden Daten aufzubereiten und an die geforderte Schnittstelle zu übergeben oder aber Daten von einer bestimmten Schnittstelle zu lesen, um sie dann über USB an das User Interface auf dem Host PC zu schicken.

Fehlerüberwachung

Über USB ankommende Nachrichten werden durch eine geeignete Methode auf Fehler überprüft. Treten Fehler auf, wird dies dem Host mitgeteilt, damit die Zustellung der Daten vom Benutzer wiederholt werden kann. Verwendet man die Transferart Bulk, ist dies für die Verbindung über USB nicht nötig, da eine fehlerfreie Übertragung garantiert wird (vgl. 2.1.10).

3.1.2. USB

Bearbeitung von Setup Requests

Setup Requests werden an das USB-2-X Modul gesendet, um Konfigurationseinstellungen zu ändern oder einfach abzufragen. Dazu gehören zum Beispiel Anfragen wie "GetConfiguration", um den Konfigurationsdeskriptor zu erhalten. Sie werden über den Control Endpunkt empfangen und sind Teil des Enumerationsprozesses, können aber auch gesondert empfangen werden. Die USB Komponente muss derartige Anfragen beantworten und entsprechend reagieren können.

Enumerationsprozess

Durch den Enumerationsprozess teilt das USB-2-X Modul dem Host PC, wie in Kapitel (2.1.13) beschrieben, seine Eigenschaften mit. Für den Enumerationsprozess gibt es gemäß USB Spezifikation einen Satz Standardanfragen, welche die USB Komponente

in jedem Fall beantworten können muss, um dem Host PC seine Eigenschaften mitzuteilen. War der Enumerationsprozess erfolgreich, ist die USB Schnittstelle des USB-2-X Moduls betriebsbereit.

3.1.3. CAN, SPI, USB und USART

Da sich die Aufgaben dieser Schnittstellen unwesentlich voneinander unterscheiden, werden sie hier zusammenfassend dargestellt. Die feinen Unterschiede werden in den nächsten beiden Kapiteln dargestellt.

Konfiguration der Schnittstellen

Damit Nachrichten über die Schnittstellen ausgetauscht werden können, müssen sie korrekt konfiguriert werden. Dabei muss die Hardware, z.B. die Clock, für die jeweilige Schnittstelle sowie die Schnittstelle selber aktiviert werden, außerdem muss festgelegt werden, mit welcher Geschwindigkeit Daten versendet werden und letztendlich, wie auf synchrone und asynchrone Ereignisse reagiert werden soll.

Überwachung der Statusregister

Damit eindeutig geregelt ist, wann und wo Daten versendet bzw. empfangen werden sollen, müssen die Statusregister der Schnittstellen entsprechend der Konfigurationseinstellungen überwacht und geeignete Mechanismen zur Verfügung gestellt werden, um den Datenfluss zwischen zwei Schnittstellen über die Kontrollregister zu regeln.

Nachrichtenempfang und -versand

Treffen Daten über einen Bus ein, werden sie in einem bestimmten Empfangsregister gespeichert und müssen anderen Programmteilen über Puffer zur Verfügung gestellt werden. Sollen Daten über eine Schnittstelle verschickt werden, müssen sie aus einem Puffer in ein Senderegister geschrieben werden.

3.2. Nichtfunktionale Anforderungen

1. Die Komponenten sollen einen hohen Grad an Wiederverwendbarkeit aufweisen, damit sie in anderen Projekten ohne großen Aufwand eingesetzt werden können.
2. Der Random Access Memory (RAM) Verbrauch soll im Rahmen der Machbarkeit niedrig sein.
3. Um die Wartbarkeit zu erleichtern, soll der Code verständlich, gut dokumentiert, sowie leicht testbar und änderbar sein.
4. Entscheidungen liegt stets ein Kompromiss zwischen Einfachheit und Umfang zu Grunde, um die Komplexität der Software nicht unnötig zu steigern.
5. Alle Quelltexte sind in der Programmiersprache C geschrieben.

4. Design

Um dem Leser die Designentscheidung verständlich zu machen, wird der Nachrichtenverlauf des USB-2-X Protokolls (vgl. A) zunächst durch ein Sequenzdiagramm[1] verdeutlicht. Im Anschluss werden die Möglichkeiten der Designentscheidung aufgezeigt und gegeneinander abgewogen.

4.1. Nachrichtenverlauf

Das Sequenzdiagramm zeigt den zeitlichen Ablauf der Zugriffe innerhalb des Gesamtsystems. Wie bereits erwähnt, muss das USB-2-X Modul Nachrichten in zwei Richtungen weiterleiten. Um ein Interface des USB-2-X Moduls zu nutzen, muss es vorher konfiguriert werden. Hierfür wird eine Init Message von dem User Interface (vgl. 5.7) auf dem Host PC an das USB-2-X Modul verschickt. Nachrichten, die benutzt werden, um Daten an einen Motor zu senden, werden als Write Message bezeichnet.

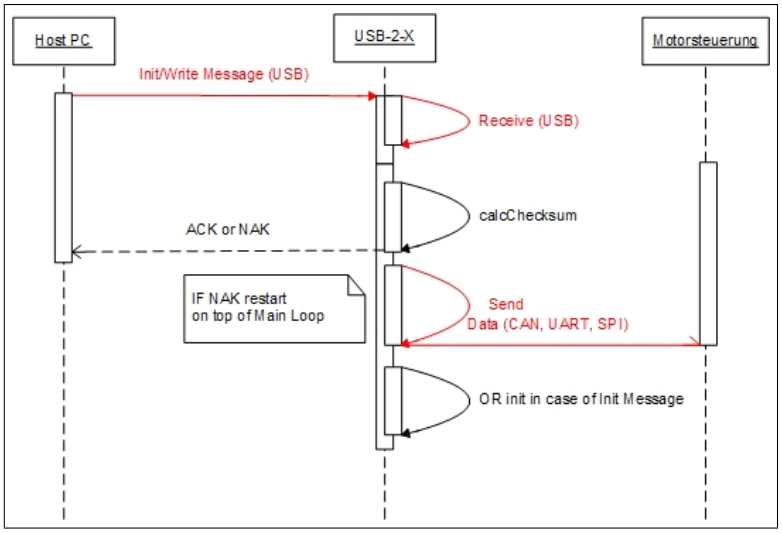

Abbildung 4.1.: Sequenzdiagramm einer Init- bzw. Write Message

[1] Vgl. Fowler (2003)

Um empfangene Daten der Motorsteuerung an den PC zu senden, wird von dem User Interface auf dem Host PC eine Read Message an das USB-2-X Modul verschickt. Sie ist der Auslöser für das USB-2-X Modul, die empfangenen Daten der Motorsteuerung an das User Interface zu schicken.

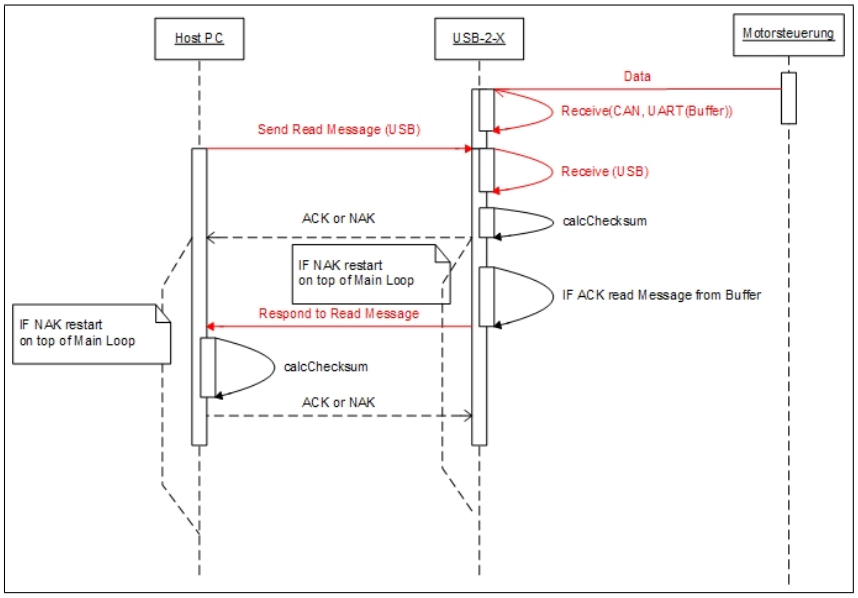

Abbildung 4.2.: Sequenzdiagramm einer Read Message

4.2. Designalternativen

Ereignisse treten bezüglich des USB-2-X Moduls in Form von Nachrichten auf (vgl. 4.1). Um auf diese reagieren zu können, gibt es im Wesentlichen zwei Verfahren, nämlich Polling und Interrupts[2], deren Vor- und Nachteile folgend dargestellt werden.

[2]Vgl. Pont (2001)

Tabelle 4.1.: Design - Entscheidungsmöglichkeiten

	Polling	Interrupts
Vorteile	Einfach zu implementieren. Weniger Hardwareaufwand nötig. Nach der Erkennung eines Ereignisses kann sofort reagiert werden	Das Auftreten eines Ereignisses wird immer überwacht. Kommunikationsleitungen werden entlastet. Es wird Strom gespart, da der Prozessor bis zum nächsten Auftreten eines Ereignisses in den Sleep Mode versetzt werden kann.
Nachteile	Das Auftreten eines Ereignisses wird nur zu gefragten Zeiten überwacht. Es gibt keine Garantie dafür, dass zum Pollingzeitpunkt auch ein Ereignis stattfindet. Es wird demnach auch Rechenzeit verschwendet und Kommunikationsleitungen unnötig belastet.	Mehr Hardwareaufwand nötig. Der Zeitraum zum Abarbeiten eines Ereignisses ist größer als der beim Polling, da bei der Interruptverarbeitung der aktuelle Befehl des Programms noch fertig ausgeführt wird, bevor die Rücksprungadresse gesichert und in die ISR verzweigt wird. Diese wiederum muss die zu benutzenden Register auf dem Stack sichern, bevor mit der eigentlichen Reaktion auf das externe Ereignis begonnen werden kann.

4.3. Designentscheidung

Um dem Leser die Designentscheidung zu verdeutlichen, wird sie für jede Komponente nach folgendem Schema dargestellt:

1. Welche Anforderungen werden bezüglich des Designs an die Komponente gestellt?
2. Begründete Wahl der Designentscheidung hinsichtlich der möglichen Alternativen zur Lösung des Problems.

4.3.1. CAN

Anforderungen bezüglich des Designs:

1. Der Nachrichtenversand wird von der Software ausgelöst und ist deshalb synchron. Er findet aus Sicht der Software zu einem bestimmten Zeitpunkt statt.
2. Der Nachrichtenempfang ist asynchron und findet aus Sicht der Software zu einem unbestimmten Zeitpunkt statt.
3. Während der Abarbeitung einer empfangenen Nachricht treten keine weiteren Ereignisse auf. Das USB-2-X Modul ist über die CAN Schnittstelle immer mit genau einem Steuermodul eines Motors verbunden. Es existiert immer genau eine Instanz, von der Ereignisse ausgelöst werden, deshalb müssen keine Prioritäten vergeben werden.

Begründete Wahl:

Da der Nachrichtenempfang asynchron erfolgt, wird die CAN Komponente mit Interrupts realisiert. So wird keine Rechenzeit verschwendet und die Kommunikationleitungen werden entlastet, da nicht ständig abgefragt werden muss, ob eine Nachricht empfangen wurde. Für den Fall, dass der Benutzer einmal vergessen sollte empfangene Daten des USB-2-X Moduls über eine Read Message anzufordern, steht ein ausreichend großer Ringpuffer zur Verfügung. Interrupts, die den Versand von Nachrichten signalisieren, werden synchron von der Software ausgelöst und nach dem Versand deaktiviert. Auch für das Versenden von Nachrichten steht ein Ringpuffer zur Verfügung, der einen sicheren Datenversand gewährleistet.

4.3.2. USART

Das Design der USART Komponente entspricht prinzipiell der CAN Komponente (vgl. 4.3.1).

4.3.3. SPI

Anforderungen bezüglich des Designs:

1. Der Nachrichtenversand wird von der Software ausgelöst und ist deshalb synchron. Er findet aus Sicht der Software zu einem bestimmten Zeitpunkt statt.
2. Der Nachrichtenempfang ist ebenfalls synchron, da der Slave die empfangenen Daten im Takt des Masters simultan zurücksendet (vgl. 2.3.2).

Begründete Wahl:

Da der Nachrichtenversand und -empfang synchron ist, wird die SPI Komponente mittels Polling realisiert. Eine Pollingloop, die abfragt, ob alle gesendeten Daten empfangen wurden, verbraucht nicht unnötig Prozessorzeit, da die Daten zu genau dem gepollten Zeitraum erwartet werden. Diese Variante des Designs ist ohne weiteren Hardwareaufwand einfach zu implementieren, ohne einen Nachteil zu erzeugen.

4.3.4. USB

Anforderungen bezüglich des Designs:

1. Der Nachrichtenversand wird von der Software ausgelöst und ist deshalb synchron. Er findet aus Sicht der Software zu einem bestimmten Zeitpunkt statt.
2. Der Nachrichtenempfang ist asynchron und findet aus Sicht der Software zu einem unbestimmten Zeitpunkt statt.
3. Während der Abarbeitung einer empfangenen Nachricht treffen keine weiteren Nachrichten ein.

Begründete Wahl:

Da im Falle der USB Komponente die Communication Device Class (CDC) von Atmel genutzt wird und diese mittels Polling realisiert wurde, wird hier ein Nachteil in Kauf genommen. Für den Nachrichtenempfang über USB müssen die Statusregister des UDPs ständig abgefragt werden. Dies führt dazu, dass ständig Prozessorzeit verbraucht wird, da nicht sicher ist, ob zu dem gepollten Zeitpunkt auch eine Nachricht empfangen wurde. Für den Nachrichtenversand ergibt sich aus dem Design der CDC kein Nachteil, da er synchron erfolgt. Die Funktionalität des USB-2-X Moduls wird durch die Verwendung der CDC nicht beeinträchtigt, allerdings hätte sich das Design der USB Komponente bei der Verwendung von Interrupts besser ins Gesamtkonzept eingefügt.

4.3.5. Hauptprogramm

Anforderungen bezüglich des Designs:

1. Es muss in regelmäßigen Abständen geprüft werden, ob USB-2-X Nachrichten empfangen wurden, damit entsprechend reagiert werden kann.
2. Während der Abarbeitung einer Nachricht können keine weiteren Nachrichten empfangen werden.

Begründete Wahl:

Das Hauptprogramm wird als Superloop gestaltet, welche zu Beginn mittels Polling auf eine USB-2-X Nachricht wartet. Hierfür wird die Funktionalität der CDC genutzt. Durch ständiges Abfragen wird eine geringe Wartezeit garantiert. Außerdem wird durch dieses Verfahren gewährleistet, dass die Nachrichtenverarbeitung nicht unterbrochen wird. Erst wenn die Verarbeitung beendet ist, wird zu Beginn der Superloop wiederholt abgefragt, ob eine weitere USB-2-X Nachricht angekommen ist.

5. Implementierung

Nachdem im vorherigen Kapitel das Design der Komponenten beschrieben wurde, wird nun auf deren Implementierung eingegangen. Insbesondere wird die programmtechnische Umsetzung näher erläutert.

5.1. Modulübersicht

Da es sich beim USB-2-X Modul um ein in C geschriebenes, eingebettetes System handelt, wird jede Komponente durch ein Sourcecode-Paar, c-Datei und Header-Datei, repräsentiert. Interruptgesteuerte Komponenten beinhalten noch eine zusätzliche c-Datei zur Interruptsteuerung, deren Funktionen von dem Advanced Interrupt Controller (AIC) angestoßen werden. Sie enthalten also den Interrupthandler für das jeweilige Interface. Durch diese Trennung der Quelltexteinheiten ist es möglich, die Funktionen des Programms für den späteren Gebrauch einfach zu selektieren und wieder zu verwenden.[1]

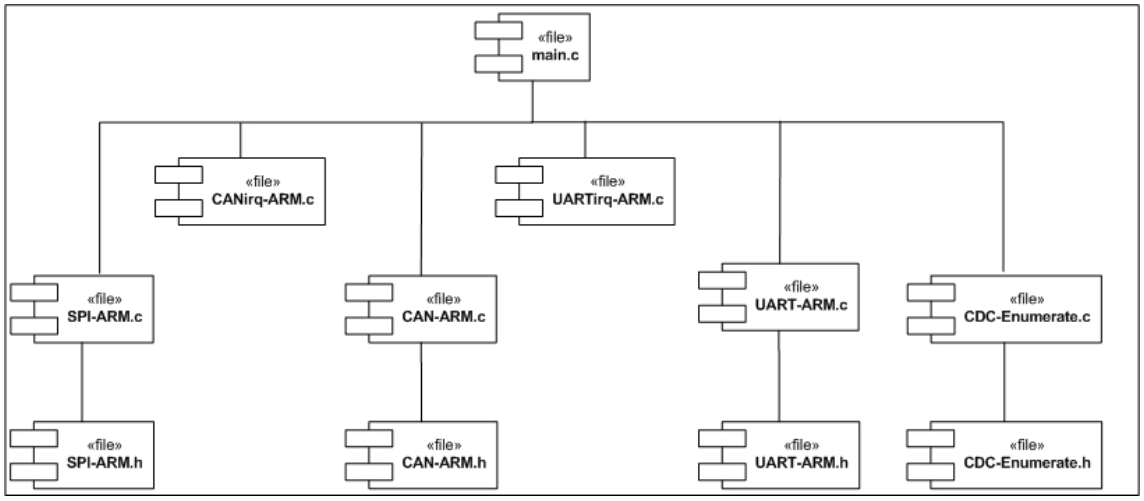

Abbildung 5.1.: Implementierung - Beteiligte Softwarekomponenten

[1] Vgl. Zimmer (2006)

5.2. Hauptprogramm

5.2.1. Nachrichtenverarbeitung

Das Zustandsdiagramm[2] zeigt, wie das USB-2-X System nach der Initialisierung arbeitet. Über die "AT91F_UDP_Read" Funktion wird abgefragt (vgl. 4.3.4 - Polling), ob eine Nachricht eingetroffen ist. Das System befindet sich dabei im Zustand "Warte auf Nachricht". Trifft eine Nachricht ein, wird sie entsprechend des USB-2-X Protokolls verarbeitet und das System kehrt zurück in den Hauptzustand. Um Daten über USB an den PC zu senden, steht die "AT91F_UDP_Write" Funktion bereit. Allerdings ist der Versand einer Nachricht über den UDP nur die Konsequenz einer empfangenen Nachricht. Der Versand kann also dem Zustand "Verarbeite Nachricht" zugeordnet werden. Beide Funktionen werden im Abschnitt (5.3.4) näher erläutert.

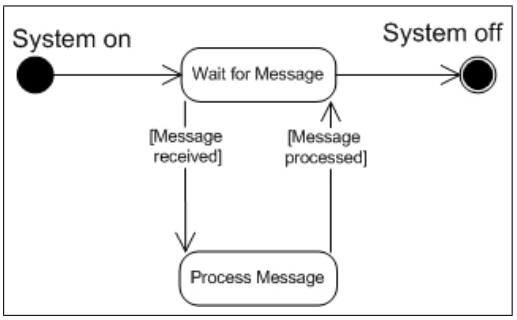

Abbildung 5.2.: Hauptprogramm - Zustandsdiagramm der Nachrichtenverarbeitung

Das Activitydiagramm[3] stellt den Ablauf des Zustandsdiagramms genauer dar. Bevor Nachrichten empfangen werden können, wird über die Funktion "AT91F_UDP_IsConfigured" (vgl. 5.3.1) geprüft, ob der UDP betriebsbereit ist. Insbesondere soll deutlich werden, dass, wenn eine Nachricht empfangen wurde, sie anhand ihrer Packet ID ausgewertet und eine entsprechende Funktion innerhalb einer Komponente (SPI, CAN, USART) ausgeführt wird. Im Falle einer Read Message werden die Daten aus einem Ringpuffer (CAN und USART) bzw. aus einem Register (SPI) des betroffenen Interfaces gelesen und über die oben erwähnte "AT91F_UDP_Write" Funktion an das User Interface auf dem Host PC gesendet. Im Falle einer Write Message werden Daten in einen Ringpuffer geschrieben und über das entsprechende Interface an eine Motorsteuerung gesendet. Im Falle einer Init Message wird das betroffene Interface initialisiert.

[2] Vgl. Fowler (2003)
[3] Vgl. Fowler (2003)

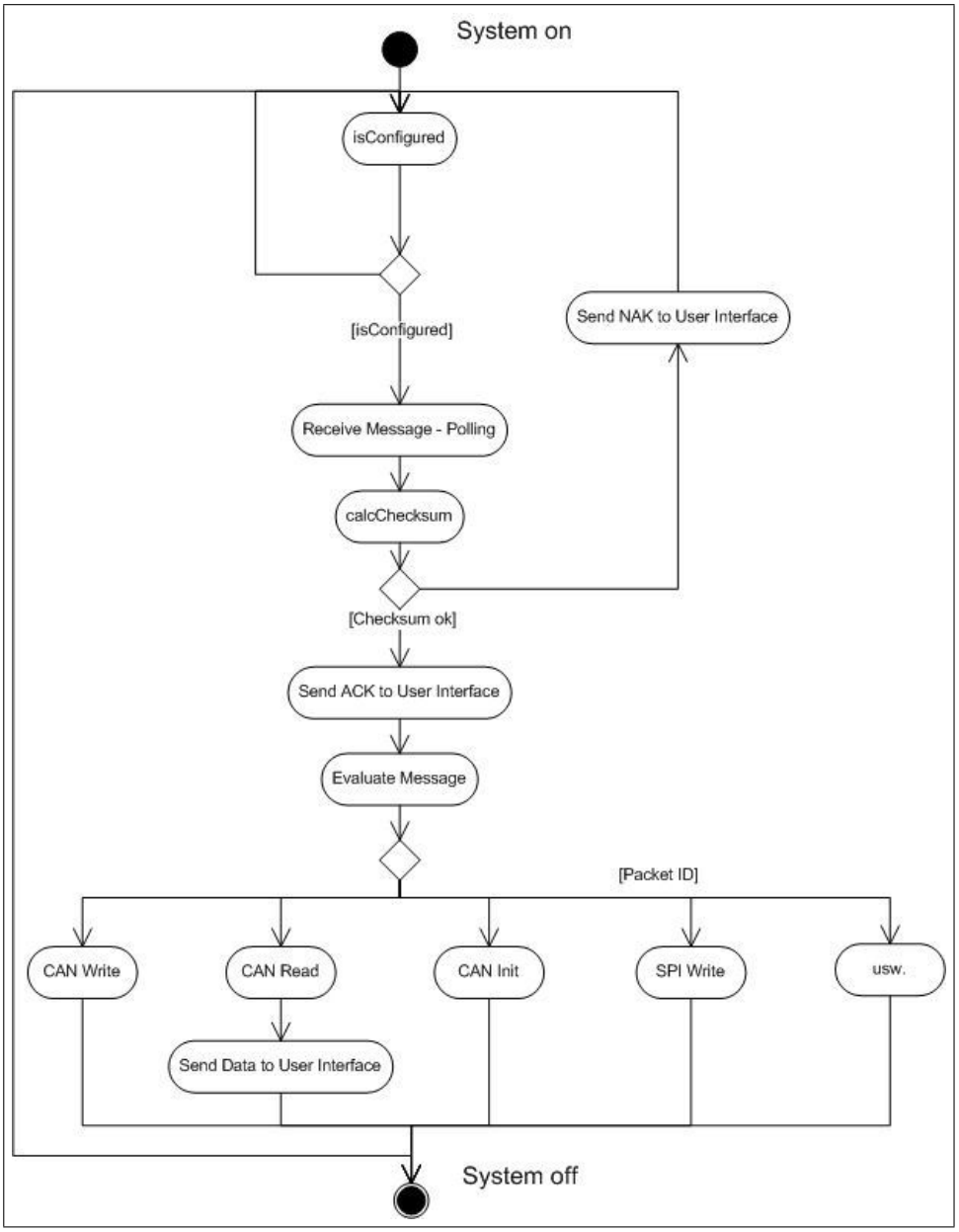

Abbildung 5.3.: Hauptprogramm - Activitydiagramm der Nachrichtenverarbeitung

Folgende Funktionen stehen innerhalb des Hauptprogramms für den Versand bzw. Empfang von Daten zur Verfügung.

Tabelle 5.1.: Hauptprogramm - Funktionen zur Nachrichtenverarbeitung

Funktion	Aufgabe
void sendCanWrite (char *pdata, unsigned int length)	Vom Host gesendete Daten in einen CAN Frame schreiben und diesen dann der "CanSendMessage" Funktion der CAN Komponente übergeben.
int sendCanRead (char *pdata)	Aufruf der "CanGetMessage" Funktion innerhalb der CAN Komponente, um Daten aus einem Puffer zu lesen und gelesene Daten in einen Char Array zu schreiben.
int sendRS485Read (char *readBuff)	Aufruf der "ReadUART" Funktion innerhalb der USART Komponente, um Daten aus einem Puffer zu lesen und gelesene Bytes in einen Char Array zu schreiben.
int sendReadWriteSPI (char *data, unsigned int length)	Aufruf der "ReadWriteSPI" Funktion innerhalb der SPI Komponente, um Daten zu senden und gesendete Bytes zu empfangen, um sie in einen Char Array zu schreiben.

5.2.2. Fehlerüberwachung

Damit sichergestellt ist, dass vom PC empfangene Daten auch korrekt übertragen wurden, werden sie vor der Weiterverarbeitung mittels einer Prüfsumme auf Fehler überprüft. Ist ein Fehler aufgetreten, wird ein NAK an das User Interface auf dem PC gesendet und die Datenübertragung beendet. So ist gewährleistet, dass keine fehlerhaften Daten an das Steuergerät eines Motors gesendet werden. Das Verfahren zur Berechnung der Prüfsumme ist dem USB-2-X Protokoll zu entnehmen (vgl. A.1). Für den Bulk Transfer ist diese Funktion überflüssig, da die Transferart selbst eine korrekte Zustellung der Daten garantiert, jedoch war das USB-2-X Protokoll wie vorgegeben zu implementieren. Folgende Funktion steht zur Fehlerüberwachung zur Verfügung:

Tabelle 5.2.: Hauptprogramm - Funktion zur Fehlerüberwachung

Funktion	Aufgabe
unsigned char calcChecksum (unsigned char payloadLength, char *pdata, unsigned int highORlow)	Berechnet die Prüfsumme für empfangene Daten gemäß USB-2-X Protokoll.

5.3. USB

5.3.1. Enumerationsprozess

Jedes Gerät muss Standard Device Requests beantworten können. Sie werden benötigt, um das Gerät zu identifizieren und zu konfigurieren. Um diese Anforderungen (vgl. 3.1.2) zu erfüllen, existieren folgende Funktionen:

Tabelle 5.3.: USB - Konfigurationsfunktionen

Funktion	Aufgabe
static uchar AT91F_UDP_IsConfigured (AT91PS_CDC pCdc)	Testet, ob der UDP des USB-2-X Moduls konfiguriert ist.
static void AT91F_CDC_Enumerate (AT91PS_CDC pCdc)	Verarbeitet ein empfangenes Setup Paket gemäß USB Spezifikation.

Mit der Funktion “AT91F_UDP_IsConfigured“ wird geprüft, ob ein Busreset erfolgte. Ist dies nicht der Fall, wird geprüft, ob ein Interrupt für den Endpunkt 0 erfolgt ist. Ist ein Interrupt erfolgt, bedeutet dies, dass ein Standard Device Request vom Host angekommen ist. Der Interrupt wird zurückgesetzt und die Enumerationsprozedur gestartet. Die “AT91F_CDC_Enumerate“ Funktion dekodiert das eingegangene Setup Paket (vgl. 5.3.2). Danach werden entsprechend seines Inhaltes Anfragen und Kommandos vom Host gemäß dem Ablauf einer Setuptransaktion beantwortet. Der vollständige Satz Standard Device Requests kann der USB Spezifikation (S.250) entnommen werden. Ist ein Busreset erfolgt, werden alle Endpunkte zurückgesetzt und konfiguriert (vgl. 2.1.13).

Das folgende Diagramm stellt den Verlauf der Funktion “AT91F_UDP_IsConfigured“ algorithmisch dar.

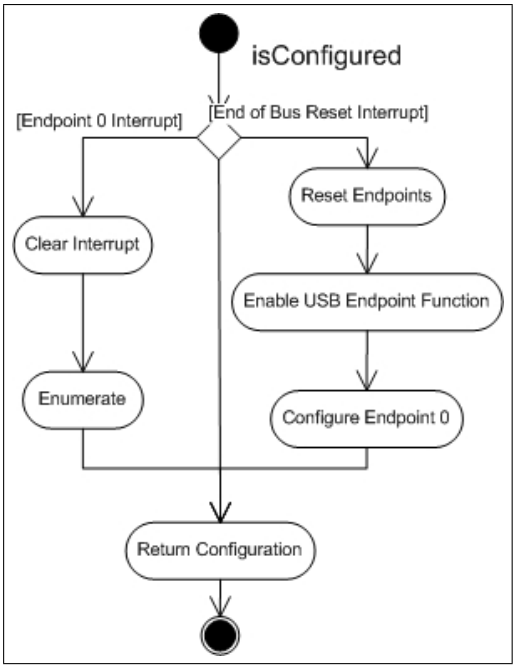

Abbildung 5.4.: USB - Activitydiagramm des Konfigurationsvorgangs

5.3.2. Setup Paket

Jeder Setup Request startet mit einem 8 Byte langen Setup Paket:

Tabelle 5.4.: USB - Das Setup Paket

Offset	Field	Size	Value	Description
0	bmRequestType	1	Bit-Map	**D7 Data Phase Transfer Direction** 0 = Host to Device 1 = Device to Host **D6...5 Type** 0 = Standard 1 = Class 2 = Vendor 3 = Reserved **D4...0 Recipient** 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	bRequest	1	Value	Request
2	wValue	2	Value	Value
4	wIndex	2	Index or Offset	Index
6	wLength	2	Count	Number of bytes to transfer if there is a data phase

Das “bmRequestType“ Feld legt im Wesentlichen die Transferrichtung, die Requestklasse und den Empfänger fest. Das “bRequest“ Feld legt fest, um welchen Request innerhalb der Klasse es sich handelt. Das “wValue“ und “wIndex“ Feld wird genutzt, um Parameter zu übergeben. Das “wLength“ Feld gibt die zu übertragenen Bytes an, wenn dem Request eine Datenphase folgt.

Bei einem Wert von 1000 0000b handelt es sich beispielsweise um einen Standard Device Request mit der Transferrichtung “Device to Host“. Angenommen, dass das “bRequest“ Feld den Wert 0x00 hat, handelt es sich um einen “GET_STATUS“ Request, der 2 Byte mit dem folgendem Format zurückgeben wird:

- D0 < “Bus-Powered“ oder “Self-Powered“
- D1 < Unterstützung von Remote Wakeup während des Zustands “Suspended“
- D2 - D15 < Reserviert

Programmtechnisch gestaltet sich der Ablauf der Dekodierung des Setup Pakets innerhalb der “AT91F_CDC_Enumerate“ Funktion dann folgendermaßen:

```
AT91PS_UDP pUDP = pCdc->pUdp;
uchar bmRequestType, bRequest;  ushort wValue, wIndex, wLength, wStatus;

// kein Setup Paket im FIFO???
if ( !(pUDP->UDP_CSR[0] & AT91C_UDP_RXSETUP) )return;
// extrahiere Setup Paket
bmRequestType = pUDP->UDP_FDR[0];
bRequest      = pUDP->UDP_FDR[0];
wValue        = (pUDP->UDP_FDR[0] & 0xFF);
wValue       |= (pUDP->UDP_FDR[0] << 8);
wIndex        = (pUDP->UDP_FDR[0] & 0xFF);
wIndex       |= (pUDP->UDP_FDR[0] << 8);
wLength       = (pUDP->UDP_FDR[0] & 0xFF);
wLength      |= (pUDP->UDP_FDR[0] << 8);
// Transferrichtung Device to Host?
if (bmRequestType & 0x80) {
pUDP->UDP_CSR[0] |= AT91C_UDP_DIR;
while ( !(pUDP->UDP_CSR[0] & AT91C_UDP_DIR) );
}
//Setup Paket wurde aus dem FIFO des EP0 gelesen
pUDP->UDP_CSR[0] &= ~AT91C_UDP_RXSETUP;
while ( (pUDP->UDP_CSR[0]  & AT91C_UDP_RXSETUP)  );

// Handle Standard Request -- Seite 250, Table 9-3 der USB Spezifikation
switch ((bRequest << 8) | bmRequestType) {

case STD_GET_STATUS_ZERO:
// Setze wStatus auf 0, um die spezifische Geräteeinstellung zu übertragen
wStatus = 0;
AT91F_USB_SendData(pUDP, (char *) &wStatus, sizeof(wStatus));
break;

...
```

Der Vollständigkeit halber muss noch erwähnt werden, dass nicht unterstützte Requests gemäß USB Spezifikation (Kapitel 8.4.5) mit einem STALL Handshake beantwortet werden. Um das USB Protokoll gemäß (2.1.10) zu erfüllen, stehen also folgende Funktionen zur Verfügung:

Tabelle 5.5.: USB - Funktionen für den Control Transfer

static void AT91F_USB_SendData (AT91PS_UDP pUdp, const char *pData, uint length)	Sendet Daten durch den Control Endpunkt.
void AT91F_USB_SendZlp (AT91PS_UDP pUdp)	Sendet ein Zero Length Paket durch den Control Endpunkt.
void AT91F_USB_SendStall (AT91PS_UDP pUdp)	Sendet ein STALL Paket zum Host.

5.3.3. Beispiel einer Setup Phase

Als letztes wird beispielhaft der Verlauf eines Setup Requests (vgl. 2.1.10) anhand eines Device Deskriptor Requests beschrieben.

In der Setup Phase sendet der Host ein SETUP Token, um dem USB-2-X Modul mitzuteilen, dass es sich bei dem nächsten Paket um ein Setup Paket handelt. Das Adressfeld enthält die Adresse des USB-2-X Moduls. Die Endpunkt Nummer ist 0, da es sich um einen Control Transfer handelt. Die Daten werden also über die Default Pipe übertragen. Danach sendet der Host ein DATA0 Paket, welches den 8 Byte langen Device Deskriptor Request enthält. Der UDP quittiert den erfolgreichen Empfang oder ignoriert das Paket, wenn ein Fehler aufgetreten ist. Der Host wird in diesem Fall das Paket erneut zustellen.

Tabelle 5.6.: USB - Setup Transaktion - Setup Phase

1. Setup Token	SYNC	PID	ADDR	ENDP	CRC	EOP
2. Data Paket	SYNC	PID	DATA0	CRC	EOP	
3. Ack Handshake	SYNC	PID	EOP			

Die oberen Pakete repräsentieren die erste USB Transaktion. In der Datenphase wird das USB-2-X Modul nun die 8 Byte dekodieren und feststellen, dass es sich um einen Device Deskriptor Request handelt.

Als Antwort wird das USB-2-X Modul den Device Deskriptor in der nächsten Transaktion senden, nachdem der Host ein IN Token gesendet hat. Da der Device Deskriptor größer als 8 Byte ist, muss die Transaktion in zwei Übertragungen aufgeteilt werden.

Tabelle 5.7.: USB - Setup Transaktion - Datenphase

1. In Token	SYNC	PID	ADDR	ENDP	CRC	EOP
2. Data Paket	SYNC	PID	DATA1	CRC	EOP	
3. Ack Handshake	SYNC	PID	EOP			
1. In Token	SYNC	PID	ADDR	ENDP	CRC	EOP
2. Data Paket	SYNC	PID	DATA0	CRC	EOP	
3. Ack Handshake	SYNC	PID	EOP			

Als letztes folgt die Statusphase. Wenn die Übertragung der Daten während der Datenphase erfolgreich war, sendet der Host ein Zero Length Paket, nachdem er ein OUT Token gesendet hat. Das Device quittiert dies mit einem ACK.

Tabelle 5.8.: USB - Setup Transaktion - Statusphase

1. Out Token	SYNC	PID	ADDR	ENDP	CRC	EOP
2. Data Paket	SYNC	PID	DATA1	CRC	EOP	
3. Ack Handshake	SYNC	PID	EOP			

5.3.4. Datenempfang

Für den Empfang von Daten wurde die Transferart Bulk gewählt, weil die Informationen, welche vom Host PC empfangen werden, nicht so zeitkritisch sind wie beispielsweise Audiodaten, aber in jedem Fall fehlerfrei empfangen werden sollen (vgl. 2.1.10). Für den Empfang von Daten über einen Bulk Endpunkt steht folgende Funktion zur Verfügung:

Tabelle 5.9.: USB - Funktion für den Nachrichtenempfang

Funktion	Aufgabe
static uint AT91F_UDP_Read (AT91PS_CDC pCdc, char *pData, uint length)	Liest die empfangenen Daten vom Bulk Endpunkt OUT.

Bei der “AT91F_UDP_Read“ Funktion wird zuerst geprüft, ob der UDP konfiguriert ist. Dies geschieht über die Funktion “AT91F_UDP_IsConfigured“ (vgl. 5.3.1). Tritt ein Fehler auf, wird der Lesevorgang abgebrochen. Ist der UDP betriebsbereit, wird über das RX_DATA_BKX (Data Receive Bank X) Flag geprüft, ob Daten angekommen sind. Sind Daten im FIFO Speicher[4] angekommen, werden sie byteweise aus dem FIFO des Endpunktes in einen Puffer gelesen, der die Größe des maximalen Data Payload einer USB-2-X Nachricht hat und das RX_DATA_BKX Flag zurücksetzt.

Nach jedem Paket wird die Speicherbank gewechselt (X=0 ODER X=1). Während der Mikrocontroller aus einem Empfangsspeicher liest, kann der UDP also in den anderen schreiben. Dieser Vorgang wird wiederholt, bis keine Daten mehr empfangen werden. Die empfangenen Daten stehen am Ende an einer bestimmten Speicherstelle. Zurückgegeben wird die Anzahl der empfangenen Bytes.

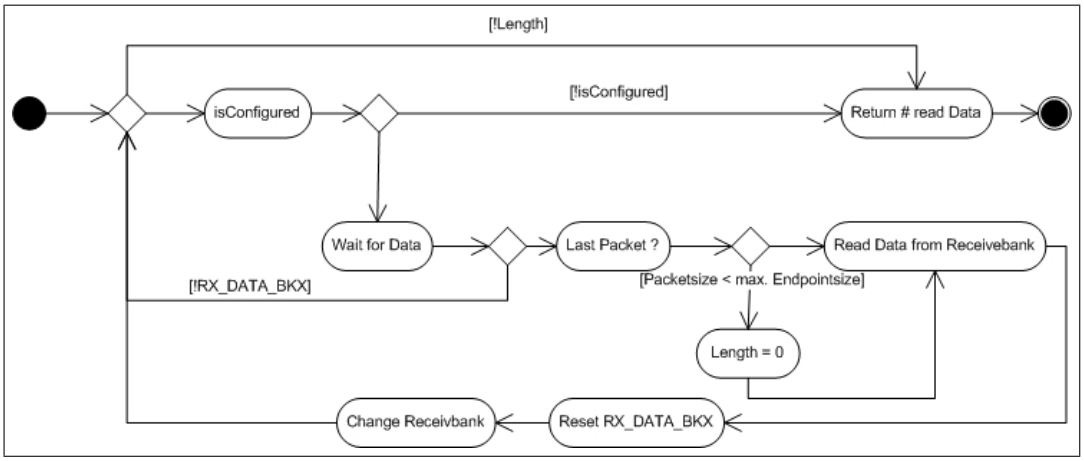

Abbildung 5.5.: USB - Activitydiagramm des Nachrichtenempfangs

[4] *FIFO* bezeichnet jegliche Verfahren der Speicherung, bei denen diejenigen Elemente, die zuerst gespeichert wurden, auch zuerst wieder aus dem Speicher entnommen werden. Vgl. wikipedia.de (2008)

5.3.5. Datenversand

Für den Versand von Daten über einen Bulk Endpunkt steht folgende Funktion zur Verfügung:

Tabelle 5.10.: USB - Funktion für den Nachrichtenversand

Funktion	Aufgabe
static uint AT91F_UDP_Write (AT91PS_CDC pCdc, const char *pData, uint length)	Versende Daten über den Bulk Endpunkt IN.

Bei der "AT91F_UDP_Write" Funktion wird zuerst die Speicherbank 0 mit Daten gefüllt und im Control Register des betroffenen Endpunktes über das Transmit Packet Ready (TXPKTRDY) Flag signalisiert, dass Daten zum Senden bereit liegen. Im Anschluss wird die zweite Speicherbank mit Daten gefüllt und gewartet, bis die Daten aus Speicherbank 0 vollständig übertragen sind. Wurden die Daten vollständig übertragen, wird das TXPKTRDY Flag vom UDP zurückgesetzt und gleichzeitig das Transfer Complete (TXCOMP) Flag gesetzt. Zum Senden der Daten aus der zweiten Speicherbank wird das TXCOMP Flag von der Software zurückgesetzt und das TXPKTRDY Flag erneut gesetzt.
Dieser Vorgang wiederholt sich, bis keine Daten zum Senden mehr vorhanden sind. Am Ende wird eine Null zurückgegeben, wenn alle Daten gesendet wurden.

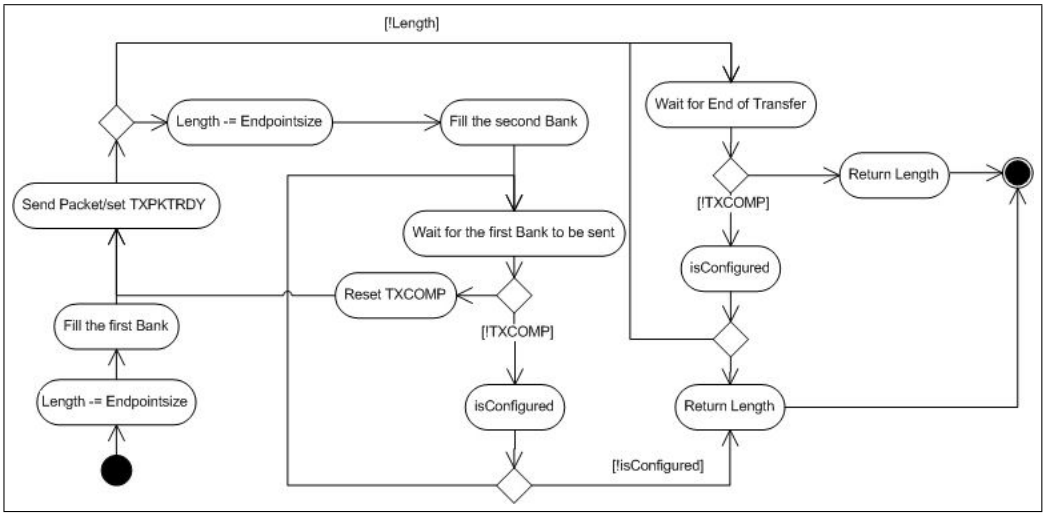

Abbildung 5.6.: USB - Activitydiagramm des Nachrichtenversands

5.4. CAN

5.4.1. Initialisierungsvorgang

Wie bereits in Abschnitt (5.2.1) erwähnt, wird der CAN Controller über eine Init Message (vgl. A.3) initialisiert. Der genaue Ablauf kann der Doxygen Dokumentation entnommen werden. Für die Initialisierung steht folgende Funktion bereit:

Tabelle 5.11.: CAN - Initialisierungsfunktion

Funktion	Aufgabe
void InitCan (unsigned int Baudrate, unsigned int AcceptedID, unsigned int SecondaryID)	Initialisiert den CAN Controller mit einer bestimmten Bitrate (vgl. 5.4.5) und legt fest, über welchen Identifier Nachrichten empfangen werden können.

5.4.2. Datenempfang

Für den Empfang von CAN Nachrichten steht folgende Funktion zur Verfügung:

Tabelle 5.12.: CAN - Funktion für dem Datenempfang

Funktion	Aufgabe
UCHAR CanGetMessage (TCanFrame *frame)	Lesen einer CAN Nachricht aus dem Eingangspuffer, sofern dieser nicht leer ist.

Eine CAN Read Message ruft innerhalb der CAN Komponente die “CanGetMessage“ Funktion auf, um die empfangenen Daten aus dem Ringpuffer zu lesen. Da der Empfang von Daten asynchron ist, wird die Interruptsteuerung für die Message Object Buffer (MOB) bereits nach einer Init Message aktiviert (vgl. 4.3.1). Für den Empfang stehen vier MOB zur Verfügung: Zwei zum Empfangen von Standard Frames oder Extended Frames mit der ersten ID und zwei zum Empfangen von Standard Frames oder Extended Frames mit der zweiten ID (vgl. 5.4.1). So können Geräte angeschlossen werden, die Daten mit unterschiedlichen IDs verschicken. Die Aufgabe, die empfangenen Daten aus dem Empfangs-MOB in den Ringpuffer zu schreiben, übernimmt der AIC Interrupthandler für den CAN Controller (vgl. 5.4.4). Die Größe des Ringpuffers beträgt 20 CAN Frames, so dass keine Daten verloren gehen, wenn der Benutzer einmal vergisst, Daten über eine Read Message (vgl. 4.1) abzuholen.

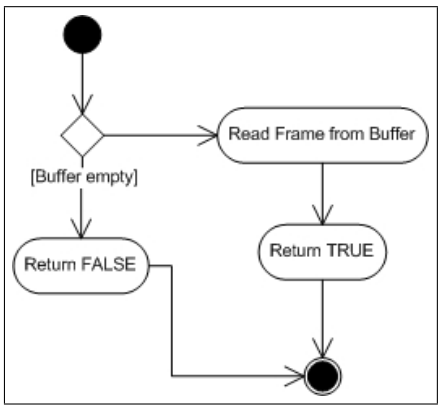

Abbildung 5.7.: CAN - Activitydiagramm der “CanGetMessage“ Funktion

5.4.3. Datenversand

Für den Versand von CAN Nachrichten steht folgende Funktion zur Verfügung:

Tabelle 5.13.: CAN - Funktion für dem Datenversand

Funktion	Aufgabe
UCHAR CanSendMessage (TCanFrame *frame)	Eine CAN Nachricht wird in den Sendepuffer geschrieben, sofern dieser nicht voll ist. Außerdem wird Mailbox 0 wieder aktiviert, falls diese gerade inaktiv ist und damit der Sendeinterrupt ausgelöst.

Eine CAN Write Message ruft innerhalb der CAN Komponente die “CanSendMessage“ Funktion auf, welche einen Frame in den Senderingpuffer schreibt und danach den MOB für das Senden freischaltet. Hier wird das Auslösen des Interrupts also über die Software gesteuert und ist deshalb synchron (vgl. 4.3.1). Die Aufgabe, Daten aus dem Ringpuffer über die jeweiligen Register zu versenden, übernimmt auch hier der AIC Interrupthandler für den CAN Controller. Die Größe des Ringpuffers ist so gewählt, dass ein sicherer Datenversand möglich ist. Es können also keine Elemente durch zu viele CAN Write Nachrichten überschrieben werden, da sie maximal einen CAN Frame transportieren. Nach dem Verschicken über die CAN Schnittstelle erfolgt zunächst die Reaktion des Motors. Ein Ringpuffer für 20 CAN Frames ist in diesem Fall ausreichend und erlaubt zukünftig Steuerbefehle, die größer als ein CAN Frame sind.

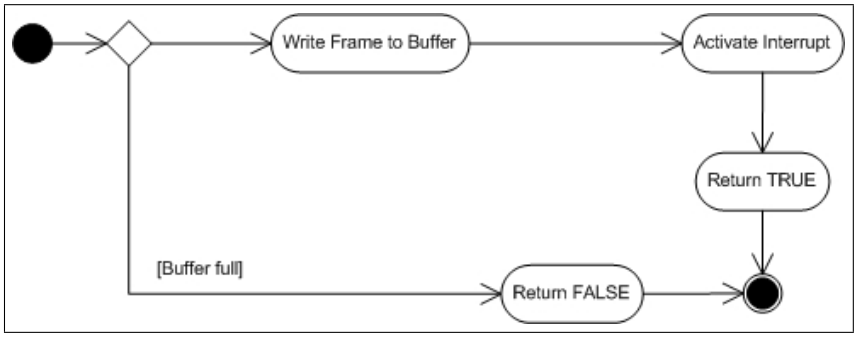

Abbildung 5.8.: CAN - Activitydiagramm der “CanSendMessage“ Funktion

5.4.4. Interrupthandling

Für das Verarbeiten von Interrupts stehen innerhalb des Interrupthandlers folgende Funktionen zur Verfügung:

Tabelle 5.14.: CAN - Funktionen des Interrupthandlers

Funktion	Aufgabe
void CanInterruptHandler(void)	Wird durch den AIC aufgerufen, wenn ein CAN Interrupt auftritt. Dies passiert, wenn eine CAN Nachricht angekommen ist oder die Mailbox 0 zum Senden frei geschaltet wird.
static void ARMCanGetMsg (volatile TCanFrame *RxFrame, AT91PS_CAN_MB Mailbox)	Empfangen einer CAN Nachricht über eine Mailbox des AT91SAM7X256. *RxFrame ist ein Zeiger auf eine Struktur vom Typ TCan-Frame und Mailbox ist ein Zeiger auf die zu lesende Mailbox-Datenstruktur.
static void ARMCanSendMsg (volatile TCanFrame *TxFrame)	Senden einer CAN Nachricht über den MOB0 des AT91SAM7X256. *TxFrame ist ein Zeiger auf eine Struktur vom Typ TCanFrame mit den zu sendenden Daten.

Das folgende Diagramm zeigt, in welchen Zuständen sich der Interrupthandler befinden kann.

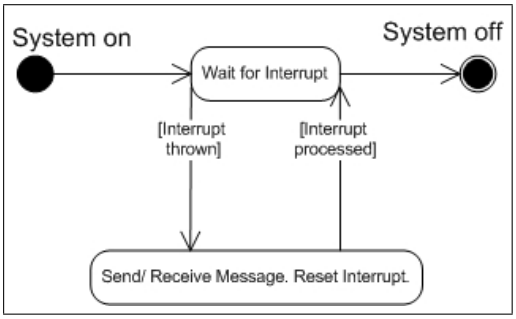

Abbildung 5.9.: CAN - Zustandsdiagramm der Interruptverarbeitung

Der Interrupthandler wartet auf einen Interrupt zum Senden oder Empfangen, nachdem er über die "InitCan" Funktion konfiguriert wurde. Tritt ein Interrupt auf, schreibt er die zu sendenden Daten aus dem Senderingpuffer in den MOB0 und aktiviert den Versand bzw. schreibt die Daten aus dem betroffenen MOB in den Empfangsringpuffer und setzt den Interrupt zurück. Im Falle des Sendens wird der MOB nach dem Versand deaktiviert, da im Transmit Mode nach erfolgreichem Versand sofort ein weiterer Interrupt geschmissen wird.

Treten Sende- und Empfangsinterrupt "gleichzeitig" auf, wird der Interrupthandler gestartet und fängt beide Interrupts über das jeweilige Flag ab. Treten Sende- und Empfangsinterrupt zeitversetzt auf, d.h. der Interrupthandler hat nicht mehr die Möglichkeit beide Flags abzufragen, wird er erneut gestartet, weil der noch nicht abgearbeitete Interrupt auch noch nicht zurückgesetzt wurde. Damit auch im Idle Mode kein Interrupt verloren gehen kann, wird der AIC ständig getaktet. Befindet sich der Prozessor im Idle Mode, reaktiviert der AIC ihn, damit ein Interrupt verarbeitet werden kann.

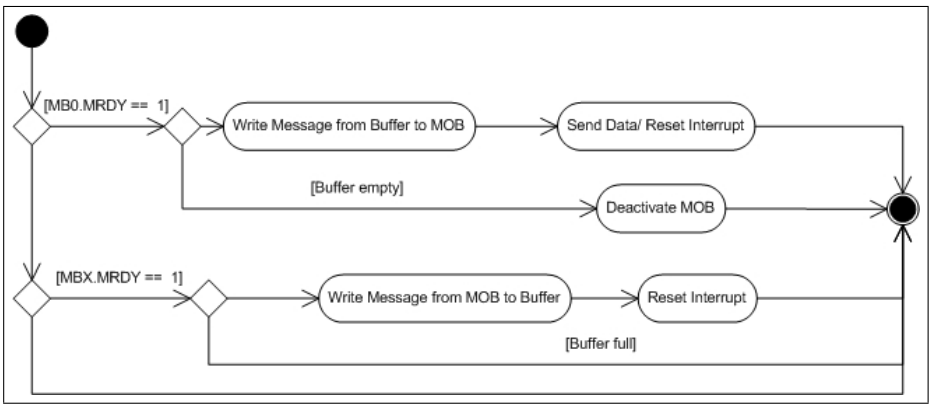

Abbildung 5.10.: CAN - Activitydiagramm der Interruptverarbeitung

5.4.5. CAN Bitrate

Alle CAN Controller, die am Bus angeschlossen sind, müssen mit der gleichen Bitrate arbeiten. Diese beträgt maximal 1 Mbit/s. Da die Clockfrequenzen der Controller voneinander abweichen, muss der Datenstrom synchronisiert werden.

Um dies zu realisieren, wird eine Bitperiode in unterschiedliche Segmente unterteilt.

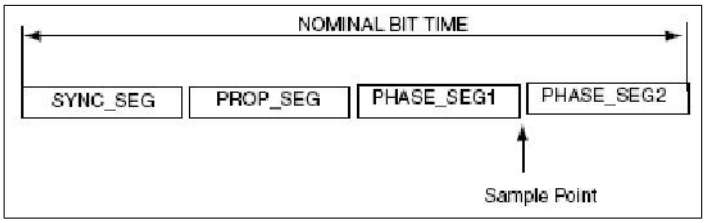

Abbildung 5.11.: CAN - Bitperiode

Jedes dieser Segmente besteht aus so genannten Time Quanta. Die gesamte Bitperiode kann aus 8-25 Time Quanta bestehen.

- Das Synchronisationssegment wird zum Synchronisieren der verschiedenen Nodes genutzt und ist ein Time Quantum lang. Der zugehörige Wert wird im Baud Rate Prescaler (BRP) Feld des CAN Baudrate Registers eingetragen.
- Das Propagationsegment soll die physikalisch bedingten Verzögerungszeiten des Busses kompensieren. Es setzt sich aus den Signallaufzeiten und Treiberschaltzeiten zusammen und kann 1-8 Time Quanta lang sein, einzutragen im PROPAG Feld des CAN Baudrate Registers.
- Die Phasensegmente eins und zwei werden genutzt, um Phasenverschiebungen durch unterschiedlich lange Clockphasen der Controller zu korrigieren. Um Verschiebungen von Flanken auszugleichen, werden sie gekürzt (Segment zwei) oder verlängert (Segment eins) und der Datenstrom wird neu synchronisiert. Die Weite dieser Verschiebung wird deshalb Re-Synchronisation Jump genannt und kann zwischen 1-4 Time Quanta lang sein. Er wird im Synchronisation Jump Width (SJW) Feld des CAN Baudrate Registers eingetragen. Der Samplingpoint[5] verschiebt sich entsprechend dieser Weite und die Bit Time wird verlängert bzw. verkürzt.

[5] Der *Samplingpoint* stellt den Zeitpunkt dar, an dem die Wertigkeit des übertragenen Bits festgestellt wird (gesampled) und liegt am Ende des Phasensegments eins.

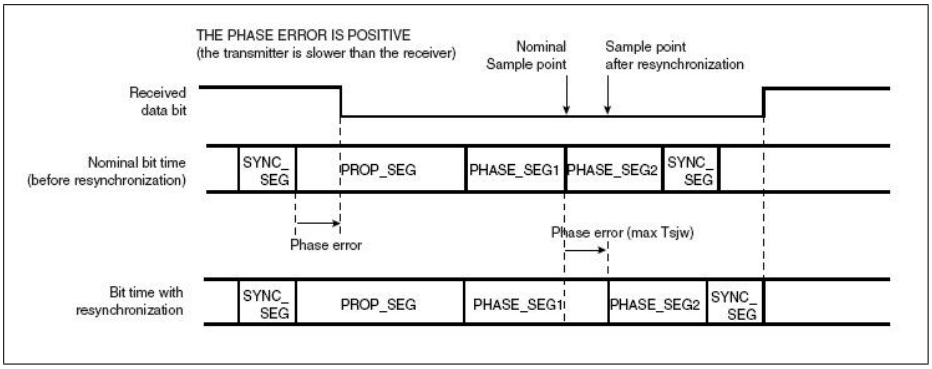

Abbildung 5.12.: CAN - Phasenverschiebung - positiv

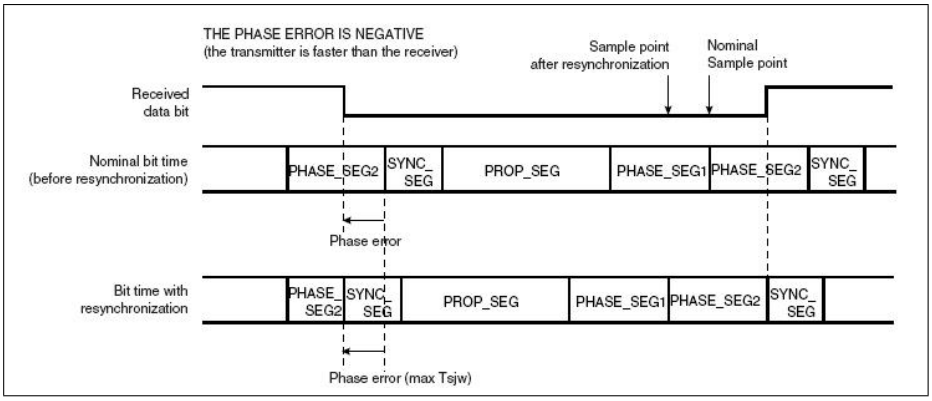

Abbildung 5.13.: CAN - Phasenverschiebung - negativ

Das Phasensegment eins kann 1-8 Time Quanta lang sein. Phasensegment zwei muss mindestens so lang sein wie die Information Processing Time (IPT), darf aber niemals länger sein als Phasensegment eins. Die IPT ist die Zeit, welche benötigt wird, um das Bitlevel einer Übertragung durch Sampling festzustellen und beträgt im Falle des Atmel CAN 2 Time Quanta. Eingestellt werden diese Parameter im PHASE1 und PHASE2 Feld des CAN Baudrate Registers.

Eine Bitperiode auf dem CAN Bus beträgt also:

$$Tbit = Tsync + Tprs + Tph1 + Tph2 \tag{5.1}$$

wobei

$$\begin{aligned} Tsync &= \frac{BRP+1}{MCK} \\ &= 1\,TQ \end{aligned} \tag{5.2}$$

Beispiel zur Berechnung der Segmente bei einer Frequenz von 48 MHz und einer Bitrate von 500 kbit/s:

$$MCK = 48\,\text{MHz} \tag{5.3}$$

$$\begin{aligned} CAN\ Baudrate &= 500\,\text{kbit/s} \\ &= 2\,\mu\text{s}/BitTime \end{aligned} \tag{5.4}$$

Verzögerungszeit von Bustreiber, Receiver und Signallaufzeit (20 m) : 190 ns

Time Quanta pro Bit Time: 16 (kann zwischen 8 und 25 liegen)

$$\begin{aligned} Tsync &= 1\ TQ \\ &= \frac{BitTime}{16} \\ &= 125\,\text{ns} \end{aligned} \tag{5.5}$$

$$\begin{aligned} BRP &= (Tsync * MCK) - 1 \\ &= 5 \end{aligned} \tag{5.6}$$

Das Propagationssegment berechnet sich wie folgt:

$$\begin{aligned} Tprs &= 2 * 190\,\text{ns} \\ &= 380\,\text{ns} \\ &= 3 * Tsync \end{aligned} \tag{5.7}$$

$$\begin{aligned} PROPAG &= \frac{Tprs}{Tsync} - 1 \\ &= 2 \end{aligned} \tag{5.8}$$

Die restliche Zeit wird auf die zwei Phasensegmente aufgeteilt. Ist die Anzahl der restlichen Zeiteinheiten gerade, wird sie gleichermaßen auf die beiden Segmente aufgeteilt.

$$\begin{aligned} Tphs1 + Tphs2 &= BitTime - Tsync - Tprs \\ &= (16 - 1 - 3)\ Tsync \\ &= 12 * Tsync \end{aligned} \tag{5.9}$$

$$\begin{aligned} Tphs1 &= Tphs2 \\ &= \frac{12}{2} * Tsync \\ &= 6\ Tsync \end{aligned} \tag{5.10}$$

$$\begin{aligned} PHASE1 &= PHASE2 \\ &= \frac{Tphs1}{Tsync} - 1 \\ &= 5 \end{aligned} \tag{5.11}$$

Ist sie ungerade, erfolgt die Verteilung wie folgt:

$$Tphs2 = Tphs1 + Tsync \tag{5.12}$$

Die Weite des Re-Synchronisation Jumps muss zwischen $Tsync$ und dem Minimum aus 4 $Tsync$ und $Tphs1$ liegen und berechnet sich wie folgt, wenn man das Maximum als sichere Variante wählt:

$$\begin{aligned} Tsjw &= Min(4 * Tsync, Tphs1) \\ &= 4\ Tsync \end{aligned} \tag{5.13}$$

$$\begin{aligned} SJW &= \frac{Tsjw}{Tsync} - 1 \\ &= 3 \end{aligned} \tag{5.14}$$

Letztendlich steht im CAN Baudrate Register dann folgender Wert: 0x00053255

5.5. RS485

Im Falle des USARTs muss die Software so gestaltet werden, dass ihre Funktionalität später mit einem RS485 Treiberbaustein ohne weiteren Aufwand kombiniert werden kann. Da ohne einen zusätzlichen Treiberbaustein das Senden über RS485 nicht möglich ist, wird der USART so konfiguriert, dass vorerst über eine normale RS232 Schnittstelle Daten asynchron übertragen werden. Es wird also ein Pin für das Empfangen von Daten (RX) und ein Pin für das Senden von Daten (TX) im Fullduplex Mode benötigt. Für die Funktionalität der RS485 Schnittstelle wird später der Request To Send (RTS) Pin benötigt, um das Senden und Empfangen zu steuern. Es werden dann beide Pins, TX und RX, für das Senden oder Empfangen im Halbduplex Mode differentiell genutzt. Die folgende Abbildung zeigt, wie der USART später mit einem zusätzlichem Treiberbaustein seine RS485 Funktionalität erhält.

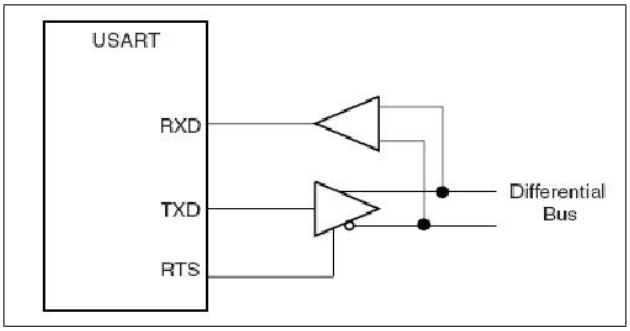

Abbildung 5.14.: Design - USART mit RS485 Funktionalität

5.5.1. Initialisierungsvorgang

Der Ablauf der Initialisierungsfunktion kann auch hier der Doxygen Dokumentation entnommen werden. Für die Initialisierung des USARTs steht folgende Funktion bereit:

Tabelle 5.15.: USART - Initialisierungsfunktion

Funktion	Aufgabe
void InitUART (UCHAR Baudrate)	Initialisierung des USARTs. Baudrate ist der Code für die Baudrate gemäß USB-2-X Protokoll (vgl. A6).

5.5.2. Datenempfang

Für den Empfang von Daten steht folgende Funktion zur Verfügung:

Tabelle 5.16.: USART - Funktion für den Empfang von Daten

Funktion	Aufgabe
UINT ReadUART (UCHAR *ch)	Lesen eines Zeichens aus dem Empfangspuffer.

Das Vorgehen beim Datenempfang entspricht prinzipiell dem der CAN Komponente (vgl. 5.4.2), nur handelt es sich bei den Daten um Bytes und nicht um Frames. Für den Nachrichtenempfang wird bereits nach einer Init Message die Interruptsteuerung aktiviert, da er, wie beim CAN Interface, asynchron ist. Die empfangenen Daten werden von dem Interrupthandler aus dem Empfangsregister in einen 30 Byte großen Empfangsringpuffer geschrieben und können danach über die "ReadUART" Funktion aus dem Ringpuffer gelesen werden (vgl. 4.3.1).

5.5.3. Datenversand

Für den Versand von Daten steht folgende Funktion zur Verfügung:

Tabelle 5.17.: USART - Funktion für den Versand von Daten

Funktion	Aufgabe
void WriteUART (UCHAR ch)	Senden eines Zeichens über den USART. (Einstellen in den Sendepuffer)

Auch der Datenversand entspricht prinzipiell dem der CAN Komponente (vgl. 5.4.3). Eine Write Message ruft die "WriteUART" Funktion auf, welche die Daten byteweise in einen 30 Byte großen Senderingpuffer schreibt und danach einen synchronen Interrupt auslöst (vgl. 4.3.1). Die Daten werden dann vom Interrupthandler verschickt.

5.5.4. Interrupthandling

Für das Verarbeiten von Interrupts steht innerhalb des Interrupthandlers folgende Funktion zur Verfügung:

Tabelle 5.18.: USART - Funktion des Interrupthandlers

Funktion	Aufgabe
void UARTInterruptHandler (void)	Der USART-Interrupthandler wird durch den AIC aufgerufen, wenn ein USART-Interrupt auftritt. Dies passiert, wenn ein Zeichen angekommen ist oder ein Zeichen gesendet werden kann.

Wie beim CAN Interface wartet der Interrupthandler auf einen Interrupt zum Senden oder Empfangen, nachdem er über die "InitUART" Funktion konfiguriert wurde. Auch der Rest des Ablaufs kann Kapitel (5.4.4) entnommen werden, da er prinzipiell gleich ist, allerdings mit dem Hinweis, dass die Register des USARTs sich zu denen des CAN Controllers im Aufbau unterscheiden.

5.5.5. USART Baudrate

Da bei der RS232 Schnittstelle eine asynchrone serielle Übertragung stattfindet, müssen beide Geräte mit der gleichen Clockfrequenz arbeiten. Synchronisiert wird über das Startbit.

Der Clock Divisor und mit ihm auch die Baudrate ergeben sich aus folgender Formel:

$$Baudrate = \frac{MCK}{Clock\ Divisor * 16} \tag{5.15}$$

d.h.

$$Clock\ Divisor = \frac{MCK}{Baudrate * 16} \tag{5.16}$$

Einzutragen ist der Clock Divisor im Baud Rate Generator Register (BRGR) des USARTs.

Der Baudrate Error sollte dabei nicht mehr als 5% betragen und errechnet sich wie folgt:

$$Error = 1 - \frac{Expected\ Baudrate}{Actual\ Baudrate} \tag{5.17}$$

Der zusätzliche Teiler 16 ergibt sich aus dem Oversampling, das verwendet wird, um im Receiver die Daten mit einer höheren Granularität zu samplen und Transmitter und Receiver synchron zu halten. Dies ist nötig, da die Clockphasen bei gleicher Frequenz unterschiedlich lang sind. Rechnerisch ermittelt der Receiver die übertragenen Zustände wie folgt:

Sei T eine Bitperiode auf der Übertragungsstrecke.

$$T = \frac{1}{Baudrate} \tag{5.18}$$

Die Samplingrate einer Bitperiode innerhalb des Receivers soll TR sein. Jeder Zustand wird 16-fach abgetastet.

$$TR = \frac{T}{16} \tag{5.19}$$

Die Zeit, die durch zusätzlichen Schaltungsaufwand innerhalb des Receivers verloren geht, beträgt zwei Takte zum Synchronisieren.

$$TS = 2 * TR \tag{5.20}$$

Synchronisiert wird über das Startbit, dieses ist der Übergang von HIGH nach LOW. Dieser Übergang wird nach acht Takten als Startbit erkannt. Im inaktiven Zustand hat die Leitung also den Wert HIGH. Wird das Startbit über einen Flankendetektor erkannt, wird ein interner Zähler auf Null gesetzt. Der Zähler wird von der Clock Domain des Receivers getaktet und sampled eine Bitperiode. Für das nächste Datenbit wird er wieder auf Null gesetzt. Der Datenstrom wird also mit dem Zähler synchronisiert.

Der Synchronisierungsvorgang dauert also TS Zeiteinheiten und das folgende Datenbit kann ($1.5\ T - TS$) Takten nach dem Beginn des Startbit 16-fach oversampled werden, wobei von einer Baudrate von 9600 Baud/s ausgegangen wird.

$$\begin{aligned} 1.5\ T - TS &= 1.5 * \frac{1}{Baudrate} - 2 * \frac{1}{\frac{Baudrate}{16}} \\ &= 1.5 * \frac{\frac{1}{Baudrate}}{16} - 2 * \frac{\frac{1}{\frac{Baudrate}{16}}}{16} \\ &= 1.5 * TR - 2 * \frac{TR}{16} \\ &= 24 * TR - 2 * TR \\ &= 156.25\ \mu s \end{aligned} \tag{5.21}$$

Es sind also 156.25 µs bis zur nächsten Bitmitte nach dem Startbit, wenn man die Schaltzeit zum Synchronisieren vernachlässigt. Von Datenbitmitte zu Datenbitmitte dauert es dann 104 µs.

Das heißt, dass nach dem Synchronisieren die nächste Bitmitte ($24\ TR - 2\ TR$) Takte nach dem Startbit erwartet werden kann. Ab dann ist jede folgende Bitmitte nach 16 Takten erreicht und kann 16-fach oversampled werden. Dafür hat der Mikrocontroller bei 9600 Baud/s 104 µs Zeit. Aus dem Ergebnis der Abtastung bildet der Receiver einen Mittelwert des übertragenen Symbols. An dieser Stelle sei noch gesagt, dass bei der RS232 Schnittstelle die Bitrate gleich der Baudrate ist, da ein Symbol über zwei Zustände, also ein Bit, kodiert wird.

Das Diagramm verdeutlicht, wann das Startbit detektiert wird und wann zum ersten Mal gesampled wird.

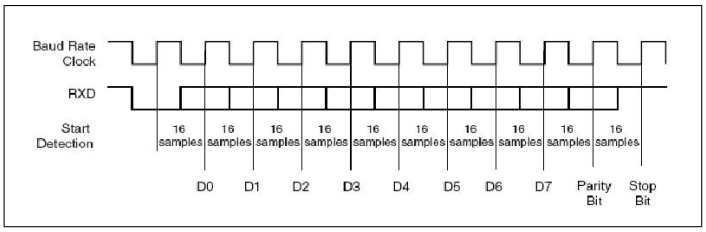

Abbildung 5.15.: USART - Diagramm des asynchronen Datenempfangs

5.6. SPI

5.6.1. Initialisierungsvorgang

Der genaue Ablauf der Initialisierungsfunktion kann der Doxygen Dokumentation entnommen werden. Für die Initialisierung des SPIs steht folgende Funktion bereit:

Tabelle 5.19.: SPI - Initialisierungsfunktion

Funktion	Aufgabe
void InitSPI(void)	Initialisierung des SPIs

5.6.2. Versand und Empfang von Daten

Für den Versand und Empfang von Daten steht folgende Funktion bereit:

Tabelle 5.20.: SPI - Funktion für den Versand und Empfang von Daten

Funktion	Aufgabe
UCHAR ReadWriteSPI (UCHAR DeviceNumber, UCHAR Data, UCHAR LastTransfer)	Senden und Empfangen über das SPI, wobei DeviceNumber die zugehörige Gerätenummer ist, Data die zu übertragenen Bytes enthält und LastTransfer signalisiert, ob das nächste Byte das letzte ist.

Um Daten an ein SPI Device zu senden, muss die zu sendende Nachricht byteweise in das Transmit Data Register (TDR) gelegt werden. Dann werden die Daten im Takt des Masters gesendet. Der Sendevorgang eines Bytes ist beendet, wenn das Receive Data Register Full (RDRF) Flag vom SPI gesetzt wird. Am Ende wird das empfangene Byte aus dem Receive Data Register (RDR) zurückgegeben. Nach dem letzten Byte wird das SPI Device über die Chip Select Leitung deselektiert.

Sollen nun Daten an einen weiteren Slave gesendet werden, übergibt man der "ReadWriteSPI" Funktion bei der nächsten Übertragung einfach die zugehörige Gerätenummer, um seine Chip Select Leitung zu selektieren. Das ist im Falle des USB-2-X Moduls nicht nötig, da alle Module sich über die Adresse 0 ansprechen lassen, jedoch lässt sich der Code dadurch später problemlos mit mehr als einem SPI Slave verwenden.

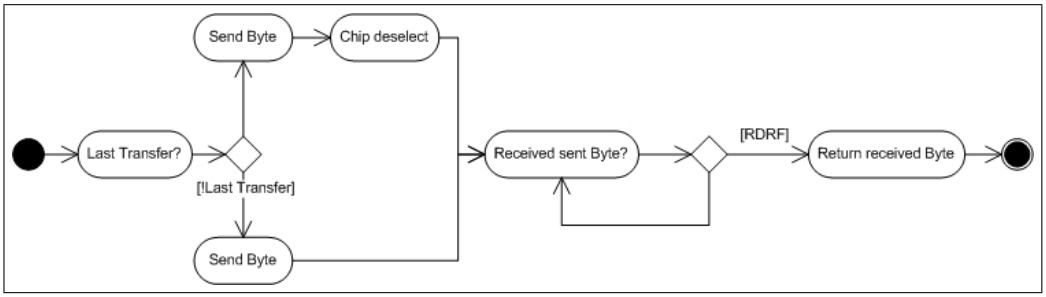

Abbildung 5.16.: SPI - Activitydiagramm des Nachrichtversands

5.7. User Interface

Zum Testen der Funktionalität des USB-2-X Moduls wird auf der Host PC Seite ein User Interface benötigt, das im Wesentlichen die verschiedenen Nachrichtentypen generiert, an das USB-2-X Modul versendet und am Ende eine einfache Übertragungsstatistik ausgibt. Da das User Interface nur ein Seiteneffekt dieser Arbeit ist, wird es nur kurz angesprochen.

5.7.1. Code-Modellierung

Das Komponentendiagramm stellt die Dateistruktur des User Interfaces dar. Dieses bindet die USB Library ein und hat so die Möglichkeit über einen Treiber der Firma Atmel mit dem USB-2-X Modul Daten auszutauschen. Der Treiber selbst wird über seinen Global Unique Identifier (GUID) identifiziert. Er entspricht der VendorID des USB-2-X Moduls. Diese muss also bei der Softwareentwicklung bekannt sein.

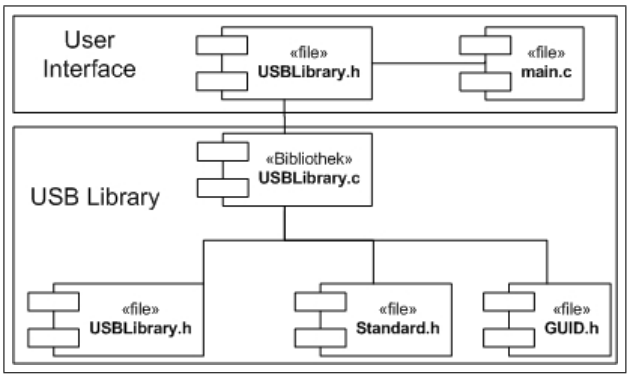

Abbildung 5.17.: User Interface - Komponentendiagramm

5.7.2. Benutzeroberfläche

Für die Entwicklungsphase wurde eine Software entwickelt, die mittels einfacher Tastatureingaben gesteuert wird. Hierfür steht ein selbsterklärendes Menü zur Verfügung, über das verschiedene Nachrichtentypen im USB-2-X Format generiert werden können und letztendlich an das USB-2-X Modul verschickt werden. Bisher gestaltet sich die Benutzeroberfläche wie folgt:

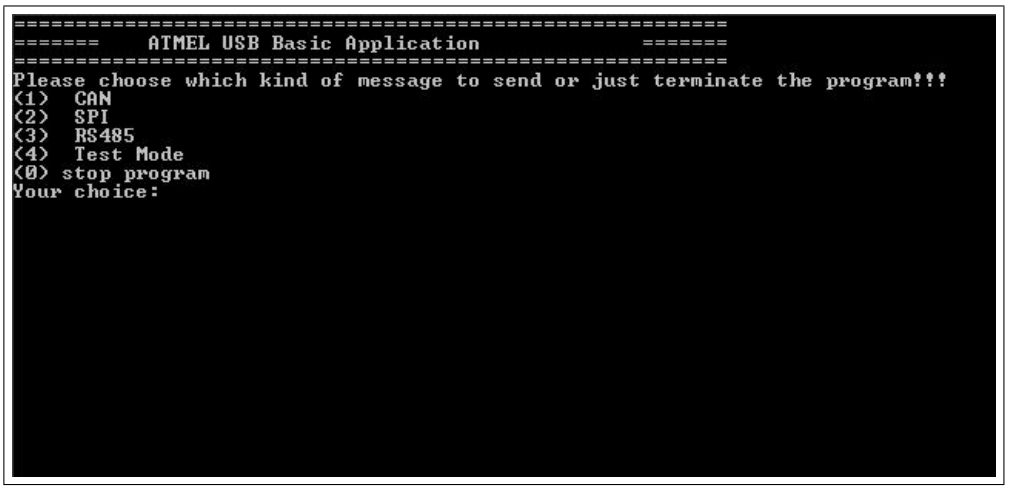

Abbildung 5.18.: User Interface - Benutzeroberfläche

5.7.3. Programmablauf

Das Zustandsdiagramm zeigt den groben Ablauf des Hauptprogramms. Es wird auf eine Benutzereingabe gewartet, die dann verarbeitet wird. Dieser Vorgang wiederholt sich, bis der Benutzer das Programm mit seiner Eingabe beendet.

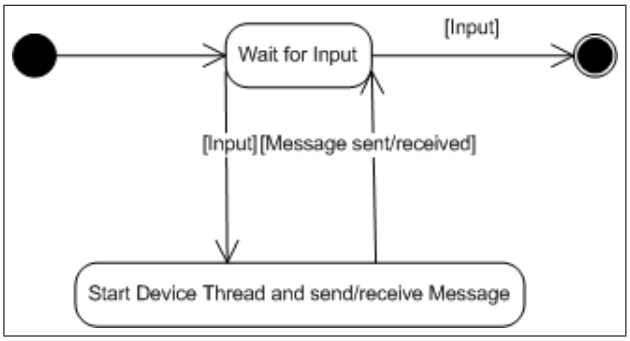

Abbildung 5.19.: User Interface - Zustandsdiagramm

Das Activitydiagramm stellt den algorithmischen Ablauf des Programms genauer dar. Der Programmablauf startet mit der Menüauswahl (Startzustand) und endet mit der Auswahl “stop program“. Hat der Benutzer seine Wahl getroffen, wird geprüft, ob das USB-2-X Modul mit dem Rechner verbunden ist. Ist dies der Fall, wird ein Device Thread gestartet. Innerhalb der Superloop wird nun solange mit der Programmfortführung gewartet, bis der Thread beendet ist. Der Thread selbst ruft Funktionen zu Nachrichtengenerierung auf, sendet die Nachricht und wartet auf eine Antwort. Nachdem die Antwort erfolgte, wird die Übertragungsstatistik erhöht und der Thread beendet. Die Superloop wird fortgeführt, gibt die Übertragungsstatistik aus und wartet auf die nächste Menüauswahl des Benutzers.

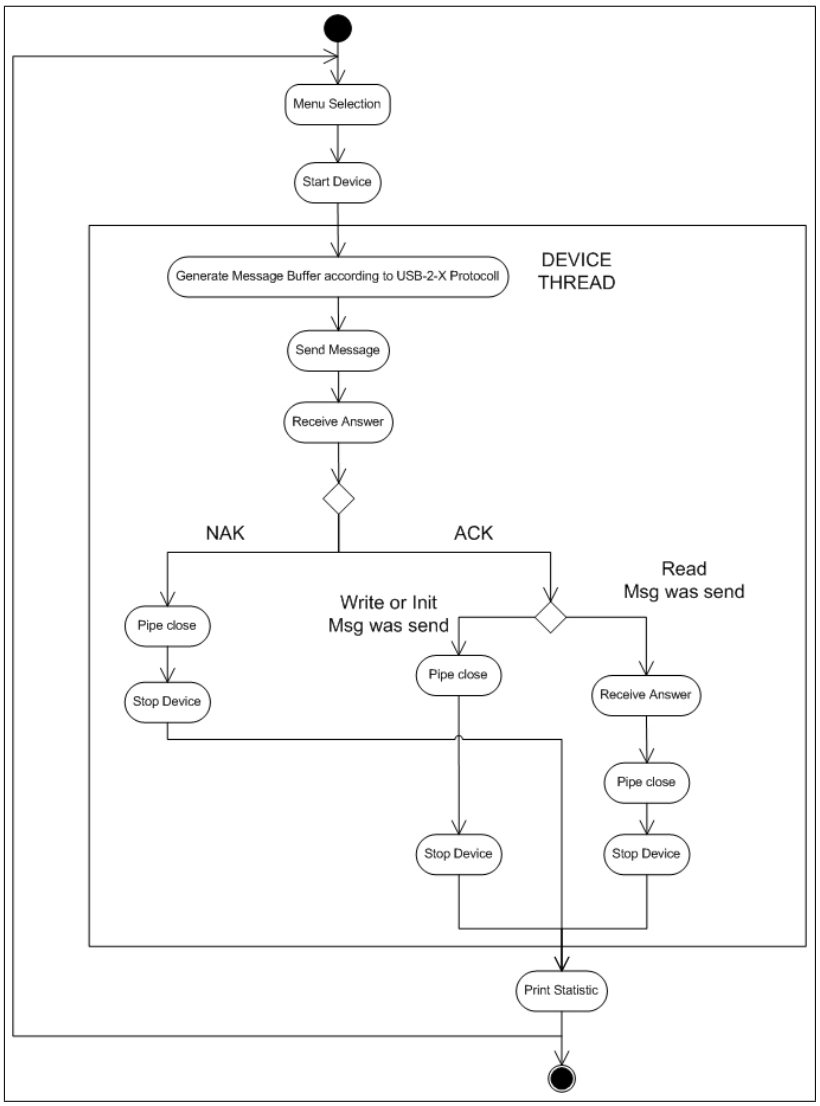

Abbildung 5.20.: User Interface - Activitydiagramm

5.7.4. Wesentliche Funktionen

Von der Darstellung der Treiberfunktionalität und der USB Library, die mit dem Treiber arbeitet, wird hier abgesehen, da diese komplett funktionstüchtig übernommen wurden und nicht Teil dieser Arbeit sind.

Tabelle 5.21.: Wesentliche Funktionen des User Interfaces

Funktion	Aufgabe
int main (int argc, char* argv[ ])	Benutzerschnittstelle und Aufruf der "startDevice" Funktion. Wenn der Device Thread beendet ist, wird die Anzahl der übertragenen Bytes ausgegeben.
HANDLE startDevice (PDEVICE ptDevice, HANDLE threadMutex)	USB-2-X Modul starten und DeviceThread aufrufen.
DWORD WINAPI DeviceThread (void *pVoidDevice)	Nachrichten über USB Pipe versenden und empfangen.
void printStatistic (PDEVICE ptDevice)	Zeigt die übertragenen Bytes über den kompletten Programmverlauf an.

6. Test

In diesem Kapitel wird zunächst beschrieben, wie die Funktionalität des Gesamtsystems getestet wurde. Ein zweiter Test prüft, ob bei der Übertragung von Daten im Bulk Modus die maximale Übertragungsrate erreicht wird, um folgend im Ausblick (vgl. 7.2) die Erweiterungsmöglichkeiten darzustellen.

6.1. Gesamtsystem

Um die Funktionalität des Gesamtsystems zu testen, wurde wie folgt vorgegangen:

1. Verschicken einer Init Message vom User Interface auf dem Host PC über USB, um das jeweilige Interface des USB-2-X Moduls gemäß USB-2-X Protokoll zu initialisieren.
2. Verschicken einer Write Message vom User Interface auf dem Host PC über USB, um zu prüfen, ob die Daten über das jeweilige Interface auf dem USB-2-X Modul weitergeleitet werden.
3. Verschicken einer Read Message vom User Interface auf dem Host PC über USB, um zu prüfen, ob das USB-2-X Modul Daten über das jeweilige Interface korrekt empfängt und sie dann über USB an das User Interface auf dem Host PC weiterleitet.

Um diesen Test zu realisieren, wurde folgender Versuchsaufbau verwendet:

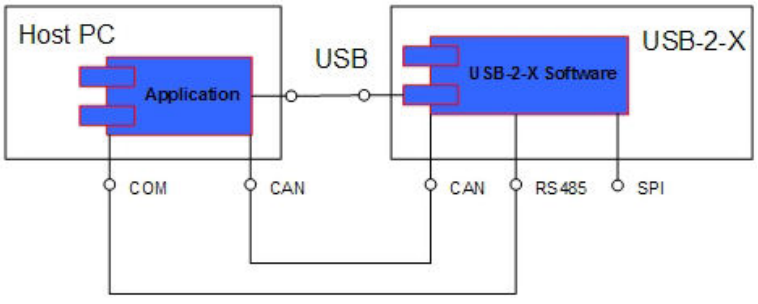

Abbildung 6.1.: Testaufbau

- Zum Testen der CAN Funktionalität des USB-2-X Moduls wurde eine CAN Schnittstelle auf dem Host PC installiert, die mit der CAN Schnittstelle des USB-2-X Moduls verbunden wurde und ein CAN Monitor verwendet, der über das User Interface verschickte Nutzdaten einer Write Message empfängt. Zum Testen der Read Message wurde eine CAN Nachricht vom CAN Monitor an die CAN Schnittstelle des USB-2-X Moduls geschickt und ihr Empfang über die USB Schnittstelle des Host PCs auf Seiten des User Interfaces geprüft.
- Zum Testen der RS485 Funktionalität des USB-2-X Moduls wurde die USART Schnittstelle des USB-2-X Moduls mit einer COM Schnittstelle des Host PCs verbunden und das Hyperterminal von Microsoft verwendet, welches über das User Interface verschickte Nutzdaten einer Write Message empfängt. Zum Testen der Read Message wurden Daten vom Hyperterminal an die USART Schnittstelle des USB-2-X Moduls geschickt und deren Empfang über die USB Schnittstelle des Host PCs auf Seiten des User Interfaces geprüft.
- Zum Testen der SPI Funktionalität des USB-2-X Moduls wurden die Pins MISO und MOSI der SPI Schnittstelle des Entwicklungsboards miteinander verbunden und die Funktion “Local Loopback Enable“ aktiviert, um die SPI Schnittstelle im Master Mode zu testen. Der Verlauf einer Read Message wurde für das SPI Interface nicht getestet, da ein SPI Slave empfangene Daten simultan an den Master zurücksendet. Es wurde also ausschließlich überprüft, ob die versendeten Daten einer Write Message als Loopback empfangen wurden.

Der Test des Gesamtsystems hat die Funktionalität des USB-2-X Moduls bestätigt. Die Daten wurden entsprechend des USB-2-X Protokolls (vgl. A) fehlerfrei verarbeitet.

6.2. USB

Bei diesem Test werden N Nachrichten über USB [1] an das USB-2-X Modul geschickt, welche dann umgehend zurückgesendet werden. Am Ende wird geprüft, ob die gesendeten Daten fehlerfrei empfangen wurden und annähernd die theoretisch mögliche Übertragungsrate erreicht wurde. Die Berechnung für den theoretischen Wert sieht wie folgt aus:

Die maximale Übertragungsrate für den Full Speed Modus beträgt 12 Mbit/s. Im besten Fall können also 12000 bit/ms übertragen werden, da jede Millisekunde ein SOF Paket generiert wird. Es werden also 83.33 ns/bit benötigt.

[1] Vgl. Sauter (2007)

Der Overhead für das Protokoll und das Bitstuffing für den Bulk Transfer betragen zusammen 13 Byte und die Fifogröße der Endpunkte für das USB-2-X Modul 64 Byte.

Die Anzahl der Übertragungen pro Frame errechnet sich deshalb wie folgt:

$$max.\,Übertragungen/Frame = \frac{\text{max. bit/ms}}{Overhead + Fifogröße} \tag{6.1}$$

$$\begin{aligned} max.\,Übertragungen/Frame &= \frac{12000\,\text{bit/ms}}{104\,\text{bit} + 512\,\text{bit}} \\ &= 19\,\text{Übertragungen/ms} \end{aligned} \tag{6.2}$$

Die maximale Übertragungsrate berechnet sich entsprechend:

$$max.\,Übertragungsrate = max.\,Übertragungen * max.\,Fifogröße \tag{6.3}$$

$$\begin{aligned} max.\,Übertragungsrate &= 19 * 512\,\text{bit} \\ &= 9.728\,\text{bit/ms} \end{aligned} \tag{6.4}$$

Vorausgesetzt, die Daten liegen zum Senden bereit, werden die ersten 64 Byte sofort versendet. Der weitere Verlauf kann allerdings wie folgt aussehen:

1. Die weiteren Daten können aus einem gemeinsamen Speicher von Prozessor und UDP übertragen werden.
2. Die Pakete müssen über eine IO Schnittstelle einem getrenntem Speicher übergeben werden.

   Für Punkt (2) gibt es wiederum zwei Möglichkeiten:

   a) Der UDP verwaltet zwei Speicherbereiche. Während der Inhalt des einen übertragen wird, kann der zweite bereits wieder mit Daten gefüllt werden.

   b) Der UDP verfügt nur über einen Speicherbereich zum Übertragen von Daten.

Für Punkt (1) gilt:

- Liegen die Pakete in einem gemeinsamen Speicher, können sie nacheinander verschickt werden.

Für Punkt (2a) gilt:

- Damit die 64 Byte im zweiten Speicher bereitstehen, wenn die ersten 64 Byte gesendet wurden, müssen sie innerhalb von 43 µs (512 ∗ 83.33 ns) von der IO Schnittstelle in den Speicher übertragen werden. Die Busgeschwindigkeit, mit der die Daten aus dem Speicher in den FIFO des Endpunktes übertragen werden, beträgt ($MCK/2$ MHz), also 24 MHz. Ein Bit wird also innerhalb von 41.67 ns übertragen, 64 Byte in 21.35 µs. Verrechnet man nun noch einen Verwaltungsoverhead mit dem Faktor 3, so kommt man auf 64 µs. Auf einem 8 Bit breiten IO Bus bedeutet dies, dass 8 µs nötig sind, um die Daten vom Static RAM (SRAM) in den FIFO Puffer zu packen.

Für Punkt (2b) gilt:

- Steht nur eine Speicherbank zur Verfügung, entsteht zwischen zwei Paketen eine Wartezeit in der Länge, die für das Auffüllen des Puffers notwendig ist.

Für das USB-2-X Modul trifft Punkt (2a) zu, da das Entwicklungsboard über einen Dual Ported RAM (DPR) verfügt. Es kann also davon ausgegangen werden, dass maximal 9973.76 bit/ms übertragen werden können.

Um die tatsächliche Datentransferrate zu ermitteln, wurden Nachrichten von unterschiedlicher Größe übertragen, bis das Gesamttransfervolumen 2000000 Byte betrug. Anschließend wurde die Dauer der Übertragung gemessen. Die Daten wurden mit der Transferart Bulk übertragen und kamen immer fehlerfrei an. Allerdings wurde der theoretische Wert von 9973.76 bit/ms niemals erreicht. Folgend die Tabelle mit den gemessenen Werten.

Tabelle 6.1.: Test - Messung der USB Transferrate

Paketgröße	Übertragungsrate
50 Byte	170696 bit/s
100 Byte	403150 bit/s
200 Byte	800000 bit/s
300 Byte	1230769 bit/s
1000 Byte	2666666 bit/s
10000 Byte	5333333 bit/s
15000 Byte	5742857 bit/s

Wie man erkennen kann, nähert sich die Übertragungsrate einem Wert von etwa 6 Mbit/s. Die Abweichung vom Maximalwert kann leider nur theoretisch begründet werden. Zum einen kann gesagt werden, dass das Entwicklungsboard nur mit der isochronen Übertragungsart und Endpunkten, die über einen DPR verfügen, laut Spezifikation eine maximale Übertragungsrate von 8 Mbit/s erreichen kann, zum anderen müssen die Daten auf der Seite des Host PCs den kompletten USB Stack durchlaufen, bevor sie innerhalb einer Applikation verarbeitet werden können, bzw. bevor sie vom Host Controller in Paketen von 64 Byte an das USB-2-X Modul verschickt werden. Das kostet Zeit, die bei den obigen Berechnungen (vgl. 6.2) nicht berücksichtigt wurde.

Nun stellt sich die Frage, warum ausgerechnet der isochrone Datentransfer zum Erreichen der maximalen Transferrate eingesetzt werden soll, wo doch der Bulk Transfer rechnerisch eine höhere Transferrate erlaubt. Dies kann dadurch begründet werden, dass beim isochronen Transfer jeder Endpunkt pro Millisekunde garantiert mit einem Datenpaket versorgt wird und dass die Daten ohne Fehlerkorrektur verschickt werden, um eine konstante Transferrate zu garantieren, nicht aber eine garantierte Zustellung der Daten. Beim isochronen Transfer werden demnach alle 16 möglichen Endpunkte ein Mal pro Millisekunde mit einem Datenpaket von 64 Byte versorgt (oder ein Endpunkt mit einem Paket, das 1023 Byte groß ist (vgl. 2.1.10)), ein Bulk Endpunkt theoretisch mit 19 Paketen, die maximal 64 Byte groß sein können. Beachtet man nun die Tatsache, dass zwischen zwei SOF Paketen maximal 1023 Byte verschickt werden können (vgl. 2.1.6), wird klar, dass 19 Pakete mit einer Größe von 64 Byte diesen Wert überschreiten. Hinzu kommt noch, dass Bulk Endpunkte die niedrigste Priorität beim Senden haben und die Eigenschaft besitzen, Daten im Fehlerfall wiederholt zuzustellen (vgl. 2.1.10).

Abschließend kann also festgehalten werden, dass der gemessene Maximalwert (vgl. 6.1) einen guten Durchschnitt darstellt, jedoch unter Einbeziehung der Minimalwerte ebenso gesagt werden kann, dass der verwendete Treiber und Übertragungsmodus für viele kleine Datenpaketen genauso ungeeignet ist wie für die Einhaltung strenger Timingparameter.

7. Schluss

7.1. Fazit

Ziel dieser Bachelorarbeit war die Spezifikation und Realisierung eines USB-2-X Moduls auf dem 32 Bit Prozessor AT91SAM7X256 mit ARM7-Kern von Atmel auf einem Entwicklungsboard mit den notwendigen Schnittstellen. Bei der Entwicklung wurden folgende Anforderungen an das Ergebnis berücksichtigt:

1. Die Komponenten sollen eine hohe Wiederverwendbarkeit aufweisen.
2. Der RAM Verbrauch soll im Rahmen der Machbarkeit niedrig sein.
3. Der Code soll leicht wartbar sein.
4. Entscheidungen liegt stets ein Kompromiss zwischen Einfachheit und Umfang zu Grunde.
5. Die funktionalen Anforderungen aus Kapitel (3.1) sollen komplett erfüllt werden.

Um Punkt (1) und (3) zu realisieren, musste viel Arbeit in das Design der Software investiert werden, da Entwickler nur durch eine klare Softwarestruktur dafür begeistert werden können, die Software in anderen Projekten zu erweitern oder einzelne Komponenten wieder zu verwenden. Eine klare Struktur wurde erreicht, indem jede genutzte Hardwarekomponente durch eine Softwarekomponente repräsentiert wird. Ebenso wurde viel Wert auf die Kommentierung des Quellcodes und die Vergabe sinnvoller Variablen- und Funktionsnamen gelegt, um die Wartbarkeit zu erleichtern. Punkt (2) wurde umgesetzt, indem bei der Vergabe von Puffergrößen sowohl der vorhandene Speicher als auch ihre Zweckmäßigkeit berücksichtigt wurde. Punkt (4) wurde umgesetzt, indem Implementierungs- und Designentscheidungen immer die Ziele dieser Arbeit zu Grunde lagen. So wurden bereits vorhandene Komponenten, wie die Communication Device Class von Atmel, implementiert und wertvolle Entwicklungszeit eingespart. Gemessen an den implementierten und geforderten Features (5) wurden die Anforderungen an das USB-2-X Modul innerhalb der Firma TRINAMIC erfüllt und konnten unmittelbar einer praktischen Nutzung zugeführt werden.

7.2. Ausblick

In diesem Abschnitt der Bachelorarbeit wird ein Ausblick auf die zukünftige Verwendung des USB-2-X Moduls gegeben und Vorschläge zur Weiterentwicklung gemacht.

Das USB-2-X Modul wurde im Rahmen dieser Arbeit entwickelt, um Funktionstests an Modulen der Firma TRINAMIC durchzuführen, ohne für jeden Interface Typ einen eigenen Schnittstellenkonverter gebrauchen zu müssen. Durch den permanent wachsenden Kundenstamm der Firma TRINAMIC wird das USB-2-X Modul vermutlich auch in Zukunft seinen Platz in Laboren finden, in denen Funktionstests an der Tagesordnung stehen, da es ohne großen Aufwand installiert werden kann. Potential erweitert zu werden hat es jedoch nur bedingt, da der Treiber für das Entwicklungsboard durch seine Funktionalität stark eingeschränkt ist. Er unterstützt nur die Transferart Bulk, welche für die fehlerfreie Übertragung großer Datenmengen geeignet ist, nicht aber für die Einhaltung strenger Timingparameter (vgl. 6.2). Auch mit einem neuen Klassentreiber wären Änderungen auf der Seite des USB-2-X Moduls mit nicht geringem Aufwand verbunden, da die Funktionen der Communication Device Class zwar ohne Weiteres mit einer anderen Transferart genutzt werden können, jedoch Berechnungen angestellt werden müssen, um die Verarbeitungszeiten des Datenaufkommens zu ermitteln und entsprechende Änderungen vorgenommen werden müssen, um einen sicheren Datenverkehr zu gewährleisten.

Die Liste der möglichen Erweiterungen sieht wie folgt aus:

- Entwicklung eines neuen Klassentreibers, der eine der beiden periodischen Transferarten unterstützt.
- Implementierung von alternativen Interface Deskriptoren, um dem Host die Möglichkeit zu bieten, zwischen verschiedenen Endpunkteinstellungen zu wählen. Dies ist zum Beispiel notwendig, wenn auf Grund von Bandbreiteneinschränkungen die gewünschte Endpunktgröße innerhalb eines Interfaces nicht verfügbar ist.
- Erweiterung der USB-2-X Software, so dass ein sicherer Datenverkehr auch bei einem hohen Datenaufkommen mit strengen Timingparametern möglich ist.
- Implementierung eines Real-Time Operating System (RTOS) auf dem USB-2-X Modul, welches die einzelnen Tasks der Softwarekomponenten zeitlich konkret verwaltet.

A. USB-2-X Protokoll

Die Kommunikation zwischen Host PC und USB-2-X Modul ist über das USB-2-X Protokoll geregelt. Die folgenden Kriterien sind für einen gezielten Nachrichtenaustausch einzuhalten.

A.1. Kommunikation

Die Kommunikation zwischen Host PC und USB-2-X Modul findet über USB statt. Das Kommunikationsprotokoll ist ein Master-Slave Protokoll, wobei der Host PC die Rolle des Masters einnimmt und Kommandos an das USB-2-X Modul sendet. Das USB-2-X Modul ist der Slave, welcher entsprechend der Nachricht Kommandos ausführt und Antworten an den Master sendet. Jeder Nachrichteneingang wird jedoch vorher durch ein ACK oder NAK entsprechend der Prüfsumme quittiert. Die Kommunikation basiert auf Datenpaketen entsprechend dem folgendem Format:

Tabelle A.1.: Format der Datenpakete

Feldlänge	Inhalt
1 Byte Start Of Text (STX)	Kontrollzeichen
1 Byte Packet ID	Darf kein Kontrollzeichen sein.
2 Byte Payload Size n (n < 256)	1^{st} Byte = 0XF0 + High Nibble of n, 2^{nd} Byte = 0XF0 + Low Nibble of n
n Byte Payload Data	Darf kein Kontrollzeichen sein
2 Byte Checksum	Ergibt sich aus folgender Addition: C = Packet ID + Payload Size + Payload Byte[i] aus Payload Data. 1^{st} Byte = 0XF0 + High Nibble of C, 2^{nd} Byte = 0XF0 + Low Nibble of C
1 Byte End Of Text (ETX)	Kontrollzeichen

Die Payload Daten dürfen keine Kontrollzeichen aus der folgenden Tabelle enthalten. Sollen doch Kontrollzeichen innerhalb der Payload Daten enthalten sein, muss deren Vorgänger das Data Link Escape (DLE) Zeichen sein. Kontrollzeichen innerhalb der Payload Daten werden in die Berechnung der Prüfsumme mit einbezogen.

Tabelle A.2.: Kontrollzeichen

Control Character	ASCII Code (dezimal)
STX	2
ETX	3
ACK	6
DLE	16
NAK	21

A.2. Die verschiedenen Datenpakete

Tabelle A.3.: Kontrollzeichen

Kommando Packet ID, Datenrichtung: Host PC zu USB-2-X	Funktion	Antwort Packet ID, Datenrichtung: USB-2-X zu Host PC
0x33	CAN Write	
0x34	CAN Read	0x44
0x38	SPI Write	0x48
0x3B	RS485 Write	
0x3C	RS485 Read	0x4C
0x52	CAN Bitrate	
0x54	SPI Init	
0x55	RS485 Bitrate	

A.3. Die Datenpakete im Detail

1. **CAN Write Message**
   - USB Host sendet:
     - Packet ID: 0x33
     - Payload Data:
       * 4 Byte CAN Identifier
       * 0...8 Byte CAN Message Data
   - USB-2-X antwortet: ACK oder NAK

Tabelle A.4.: Standard CAN Identifier

31	30	29 . . . 11	10 . . . 0
IDE=0	RTR	don't care	Standard Identifier Bits

Tabelle A.5.: Extended CAN Identifier

31	30	29	28 . . . 0
IDE=1	RTR	don't care	Extended Identifier Bits

2. **CAN Read Message**
   - USB Host sendet:
     - Packet ID: 0x34
     - Payload Data:
       - No Payload Data
   - USB-2-X antwortet: ACK oder NAK
   - Dann werden die Daten im folgenden Paket übertragen:
     - Packet ID: 0x44
     - Payload Data:
       - 1 Byte Error Code
         - 0 = No Error
         - 1 = Software Receive Buffer Overflow
         - 2 = Hardware Receive Buffer Overflow
         - 3 = Software and Hardware Receive Buffer Overflow
       - 4 Byte Reception Time (evtl.)
       - 4 Byte CAN Identifier gemäß (A.4) oder (A.5)
       - 0...8 Byte CAN Message Data
   - USB Host antwortet: ACK oder NAK

3. **CAN Bitrate Message**
   - USB Host sendet:
     - Packet ID: 0x52
     - Payload Data:
       * 1 Byte CAN Bitrate
   - USB-2-X antwortet: ACK oder NAK

Tabelle A.6.: Extended CAN Identifier

Parameter für InitCan	Bitrate in kbit/s
2	20
3	50
4	100
5	125
6	250
7	500
8	800
9	1000

4. **SPI Write Message**
   - USB Host sendet:
     - Packet ID: 0x38
     - Payload Data:
       * Data Bytes to send
   - USB-2-X antwortet: ACK oder NAK
   - Dann sendet USB-2-X im folgenden Paket:
     - Packet ID: 0x48
     - Payload Data:
       * Empfangene Datenbytes, Anzahl gleich der gesendeten Anzahl Bytes
   - USB Host antwortet: ACK oder NAK

5. **SPI Init Message**
    - USB Host sendet:
        - Packet ID: 0x54
        - Payload Data:
            - No Payload Data
    - USB-2-X antwortet: ACK oder NAK

6. **RS485 Bitrate Message**
    - USB Host sendet:
        - Packet ID: 0x55
        - Payload Data:
            - 1 Byte Bitrate
                - 0 = 9600 bit/s
                - 1 = 14400 bit/s
                - 2 = 19200 bit/s
    - USB-2-X antwortet: ACK oder NAK

7. **RS485 Write Message**
    - USB Host sendet:
        - Packet ID: 0x3B
        - Payload Data:
            - Data Bytes to send
    - USB-2-X antwortet: ACK oder NAK

8. **RS485 Read Message**
    - USB Host sendet:
        - Packet ID: 0x3C
        - Payload Data:
            * No Payload Data
    - USB-2-X antwortet: ACK oder NAK
    - USB-2-X gibt dann die Daten aus dem Receive Buffer zurück (Dieses Paket wird auch bei leerem Receive Buffer gesendet):
        - Packet ID: 0x4C
        - Payload Data:
            * Bytes, welche aus dem Receive Buffer gelesen wurden.
    - USB Host antwortet: ACK oder NAK.

B. Hardware

Folgende Anforderungen wurden an den Entwicklungsrechner gestellt:

- Mindestens zwei USB Ports für die Verbindung des USB-2-X Moduls mit dem Joint Test Action Group (JTAG) Debugger und der normalen USB Verbindung zum Senden und Empfangen von Nachrichten.
- Ein Communication (COM) Port zum Testen der RS485 Funktionalität und der Debug-Ausgaben über die DBGU.
- Eine CAN Interface Karte zum Testen der CAN Funktionalität.

Folgende Hardware wurde für den Entwicklungsprozess benötigt:

- Der JTAG Debugger von Olimex, der auch zum Programmieren des USB-2-X Moduls verwendet wird.
- Das Entwicklungsboard AT91SAM7X von Olimex. Der Schaltplan des Entwicklungsboards kann auf der Seite http://olimex.com eingesehen werden.
- Ein Netzteil zum Betreiben des Entwicklungsboards.

C. Software

Da die Software des USB-2-X Moduls kostengünstig realisiert werden sollte, wurde bei der Entwicklung darauf geachtet, dass, soweit möglich, mit Open Source Programmen gearbeitet wurde.

- Eclipse mit C/C++ Development Tools (CDT) Plugin als freie Entwicklungsumgebung mit C Syntax.
- GNU Compiler Collection (GCC) als freier C Compiler.
- Open-On-Chip Debugger (OCD) als ARM-USB-OCD zum Debuggen und Programmieren des USB-2-X Moduls.
- CAN Monitor zum Testen der CAN Funktionalität.
- Windows Hyperterminal zum Testen der RS485 Funktionalität und für die Ausgabe von Nachrichten über die DBGU des USB-2-X Moduls.
- Microsoft Visual C++

D. Installation

Die Software des USB-2-X Moduls wird mit Hilfe der GCC und make kompiliert und anschließend mit dem Open-OCD in den Flash Speicher des Entwicklungsboards geschrieben. Hierfür wird der JTAG Debugger von Olimex und eine Konfigurationsdatei für den Flash-Vorgang benötigt. Die Konfigurationsdatei wird dem Open-OCD mit der Option "-f" als Parameter übergeben. Der JTAG Debugger wird mit dem JTAG Anschluss des Entwicklungsboards und einem USB Port des Entwicklungsrechners verbunden. Auf dem Entwicklungsrechner müssen Treiber für den Debugger vorhanden sein. Im Makefile müssen evtl. die Verzeichnisse, in denen sich der Quellcode befindet, angepasst werden.

Wie das USB-2-X System mittels dem Windows Hyperterminal und dem CAN Monitor getestet wird, kann Kapitel 6.1 entnommen werden.

E. Quellcode

Der gedruckten Ausgabe der Bachelorarbeit ist eine Compact Disc (CD)-ROM mit dem Quellcode der Software des USB-2-X Moduls, dem Quellcode der Testapplikation sowie dem Treiber des Entwicklungsboards beigelegt. Auf der CD befinden sich ebenfalls die LaTeXDokumente, eine PDF-Version der Bachelorarbeit und die Doxygen Dokumentation der Quellcodeeinheiten.

Verzeichnisstruktur der CD-ROM:

- **USB2X**
  - `UART` Implementierung des USARTs und des zugehörigen Interrupthandlers
  - `CAN` Implementierung der CAN Schnittstelle und des Interrupthandlers
  - `SPI` Implementierung der SPI Funktionalität
  - `CDC` Implementierung der Communication Device Class
  - `STARTUP` Implementierung des Startup Codes
  - `DBGU` Implementierung der DBGU
  - `INCLUDE` Software Application Programming Interface (API) Definitionen (AIC, SPI, ...), Definitionen für die Module
  - `DOC` Doxygen Dokumentation
- **TESTAPP**
  - `INF_DLL` Enthält die INF-Datei für den Treiber des Entwicklungsboards und den Treiber selbst.
  - `PC_APPLICATION` Enthält die PC Applikation zum Testen des USB-2-X Moduls inklusive Sourcecode und Doxygen Dokumentation.
  - `PCC_DLL` Enthält die Dynamic Link Library (DLL), welche die API für den Treiber anbietet, inklusive Sourcecode und Doxygen Dokumentation.
- **LATEX** Enthält das TEX Projektverzeichnis.

- **DOC** Enthält die Bachelorarbeit im PDF-Format als Beschreibung.

Literaturverzeichnis

[Ament 2005] Ament, Prof. Dr.-Ing. C.: *Eingebettete Systeme*. Website. 2005. – Available online at `http://www.imtek.de/systemtheorie/content/upload/vorlesung/2005/steuerungsentwurf-ss2005-01.pdf`; visited on May 2008.

[Atmel 2007] Atmel: *AT91 ARM7 Spezifikation*. Website. 2007. – Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc6120.pdf`; visited on March 2008.

[Axelson 2006] Axelson, Jan: *USB 2.0 Handbuch für Entwickler*. Vmi Buch, 2006. – ISBN 3826616901

[Bibliographisches-Institut und Brockhaus-AG 2007] Bibliographisches-Institut ; Brockhaus-AG: *Interrupt*. Website. 2007. – Available online at `http://lexikon.meyers.de/meyers/Interrupt`; visited on May 2008.

[Böbel 2005] Böbel, Dr. F.: *Hintergrundwissen Elektromotoren*. Website. 2005. – Available online at `http://www.torqeedo.com/de/hn/hintergrundwissen/motor-und-leistungselektronik.html`; visited on April 2008.

[Büch 2006] Büch, Christian: *SPI - Serial Peripheral Interface*. Universität Koblenz-Landau. 2006. – URL `http://www.uni-koblenz.de/~physik/informatik/MCU/SPI.pdf`

[Compaq u. a. 2000] Compaq ; Hewlett-Packard ; Intel ; Lucent ; Microsoft ; NEC ; Philips: *USB Spezifikation 2.0*. Website. 2000. – Available online at `http://www.usb.org/developers/docs/`; visited on March 2008.

[Elektronik-Projekt 2008] Elektronik-Projekt: *RS485 Bus*. Website. 2008. – Available online at `http://wiki.elektronik-projekt.de/w/index.php/RS485_Bus`; visited on March 2008.

[Etschberger 1994] Etschberger, Konrad: *CAN Controller Area Network. Grundlagen, Protokolle, Bausteine, Anwendungen*. Hanser Fachbuch, 1994. – ISBN 3446175962

[Flik 2005] Flik, Thomas: *Mikroprozessortechnik und Rechnerstrukturen*. Springer, 2005. – ISBN 3540222707

[Fowler 2003] Fowler, Martin: *UML konzentriert*. Addison-Wesley, 2003. – ISBN 978-3827321268

[mikrocontroller.net 2008] mikrocontroller.net: *AT91SAM7X256-Board*. Website. 2008. – Available online at `http://www.mikrocontroller.net/articles/Olimex_AT91SAM7X256-Board_SAM7-EX256`; visited on March 2008.

[Müller u. a. 2007] Müller, Marko ; Oster, Tanja ; Schweickart, Kai ; Volk, Bastian: *Schaltung zur Spannungsumsetzung mit geschalteten Kapazitäten*. Website. 2007. – Available online at `http://www.et.hs-mannheim.de/vorlesungen/MWA/SS2007/G1A_SS07.pdf`; visited on March 2008.

[Peacock 2007] Peacock, Craig: *USB in a NutShell - Making sense of the USB standard*. Website. 2007. – Available online at `http://www.beyondlogic.org/usbnutshell/usb-in-a-nutshell.pdf`; visited on March 2008.

[Pont 2001] Pont, Michael J.: *Pattern for Time - Triggered Embedded Systems*. Addison-Wesley Longman, 2001. – ISBN 0-201-33138-1

[roboternetz.de 2005] roboternetz.de: *Grundlagen*. Website. 2005. – Available online at `http://www.roboternetz.de/wissen/index.php/Kategorie:Grundlagen`; visited on March 2008.

[Sauter 2007] Sauter, Benedikt: *USB-Stack für Embedded-Systeme*. Fachhochschule Augsburg. 2007. – URL `http://www.ixbat.de//files/admin/projekte/usbport/sauter_thesis.pdf`

[Sauter 2008] Sauter, Benedikt: *USB Grundkurs*. Website. 2008. – Available online at `http://www.usb-projects.net/cwiki.php?page=Grundkurs`; visited on March 2008.

[Thomas 2008] Thomas, Martin: *Arm Projects*. Website. 2008. – Available online at `http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects/`; visited on March 2008.

[Wedemeyer 1999] Wedemeyer, Thomas: *Grundlegende Informationen zum CAN-Bus*. Universität Bremen. 1999. – URL `http://www.thomas-wedemeyer.de/elektronik/CAN.PDF`

[wikipedia.de 2008] wikipedia.de: *Enzyklopädie*. Website. 2008. – Available online at `http://de.wikipedia.org/wiki/Hauptseite`; visited on March 2008.

[Zimmer 2006] Zimmer, Marcel: *Entwicklung eines Antikollisionssystems für Kraftfahrzeuge mit MARMOT*. Universität Kaiserslautern. 2006. – URL `http://kluedo.ub.uni-kl.de/volltexte/2006/2013/pdf/ausarbeitung.pdf`

Glossar

AIC	Der *AIC* kann bis zu 32 priorisierte Interruptquellen verwalten und wurde entwickelt, um den Softwareoverhead innerhalb eines Embedded Systems zu senken. Bei den Interruptquellen kann es sich sowohl um interne als auch um externe Interruptquellen handeln, die nach ihrer Priorität abgearbeitet werden.
*AT*91*SAM7X*256	Der *AT91SAM7X256* ist ein Flash Mikrocontroller mit integrierter Ethernet, USB, SPI und CAN Schnittstelle und basiert auf dem 32 Bit ARM7TDMI Reduced Instruction Set Computer (RISC) Prozessor. Er verfügt über einen 256 kB Flash Speicher und einen 64 kB RAM.
DPR	Ein *DPR* verfügt über zwei getrennte Speicherbänke, die simultane Lese- und Schreibzugriffe ermöglichen.
Embedded System	Unter eingebetteten Systemen versteht man Hard- und Softwaresysteme, die eingebettet in umgebende technische Systeme komplexe Steuerungs-, Regelungs- und Datenverarbeitungsaufgaben übernehmen. Aus Software-Engineering-Sicht ist ein *Embedded System* ein reaktives Softwaresystem, welches die es umgebende Hardware kontrolliert. [1]
Interrupt	Als *Interrupt* definiert man das Aussetzen des Programmablaufs auf Grund einer besonderen Unterbrechungsaufforderung durch eine Anweisung (Software-Interrupt - synchron) oder durch ein Systemteil (Hardware-Interrupt - asynchron; z. B. Ein- oder Ausgabegerät) zum Zwecke der Koordinierung bestimmter Operationen des Prozessors. Im Falle eines Interrupts werden die Inhalte des Befehlszählers und der Register gesichert. Nach der Abarbeitung des Interrupthandlers wird das ausgesetzte Programm an der Stelle fortgesetzt, an der es unterbrochen wurde. [2]

[1] Vgl. Ament (2005)
[2] Vgl. Bibliographisches-Institut und Brockhaus-AG (2007)

MOB	Bei einem *MOB* handelt es sich um das Datenregister eines CAN Controllers. In ihm wird ein kompletter CAN Frame gespeichert. Identifiziert wird jeder MOB über seinen Identifier.
PIO Controller	Der *PIO Controller* verwaltet bis zu 32 IO Leitungen. Jede dieser Leitungen kann entweder der Peripherie des Entwicklungsboards zugewiesen werden oder aber über den Programmable IO (PIO) Controller für andere Zwecke genutzt werden, wie z.B. dem Ansteuern einer Diode.
Polling	*Polling* bezeichnet den Vorgang einer zyklischen Abfrage von Informationen. Bei den Informationen kann es sich beispielsweise um den Status bestimmter Hardwareelemente handeln.
Ringpuffer	Die Besonderheit des *Ringpuffers* ist, dass er eine feste Größe besitzt. Die Elemente des Ringpuffers werden über einen Index angesprochen. Erreicht der Index die Obergrenze, springt dieser auf Null zurück. Haben Lese- und Schreibindex den gleichen Wert, wird der Puffer als leer angesehen. Wenn 'Write+1' und 'Read' identisch sind, wird der Puffer als voll angesehen.
Superloop	Eine *Superloop* ist eine Programmstruktur, welche aus einer Endlosschleife besteht, die alle wesentlichen Aufgaben des Systems beinhaltet.
User Interface	Das *User Interface* ermöglicht es dem Benutzer mit dem USB-2-X Modul über ein einfaches Menü zu interagieren. Über das Menü kann er verschiedene Nachrichtentypen versenden und die Reaktion des USB-2-X Moduls abfragen. Treten Fehler während der Nachrichtenübertragung auf, werden diese dem Benutzer über das User Interface mitgeteilt.

Index

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 22. September 2008
Ort, Datum

Unterschrift