Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Simon Hillebrecht

VHDL Design und Implementierung eines CAN-Controllers in einem FPGA

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Simon Hillebrecht

VHDL Design und Implementierung eines CAN-Controllers in einem FPGA

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. rer. nat. Michael Schäfers
Zweitgutachter : Prof. Dr. rer. nat. Hans H. Heitmann

Abgegeben am 2. Januar 2010

Simon Hillebrecht

Thema der Bachelorarbeit

VHDL Design und Implementierung eines CAN-Controllers in einem FPGA

Stichworte

VHDL, CAN, Controller Area Network, FPGA, ISE, ModelSim

Kurzzusammenfassung

In dieser Bachelorarbeit wird ein CAN-Controller in VHDL designed und in einem FPGA implementiert.

Simon Hillebrecht

Title of the paper

VHDL-design and implementation of an CAN-controller for usage in an FPGA

Keywords

VHDL, CAN, Controller Area Network, FPGA, ISE, ModelSim

Abstract

This bachelor thesis descripes the design of an CAN-controller in VHDL and its implementation in an FPGA.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

In diesem Kapitel wird erklärt, was die Motivation ist und die Ziele dieser Arbeit sind.

1.1 Motivation

CAN[1] ist ein von Bosch entwickeltes Bus-System. Es wurde ursprünglich für die Automobilindustrie entwickelt, fand aber auf Grund seiner Eigenschaften, auch den Weg in viele andere Bereiche. CAN wird auch an der HAW, sowohl in Lehrveranstaltungen als auch in Projekten, eingesetzt. Zur Zeit wird an der HAW ein SoC[2] auf Basis eines FPGAs[3] entwickelt. Für dieses SoC befinden sich im Moment mehrere Teilsysteme in Entwicklung. FPGAs sind rekonfigurierbare Hardwarebausteine deren Verhalten mittels einer Hardwarebeschreibungssprache festgelegt werden kann. FPGAs werden hauptsächlich für den Bau von Prototypen und für Kleinserien eingesetzt. VHDL[4] ist eine solche Hardwarebeschreibungssprache.

1.2 Zielsetzung

Der in dieser Arbeit entwickelte CAN-Controller soll als eines dieser Teilsysteme in diesem SoC dienen. An der HAW werden FPGAs aber für eine Vielzahl von anderen Projekten eingesetzt, daher soll der hier entwickelte CAN-Controller möglichst modular aufgebaut sein, um so leicht anpassbar und erweiterbar zu sein. Um möglichst flexibel bei der Wahl des FPGA zu bleiben soll dieser Controller nicht an einen speziellen FPGA angepasst werden und auch keine Hersteller-spezifischen Bibliotheken verwendet werden.

[1] Controller Area Network
[2] System on Chip
[3] Field Programmable Gate Array
[4] Very high speed integated circuit Hardware Description Language

2 Analyse

In diesem Kapitel soll analysiert werden, welche Standards für CAN existieren und was diese umfassen, wie der CAN-Controller in das SoC integriert werden kann, ob es bereits fertige Lösungen für FPGA gibt und was diese ggf. können. Weiterhin soll analysiert werden welche Freiheitsgrade es bei der Umsetzung gibt.

2.1 Verfügbare Standards

Zu CAN existiert zum einen eine Spezifikation von Bosch und zum anderen ein ISO-Standard.

2.1.1 ISO-Standard

Der ISO-Standard ist in mehrere Teile aufgeteilt, welche unterschiedliche Aspekte von CAN beschreiben. Diese sind:

- **11898-1**: Data Link Layer and physical signalling
- **11898-2**: High-speed medium access unit
- **11898-3**: Low-speed, fault-tolerant, medium-dependent interface
- **11898-4**: Time-triggered communication
- **11898-5**: High-speed medium access unit with low-power mode

2.1.2 Bosch Spezifikation

Bosch stellt Spezifikation in der Version 2.0 zum CAN-Protokoll auf seiner Website zur Verfügung. Diese Spezifikation ist unterteilt in Teil A und Teil B. Mitunter werden diese auch nur als 2.0A und 2.0B bezeichnet. Teil A stellt eine Überarbeitung der Spezifikation 1.2 dar. Teil B beinhaltet den kompletten Teil A sowie eine Erweiterung um ein zusätzliches Nachrichtenformat. Die CAN-Spezifikation 2.0B diente als Vorlage für den ISO-Standard 11898-1.

2.1.3 Wahl der Spezifikation

Da der ISO-Standard nicht unentgeltlich und auch nicht frei verfügbar ist fällt die Wahl hier auf die CAN-Spezifikation 2.0B. Diese stellt die aktuellste frei verfügbare Version dar. Der ISO-Standard ist allerdings aktueller und es kann nicht ausgeschlossen werden, dass dieser Änderungen gegenüber der Spezifikation 2.0B enthält. In der Literatur werden allerdings keine Änderungen erwähnt.

2.2 Aufbau eines System on (a) Chip

Ein SoC besteht im allgemeinen aus einem Prozessor, Speicher und einer Vielzahl von Peripheriegeräten. Die Peripheriegeräte und der Prozessor kommunizieren dabei über ein Bus-System. Abbildung 2.1 zeigt, stark vereinfacht, wie ein solches SoC aufgebaut sein kann und wo ein CAN-Controller dort platz finden könnte.

Um dieses Peripherie-Bus-System zu nutzen, benötigt der CAN-Controller ein Interface. Für dieses wurde ein Baukasten verwendet, welcher von Herrn Alexander Pautz im Rahmen einer Bachelorarbeit entwickelt wurde. Als CPU soll ein ARM9 zum Einsatz kommen. Weiterhin benötigt der CAN-Controller noch ein Interface nach Außen um mit einem CAN-Busmedium verbunden zu werden. Dies ist ebenfalls stark vereinfacht in Abbildung 2.1 verdeutlicht.

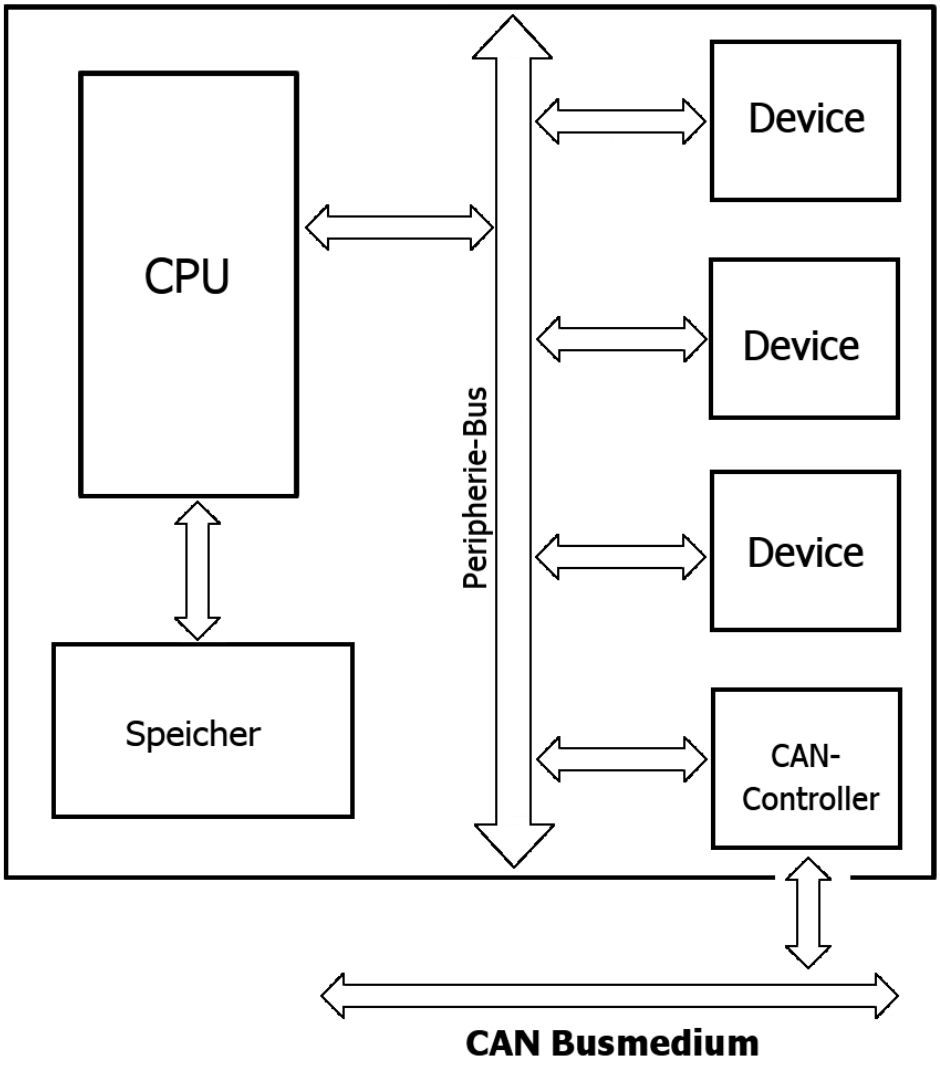


Abbildung 2.1: Beispielhafter Aufbau eines System on (a) Chip

2.3 Peripherie-Bus-Interface

Das Interface stellt eine Menge an vorgefertigten Registern zur Verfügung sowie einen Generator mit dem diese leicht im Code erzeugt werden können.

Zur Verfügung stehen:

- **Datenregister**

- **Kontrollregister**
- **Statusregister**
- **Counterregister**
- **Pufferregister**
- **Main-Interruptregister**

Diese Register, mit Ausnahme des Main-Interruptregisters, stehen in unterschiedlichen Versionen zur Verfügung. Sie sind unterteilt in Register, die von der Peripherie aus beschrieben werden können und Register, die vom Prozessor aus gesetzt werden können. Weiterhin gibt es Unterschiede im Funktionsumfang dieser Versionen von Registern.

2.4 Verfügbare CAN IP-Cores

In diesem Abschnitt soll zunächst analysiert werden welche existierenden Lösungen, sogn. IP-Core[1] es am Markt gibt und ob diese verwendet werden können.

2.4.1 kommerzielle IP-Cores

Es gibt am Markt einige Hersteller, die CAN IP-Cores für FPGA anbieten. CAST(CAST), Xilinx(Xilinx) und auch Bosch(DCAN) selber bieten z.B. IP-Cores an. Diese IP-Cores haben allerdings den Nachteil, dass diese kostenpflichtig sind meist nur in Form von Netzlisten oder verschlüsselter Code vorliegen. Eine spätere Erweiterung oder Modifikation des Controllers ist so nicht möglich. Weiterhin sind einige dieser IP-Cores auf spezielle FPGAs abgestimmt. So bietet Bosch nur Lösungen für Altera FPGAs an. Xilinx bietet nur Lösungen für eigene FPGAs an.

2.4.2 freie IP-Cores

Opencores(OpenCores) bietet zwei CAN-Controller als Opensource IP-Cores an. Eine Version in der Hardwarebeschreibungssprache Verilog und eine in VHDL. Die Verilog-version ist allerdings nur sehr schlecht bis gar nicht dokumentiert und der generelle Aufbau des Codes sehr unübersichtlich. Die VHDL-version ist eine durch XHDL automatisch übersetzte Fassung der Verilog-version. Zu dem wird diese Version seid November 2007 nicht mehr gepflegt.

[1] Intellectual Property Core

2.4.3 Neuentwicklung

Es zeigt sich, dass keine der existierenden Lösungen an Anforderungen gerecht werden. Die kommerziellen Lösungen sind mit hohen Kosten verbunden, zumeist closed-source und herstellerspezifisch. Die freien Projekte sind unzureichend dokumentiert und strukturiert. Eine Anpassung dieser Projekte an die Anforderungen würde zu viel Zeit in Anspruch nehmen.

2.5 Lizenz

CAN ist von Bosch entwickelt und patentiert. Zusätzlich zu erworbenen IP-Cores, als auch für selbst entwickelte CAN-Module muss eine CAN-Protokoll-Lizenz erworben werden. Bosch bietet eine Lizenz für ASIC[2]-Erstellung und FPGA-Massenprogrammierung sowie eine Lizenz für FPGA-Programmierung an.

2.6 Das CAN-Protokoll nach der Spezifikation 2.0B

In diesem Abschnitt soll das CAN-Protokoll nach der Spezifikation 2.0B analysiert werden.

2.6.1 Umfang der Spezifikation

Die Spezifikation ist unterteilt in Physical-Layer und Data-Link-Layer. Der Physical-Layer definiert wie Signale übertragen werden. Der Physical-Layer umfasst das Bit-Timing, das Bit-Encoding und die Synchronisaton. Der Physical-Layer macht keine Angaben zur Busankopplung oder zum Bustreiber. Dies ist in dieser Spezifikation explizit frei gelassen worden. „...within the CAN specifcations, the characteristics of the driver/receiver of the physical layer are not defined, so that the transport medium and the signal levels can be optimized for any given application.“ (Paret, 2007, S.28)

Der Data-Link-Layer ist unterteilt in Medium-Access-Control-Layer(MAC) und Logical-Link-Control-Layer(LLC). Der MAC-Layer stellt den Kern des CAN-Protokolls dar. Er beschreibt das Message-Framing, die Arbitrierung sowie die Fehlererkennung und Signalisierung. Der LLC-Layer beschreibt das Message-Filtering, die Overload-Notification und das Recovery Management.

[2] application-specific integrated circuit

2.6.2 Buszugriffsverfahren

CAN ist ein asynchroner Bus d.h. dass jeder sendebereite Busteilnehmer senden darf sobald der Bus frei ist. Es ist also möglich, dass mehrere Sender gleichzeitig senden und sich dabei überschreiben. Es also zu Kollisionen auf dem Bus kommt. Es gibt mehrer Verfahren mit solchen Kollisionen umzugehen. Ethernet z.B. benutzt ein Verfahren, welches als CSMA/CD(Carrier Sense Multiple Access / Collision Detection) wird. Dabei sendet jede Station ihre Nachricht und liest diese nach einer bestimmten Zeit, der sogn. Roundtriptime wieder vom Bus ein. Empfängt sie die Nachricht dabei wieder so, wie diese gesendet wurde, ist alles in Ordnung. Sollte die Nachricht allerdings verändert zurückkommen so liegt eine Kollision vor. Die Station wartet daraufhin eine Zeitspanne ab und versucht danach die Nachricht nochmal zu senden.

CAN löst dieses Problem durch eine Arbitrationsphase. In dieser Phase handeln alle sendebereiten Stationen untereinander aus welche Station den Zugriff auf den Bus erhält. Dabei löst die Station, die als erste zu senden beginnt diese Phase aus; alle anderen Stationen synchronisieren sich auf diese. Den Zugriff auf dem Bus erhält dabei die Station mit der höchsten Priorität. Diese Priorität wird durch den **Identifier** der Nachricht festgelegt.

Dieses Buszugriffsverfahren wird in der Literatur mal als CSMA/CA, CSMA/CR oder CSMA/-BA bezeichnet. Die Erweiterungen hinter dem / stehen dabei für:

CA: Collision Avoidence - Kollisionsvermeidung

CR: Collision Resolution - Kollisionsauflösung

BA: Bitwise Arbitration - bitweise Arbitrierung

In der Spezifikation von Bosch wird allerdings kein Name für dieses Verfahren angegeben.

2.6.3 Bitkodierung und Bitinterpretation

Die Bitkodierung bei CAN erfolgt in NRZ-Codierung(Non-Return-to-Zero), der Pegel wird also über die gesamte Dauer eines Bits beibehalten. Im Gegensatz dazu sei die sogn. Manchesterkodierung erwähnt, wie sie bei Ethernet Verwendung findet. Bei Manchester findet innerhalb eines jeden Bits ein Flankenwechsel statt. Bild 2.2 zeigt den Unterschied zwischen NRZ- und Manchesterkodierung.

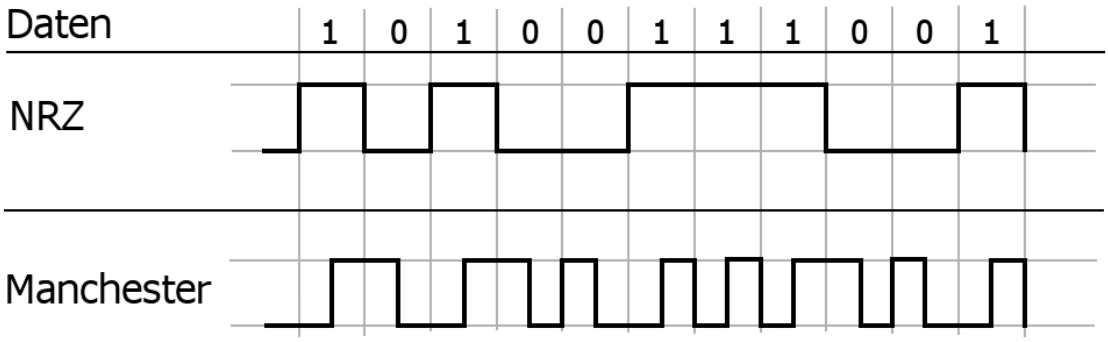
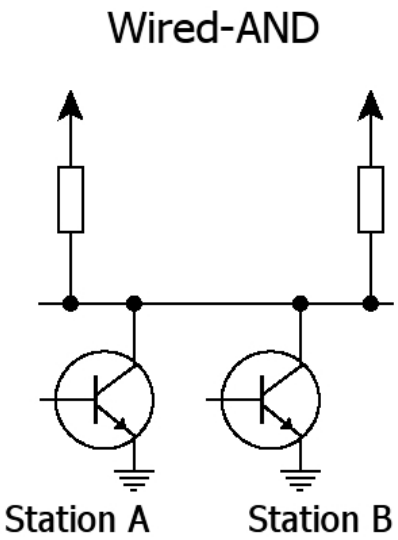


Abbildung 2.2: NRZ: 1 entspricht einem hohen Pegel; 0 einem niedrigen Pegel
Manchester: 1 entspricht einem Wechsel von niedrigem zu hohem Pegel; 0 einem Wechsel von hohem zu niedrigem Pegel

CAN interpretiert die Signale auf dem Bus als **dominant** und **rezessiv**. CAN geht dabei von der Wired-AND-Verknüpfung als Open-Collector-Schaltung aus vergleiche Abbildung 2.3. Dabei werden 0 als dominant und 1 als rezessiv angesehen(vgl. CAN 2.0, S.8).

Hierdurch ist es möglich, dass Signale auf dem Bus kollidieren und trotzdem ein logischer Buspegel erhalten bleibt. Bei Manchesterkodierung wäre dies nicht möglich.



Station A	Station B	Bus
dominant	dominant	dominant
dominant	rezessiv	dominant
rezessiv	dominant	dominant
rezessiv	rezessiv	rezessiv

dominant = 0
rezessiv = 1

Abbildung 2.3: Bit-Interpretation durch Wired-AND-Verknüpfung

2.6.4 Frameformate

Es existieren zwei verschieden Frameformate. Diese unterscheiden sich in der Länge des Identifiers:

- Standard-Frame: Der Identifier ist 11 Bit lang.
- Extended-Frame: Der Identifier ist 29 Bit lang.

2.6.5 Frametypen

CAN kennt vier Arten von Nachrichten, den sogn. Frames(engl. Rahmen oder Telegramm). Diese erfüllen unterschiedliche Aufgaben.

- **Dataframe**(Datentelegramm) überträgt die Daten von einem Sender zu den Empfängern.
- **Remoteframe**(Anforderungstelegramm) wird von einer Station gesendet, um ein Dataframe mit gleichem IDENTIFIER anzufordern.
- **Errorframe**(Fehlertelegramm) wird von jeder Station, die einen Fehler erkennt gesendet.
- **Overloadframe**(Überlasttelegramm) dient dazu eine Übertragung hinauszuzögern, um einen Delay zwischen vorhergehendem und nachfolgendem Data- oder Remoteframe einzufügen.

Data- und Remoteframe können sowohl im Standard-Frame- als auch im Extended-Frame-Format verwendet werden. Data- oder Remoteframes werden immer durch einen **Interframespace** von anderen Frames, egal ob Dataframe, Remoteframe, Overloadframe oder Errorframe, getrennt. Für Errorframes und Overloadframes gilt dies nicht. Auf ein Erroframe kann direkt ein Errorframe oder ein Overloadframe folgen. Für das Overloadframe gilt das selbe.

2.6.6 Dataframe und Remoteframe

Data- und Remoteframe sind in Bitfelder, den Fields(engl. Felder) unterteilt. Siehe Abbildung 2.4

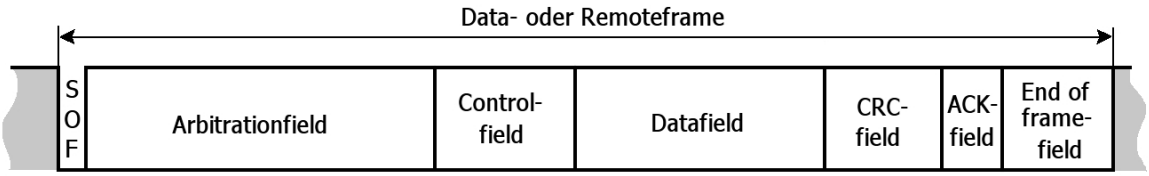


Abbildung 2.4: Bitfelder von Data- und Remoteframe

Diese sind:

- **Start-of-frame-bit**(Telegrammanfangs-bit) S. 19
- **Arbitrationfield**(Vorrangfeld) S. 19
- **Controlfield**(Kontrollfeld) S. 20
- **Datafield**(Datenfeld) S. 21
- **CRC-field**(CRC-feld) S. 21
- **ACK-field**(Quittungsfeld) S. 22
- **End of frame-field**(Telegrammendefeld) S. 22

Diese Felder können aus einem oder mehreren Bit zusammengesetzt sein. Data- und Remoteframe unterscheiden sich hinsichtlich des Datafields. Beim Dataframe 2.5 ist dieses Feld optional. Es kann dort 1 bis 8 Byte umfassen oder auch ganz entfallen. Das Remoteframe 2.6 hingegen enthält dieses Feld niemals.

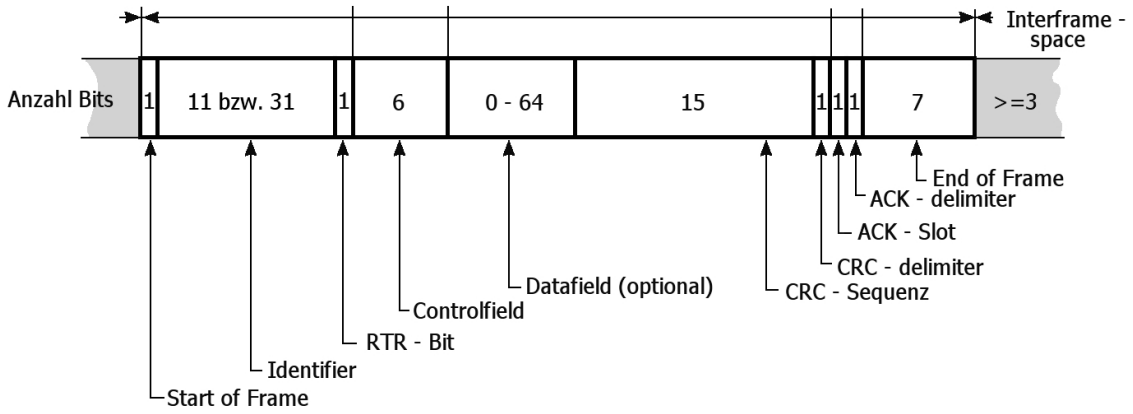


Abbildung 2.5: Dataframe

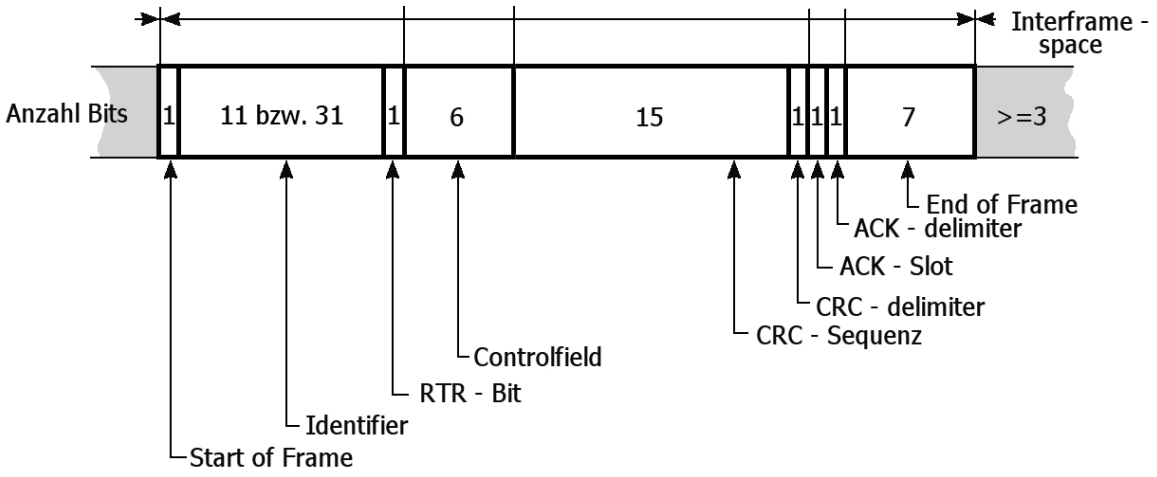


Abbildung 2.6: Remoteframe

Start-Of-Frame-bit

Das Start-Of-Frame-field besteht aus einem einzigen dominanten Bit. Durch dieses Bit wird der Beginn eines neuen Frames gekennzeichnet.

Arbitrationfield

Das Arbitrationfield setzt sich aus dem Identifier des Frames und dem RTR-Bit(RemoteTransmissionRequest-Bit) zusammen. Das RTR-Bit gibt an, ob es sich um ein Dataframe oder ein Remoteframe handelt. Ein **dominantes** RTR-Bit steht dabei für ein Dataframe; ein **rezessives** RTR-Bit für ein Remoteframe.

Der Identifier kann sich dabei, in Abhängigkeit vom Format des Frames, aus mehreren Teilen zusammensetzen(siehe Abbildung 2.7). Im Standardformat umfasst dieser 11 Bit. Im 2.0B-Extendedformat setzt sich dieser aus zwei Identifiern zusammen:

- **Base-ID**(Basis-identifier) umfasst 11 Bit
- **Extended-ID**(Erweiterter Identifier) umfasst 18 Bit

Base-ID und Extended-ID werden durch 2 Bit getrennt. Zu einen das SRR-Bit (SubstitudeRemoteRequest-Bit) und zum anderen das IDE-Bit(IdentifierExtension-Bit).

Das SRR-Bit ist immer **rezessiv**. Das IDE-Bit ist im Extendedformat ebenfalls immer **rezessiv**.

Die Gesamtlänge des Arbitrationfields beträgt 12 Bit im Standardformat, sowie 31 Bit im Extendedformat.

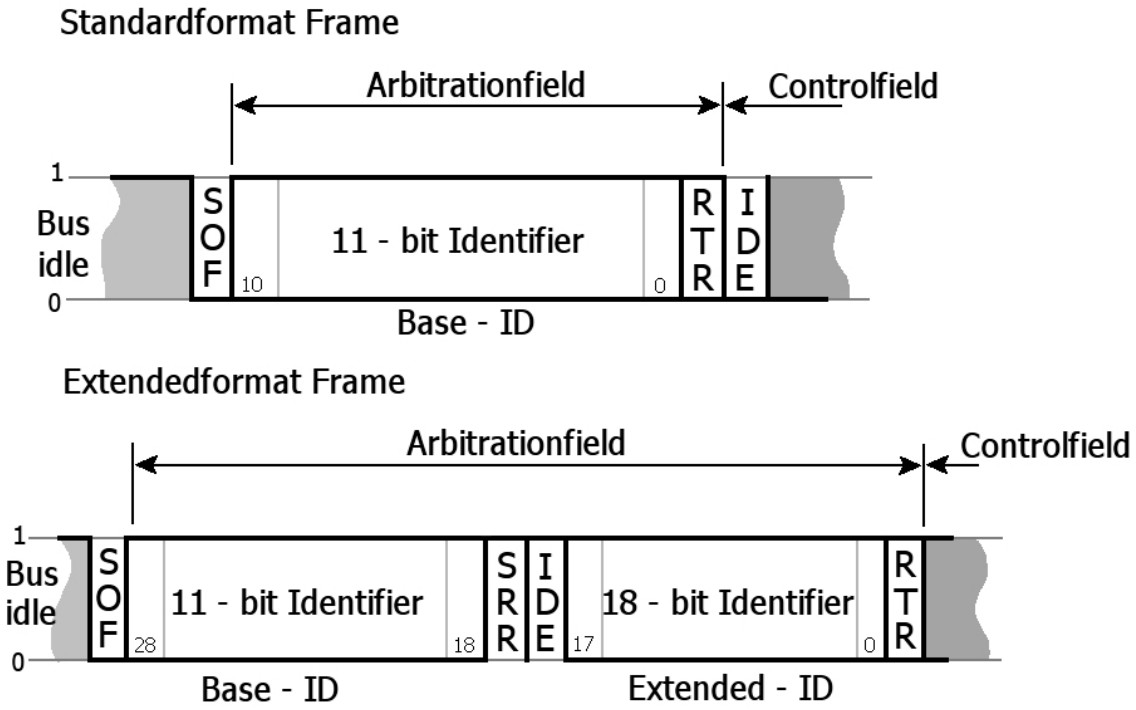


Abbildung 2.7: Arbitrationfield

Controlfield

Das Controlfield besteht aus 6 Bit. Dieses Feld ist in zwei Teile unterteilt.

reservierte Bits umfasst die Bits R1 und R0.

Datalengthcode umfasst 4 Bit.

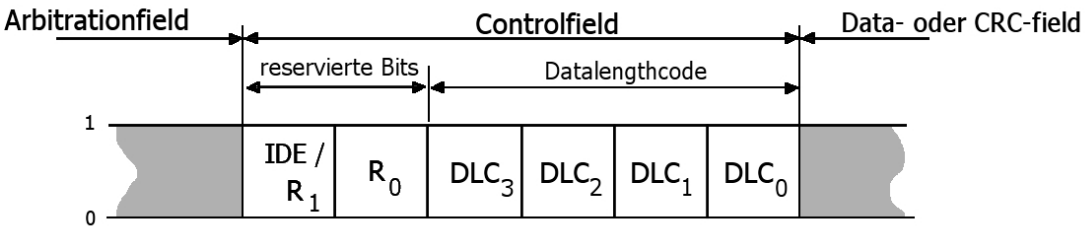


Abbildung 2.8: Controlfield

Im Standardformat ersetzt das IDE-Bit das Bit R1. Im Extendedformat existiert das R1-Bit noch. Die reservierten Bits müssen immer als **dominante** Bits übertragen werden. Allerdings toleriert ein CAN-Controller hier alle Bitkombinationen ohne dies als Fehler anzusehen.

Der Datalengthcode(Datenlängeangabe) gibt beim Dataframe an, wie viele Bytes das Datafield umfasst. Beim Remoteframe kann dort angegeben werden, wie viele Bytes man anfordern möchte. Dabei bildet DLC3 das MSB(Most Significant Bit - höchstwertigstes Bit) und DLC0 das LSB(Least Significant Bit - niederwertigstes Bit). Wird ein Zahlenwert von 0 im Datalengthcode angegeben entfällt das Datafield beim Dataframe.

Wichtig: Ein CAN-Controller kann nur max. 8 Byte in einem Frame übertragen. In 4 Bit lassen sich allerdings die Zahlenwerte von 0 bis 15 kodieren. Ein CAN-Controller akzeptiert aber auch Werte größer 8. In diesem Fall kann der Controller eine Warning ausgeben. Er wird dann das Datafield auf eine Länge von 8 Byte begrenzen.

Datafield

Das Datafield beinhaltet die Nutzdaten des Dataframes. Es besteht, wenn es denn nicht ausgelassen wird, aus mindestens einem bis maximal 8 Bytes. Die Übertragung beginnt dabei mit $Byte_0$ und endet mit $Byte_{Datalengthcode\ -\ 1}$, wenn Datalengthcode kleiner oder gleich 8 ist, ansonsten mit $Byte_7$. Die Übertragung der einzelnen Bytes erfolgt dabei MSB to LSB. Siehe Abbildung 2.9.

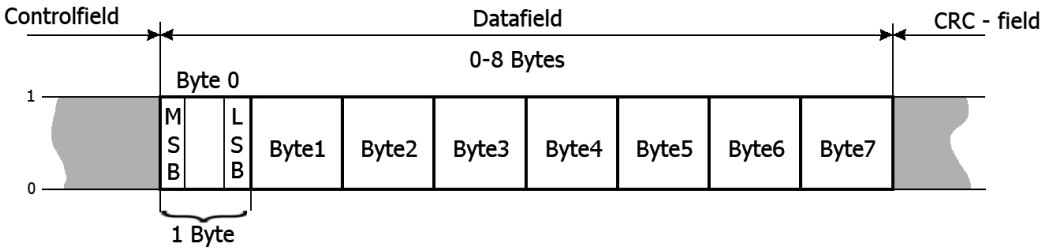


Abbildung 2.9: Datafield

CRC-field

Das CRC-Feld besteht aus der 15 Bit breiten CRC-Checksumme und dem CRC-Delimiter. Der **CRC-Delimiter** muss **immer rezessiv** sein. Ein dominantes Bit an dieser Position ist als Fehler zu werten. Abbildung 2.10 zeigt das CRC-field.

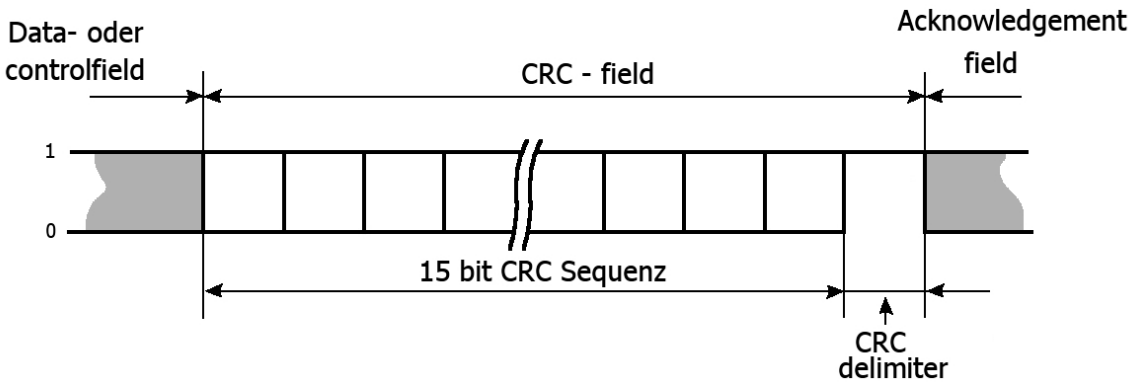


Abbildung 2.10: CRC-field

Acknowledgementfield / ACK-field

Das Acknowledgementfield besteht aus 2 Bit. Dem ACK-Slot und dem ACK-Delimiter. Siehe Abbildung 2.11. Der **ACK-delimiter** muss immer **immer rezessiv** sein.

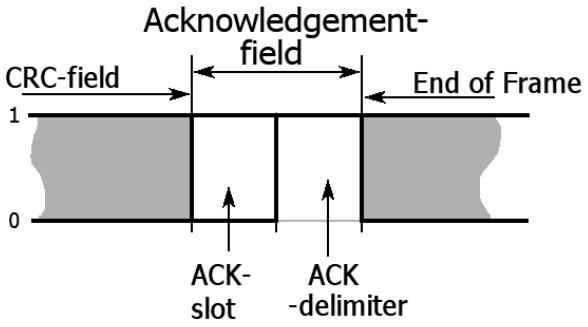


Abbildung 2.11: Acknowledgementfield

End-Of-Frame-Field

Das End-of-frame-field bildet den Abschluss des Frames und besteht aus 7 rezessiven Bit. Abbildung 2.12 zeigt den Aufbau dieses Feldes.

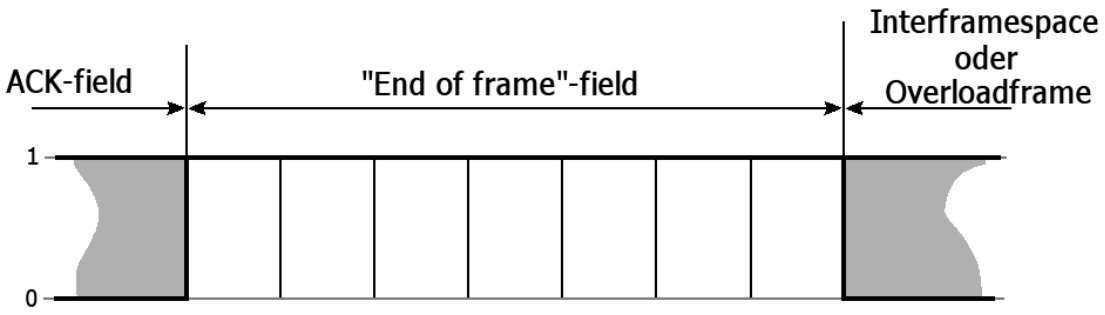


Abbildung 2.12: End of frame-field

2.6.7 Errorframe

CAN hat zwei verschiedene Errorframes. Das Active-Errorframe und das Passive-Errorframe. Das Faultconfinement(siehe **Faultconfinement** 2.6.10) gibt vor, wann welches davon verwendet wird. Beide Errorframes sieht in zwei Teile unterteilt. Dem **Errorflag** und dem **Errordelimiter**. Active-Errorframe und Passive-Errorframe unterschieden sich durch den Aufbau des Errorflags.

Active-Errorframe

Das Errorflag des Active-Errorframes besteht aus 6 aufeinander folgenden **dominanten** Bits. Abbildung 2.13 zeigt ein solches Active-Errorframe.

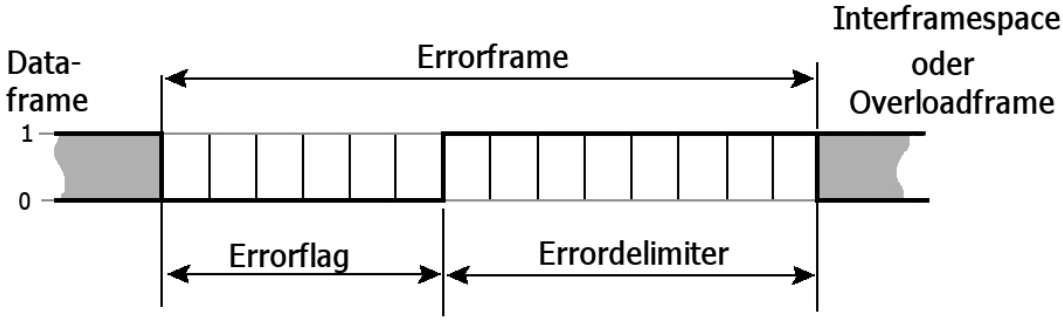


Abbildung 2.13: Active-Errorframe

Passive-Errorframe

Das Errorflag des Passive-Errorframes besteht aus 6 aufeinander folgenden **rezessiven** Bits. Abbildung 2.14 zeigt ein komplettes Passive-Errorframe. Während des Sendens eines Passive-Errorflags schaltet eine Station die Sendeleitung auf rezessiv und erwartet eine Folge von 6 aufeinander folgenden Bits gleicher Polarität.

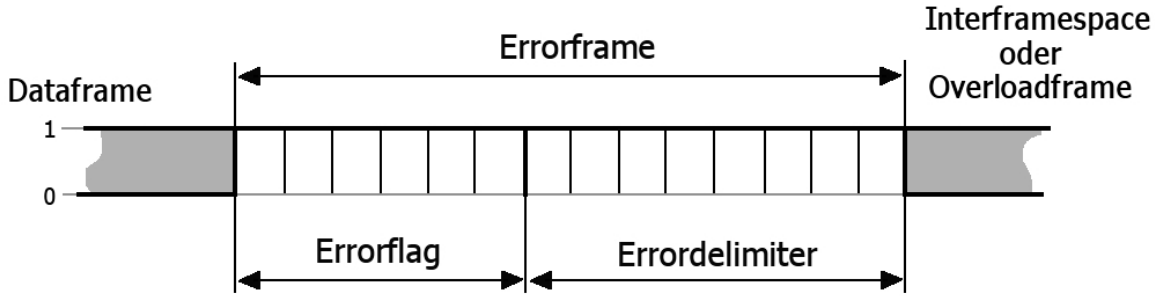


Abbildung 2.14: Passive-Errorframe

Errordelimiter

Der Errordelimiter besteht aus 8 aufeinander folgenden rezessiven Bits. Eine Station die einen solchen Delimiter sendet legt eine rezessives Bit auf den Bus und wartet bis ein rezessives Bit empfangen wird. Danach wird sie weitere 7 rezessive Bits senden um den Delimiter zu komplettieren.

2.6.8 Overloadframe

Das Overloadframe hat die gleiche Form wie das Active-Errorframe. Der Unterschied zwischen diesen Frames besteht darin, wann dieses Frame gesendet werden darf und welche Auswirkungen dieses auf das Faultconfinement hat. Abbildung 2.15 zeigt das Overloadframe.

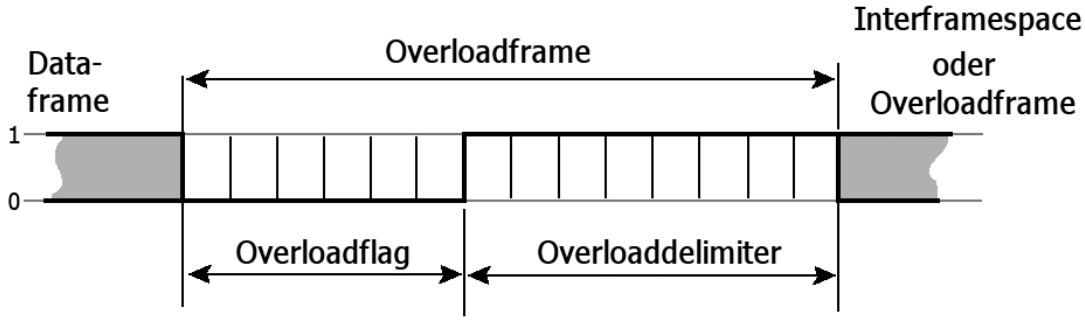


Abbildung 2.15: Overloadframe

2.6.9 Interframespace

Der Interframespace besteht aus zwei oder drei Feldern. Dies ist wiederum vom Faultconfinement abhängig. Diese Felder sind:

- **Intermission-field**(Unterbrechungsfeld)
- **Suspended-Transmission-field**(Ausgesetzte-Übertragung-feld)
- **Bus-Idle-field**(Bus-Ungenutzt-feld)

Intermission-field

Das Intermission-field besteht aus 3 aufeinander folgenden rezessiven Bits. Während dieses Feld übertragen wird darf keine Station damit beginnen aktiv ein Dataframe oder eine Remoteframe zu senden. Es ist allerdings erlaubt ein Overloadframe zu senden. Eine Station die innerhalb der ersten zwei Bit dieses Feldes ein dominantes Bit erkennt wird dieses als Begin eines Overloadframes interpretieren und darauf hin mit dem nächsten Bit selber ein Overloadframe senden.

Wird im dritten Bit des Intermission-fields ein dominantes Bit erkannt, so wird dies als Start-Of-Frame-field interpretiert. Eine sendebereite Station wird darauf hin, mit dem nächsten Bit anfangen, mit der Übertragung des ersten Bits des Identifiers ihrer Nachricht beginnen ohne selber ein Start-Of-Frame gesendet zu haben.

Eine sendebereite Station, die im dritten Bit des Intermission-field ein rezessives Bit festgestellt hat wird, mit dem nächsten Bit ein Start-Of-Frame senden und so die Übertragung ihrer Nachricht einleiten.

Suspended-Transmission-field

Das Suspended-Transmission-field besteht aus 8 rezessiven, aufeinander folgenden Bits. Eine error-passive(siehe Faultconfinement 2.6.10) Station muss nach dem Senden einer Nachricht erst dieses Feld durchlaufen bevor sie den Bus wieder als ungenutzt erkennt und erneut eine Nachricht senden darf. Wird die Station innerhalb dieses Feldes ein dominantes Bit feststellen, so interpretiert sie dieses als Start-Of-Frame und wird Empfänger dieser Nachricht. Erkennt sie innerhalb dieses Feldes darf sie direkt nach Ablauf des achten Bit ein Start-Of-Frame senden und eine Nachricht übertragen, so sie denn etwas zu senden hat.

Bus-Idle-field

Das Bus-Idle-field hat keine definierte Länge. Jede Station, die einen Sendewunsch hat, darf ab dem Beginn dieses Feldes mit dem Senden beginnen. Wird innerhalb dieses Feldes ein dominantes Bit erkannt wird dies als Start-Of-Frame interpretiert. Sendebereite Stationen werden dann, mit dem nächsten Bit, das erste Bit des Identifiers ihrer Nachricht senden. Stationen ohne Sendewunsch werden sofort zum Empfänger der Nachricht.

Abbildung 2.16 zeigt den Interframespace.

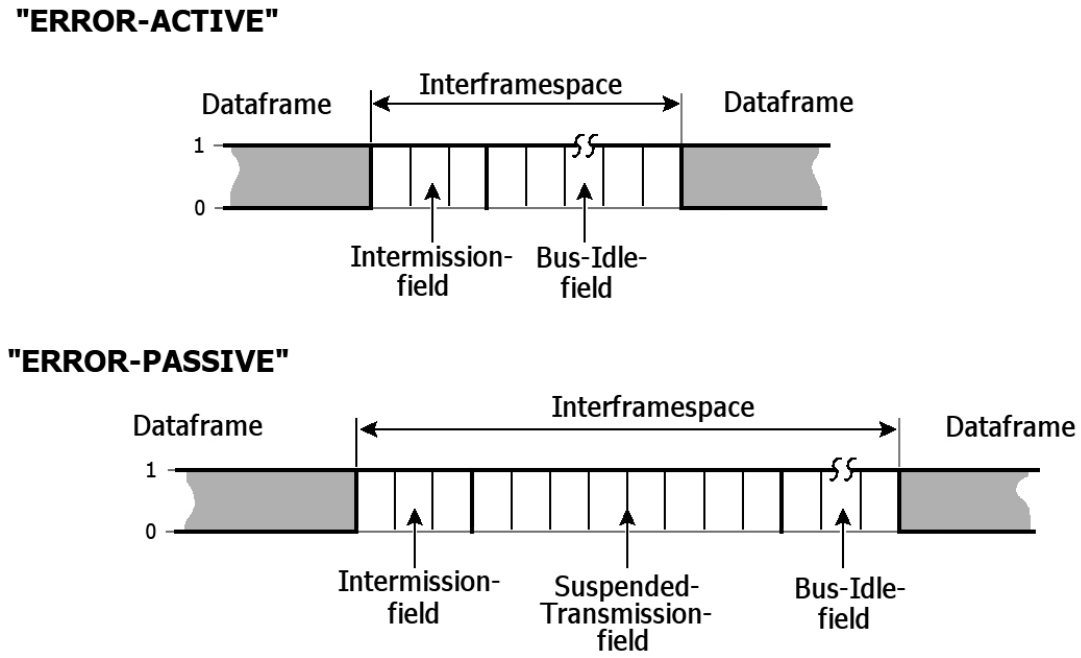


Abbildung 2.16: Interframespace

2.6.10 Faultconfinement

Das Faultconfinement verfügt über drei Zustände, die die Kommunikationsmöglichkeiten der Station festlegen:

- **Error-Active**
- **Error-Passive**
- **Bus-Off**

Der Zustand des Faultconfinements hängt dabei davon ab, wie viele und welche Fehler eine Station bekannt gegeben hat. Dies wird in zwei Zählern vermerkt. Dem Transmit-Error-Count(Anzahl der Sendefehler) und dem Receive-Error-Count(Anzahlt der Empfangsfehler).

Error-Active: Dieser Zustand stellt den Startzustand des Faultconfinements dar. Eine Station, die sich in diesem Zustand befindet darf ohne Einschränkung an der Kommunikation auf dem Bus teilnehmen. Sie darf Fehler mittels eines Active-Errorframes mitteilen und muss nach dem Senden einer Nachricht nicht noch das Suspended-Transmission-feld durchlaufen. Die Station bleibt solange Error-Active bis Transmit-Error-Count **oder** Receive-Error-Count einen Wert von mehr als 127 angenommen haben. Wenn einer der Zähler einen Wert größer als 127 angenommen hat wechselt die Station in den Zustand Error-Passive.

Error-Passive: Eine Station in diesem Zustand darf Fehler nur noch durch ein Passive-Errorframe kenntlich machen. Weiterhin muss eine Error-Passive Station, die eine fehlerhafte Nachricht gesendet hat, vor einem erneuten Senden einer Nachricht noch das Suspended-Transmission-feld durchlaufen. Die Station bleibt Error-Passive bis ihr Transmit-Error-Count **und** ihr Receive-Error-Count einen Wert kleiner oder gleich 127 angenommen haben. Sollte beide Zähler einen Wert darunter angenommen haben kehrt die Station in den Error-Active-Zustand zurück. Nimmt der Transmit-Error-Count einen Wert größer als 255 an, so geht die Station in den Bus-Off-Zustand über.

Bus-Off: In diesem Zustand darf die Station die Kommunikation auf dem Bus nicht mehr beeinflussen. Sie darf keine Nachrichten mehr senden, keine Errorframes mehr senden, kein Overloadframe senden und auch kein Acknowledge (siehe Acknowledgement 2.6.13) senden. Der Station ist es allerdings erlaubt den Bus noch abzuhören. Eine Station kann den Bus-Off nur durch zwei Möglichkeiten verlassen. Zu einen durch einen Reset des Controllers. Zum anderen darf die Station den Bus-Off wieder verlassen wenn sie 128 mal 11 direkt aufeinander folgende, rezessive Bits empfangen hat. In beiden Fällen werden Transmit-Error-Count und Receive-Error-Count auf Null zurückgesetzt und die Station geht wieder in den Error-Active-Zustand über.

Abbildung 2.17 zeigt dieses noch einmal.

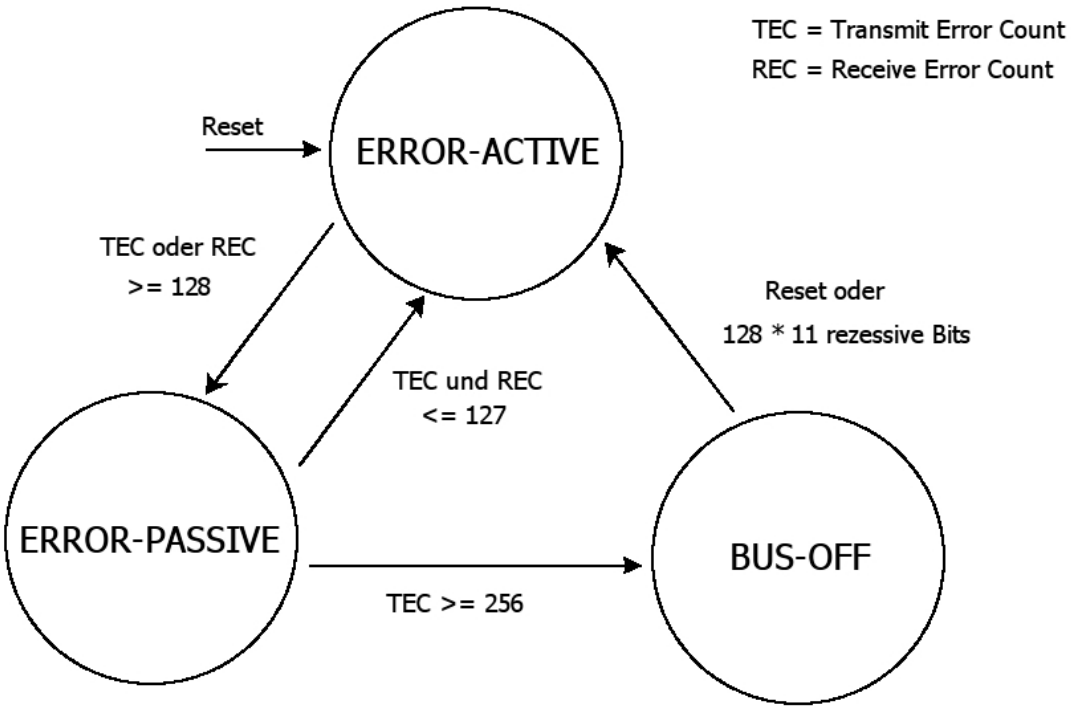


Abbildung 2.17: Faultconfinement

11 direkt aufeinander folgende rezessive Bits können auf Mehreres schließen. So ergibt die Summe von ACK-Delimiter, End-of-Frame-field und Intermission-field 11 Bit. Ebenfalls ergeben sich aus Error- oder Overload-delimiter und Intermission-field 11 Bit. Hierdurch ist also sichergestellt, dass eine fehlerhafte Station zumindest für eine Dauer von 128 Nachrichten nicht den Bus stören kann. Gängige CAN-Controller bieten dem Benutzer an, diesen Mechanismus abzuschalten. In diesem Fall kann die Station nur durch einen Reset den Bus-Off-Zustand wieder verlassen.

Wenn einer der Fehlerzähler den Wert von 96 überschreitet darf eine Warnung ausgegeben werden, da dieser Wert auf einen stark gestörten Bus hinweist. Dies ist in der Spezifikation nicht explizit vorgeschrieben gängige CAN-Controller bieten dies allerdings an.

2.6.11 Arbitrierung

Die Arbitrierung beginnt mit dem Start-Of-Frame. Jede sendebereite Station wird nach diesem Bit anfangen ihre Nachricht zu senden.

Der Arbitrationsalgorithmus verläuft dabei wie folgt:

1. lege nächstes Bit auf den Bus

2. überprüfe, ob Bit auf dem Bus angekommen ist. Wenn ja: weiter mit Punkt 1. Wenn nein: weiter mit Punkt 3.
3. stelle Senden ein und wechsle in Empfängermodus

Abbildung 2.18 zeigt den Arbitrierungsvorgang am Beispiel von 3 sendenden Stationen.

Die Arbitrationsphase endet mit dem Ende des Arbitrationfields. Wird also vom Sender ein abweichendes Bit nach diesem Feld festgestellt so wertet dieser dies als Fehler.

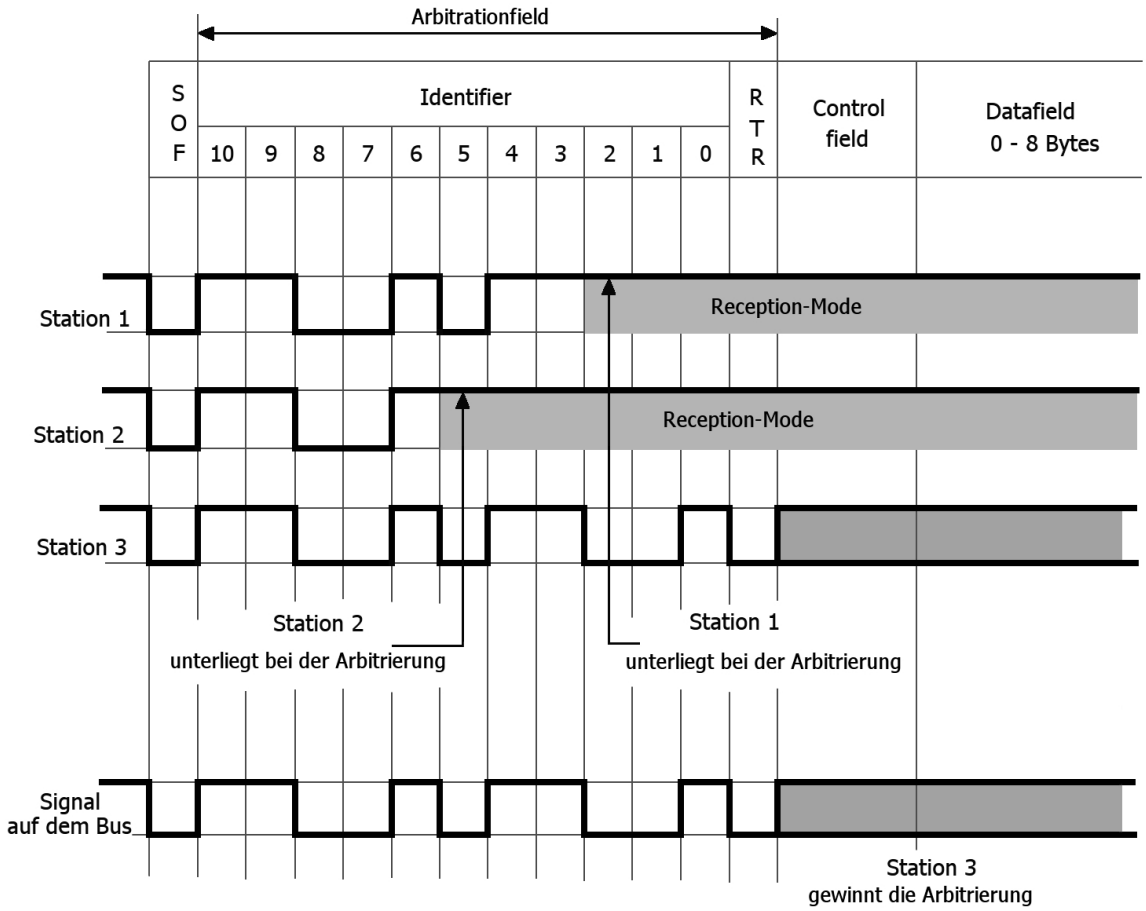


Abbildung 2.18: Arbitrierung:
Alle Station beginnen gemeinsam mit den Start-Of-Frame-bit(SOF) und schalten nacheinander die Bits ihres Identifiers auf. In Bit 5 setzen sich die Bits der Stationen 1 und 3 gegen das rezessive Bit von Station 2 durch. Station 2 wechselt darauf hin in den Empfänger-Modus(Reception-Mode). In Bit 2 unterliegt Station 1 Station 3 und wechselt auch in den Empfänger-Modus. Mit dem Beginn des Controlfields gewinnt Station 3 die Arbitrierung und übernimmt den Bus.

2.6.12 Definition von Sender und Empfänger

Eine Station, die eine Nachricht, also Data- oder Remoteframe, sendet ist ein **Sender**. Die Station bleibt so lange Sender bis der Bus wieder idle ist oder die Station die Arbitrierung verloren hat.

Als **Empfänger** gelten alle Stationen, die **nicht** Sender sind. Sobald der Bus wieder idle ist werden alle Stationen Empfänger.

Nach dieser Definition ergibt sich, dass eine Station, obwohl diese Empfänger ist, dennoch senden kann. Dies wäre beim senden eines Errorframes der Fall.

2.6.13 Acknowledgement

Durch das Acknowledgement wird dem Sender der Nachricht mitgeteilt, ob die Nachricht fehlerfrei empfangen wurde. Genauer gibt dies bei CAN an, ob die empfangene CRC-Summe mit der errechneten CRC-Summe der Empfänger übereinstimmt.

Der Sender des Frames muss im ACK-Slot immer sein rezessives Bit senden. Die Empfänger, die die CRC-Summe korrekt empfangen haben, werden im ACK-Slot ein dominantes Bit senden. Die ist das sogn. ACK. Abbildung 2.19 zeigt dies.

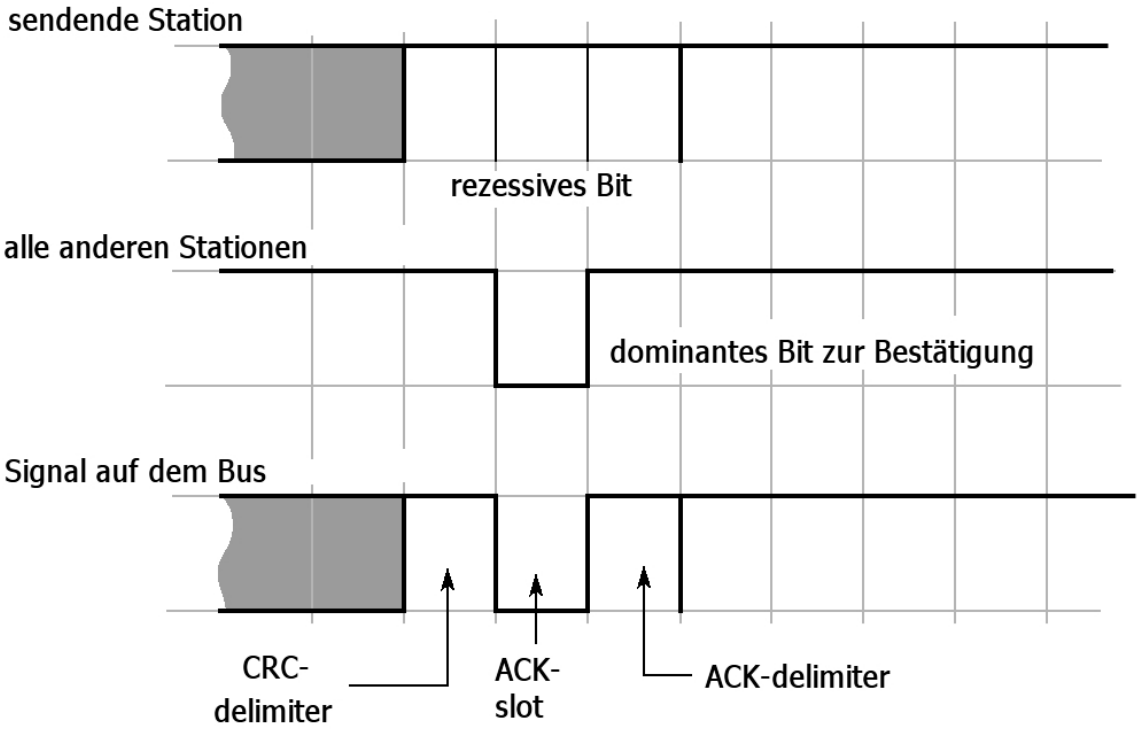


Abbildung 2.19: Positive Acknowledgement - ACK

Stationen, die keine Übereinstimmung von errechneter und empfangener CRC-Summer feststellen zeigen dies an indem sie ein rezessives Bit im ACK-Slot senden. Dies ist das sogn. Negative-Acknowledgement auch NAK genannt. Dies ist in Abbildung 2.20 dargestellt. Ein Sender wird wenn er kein ACK im ACK-Slot empfangen hat nach dem ACK-Slot beginnen ein Errorframe zu senden.

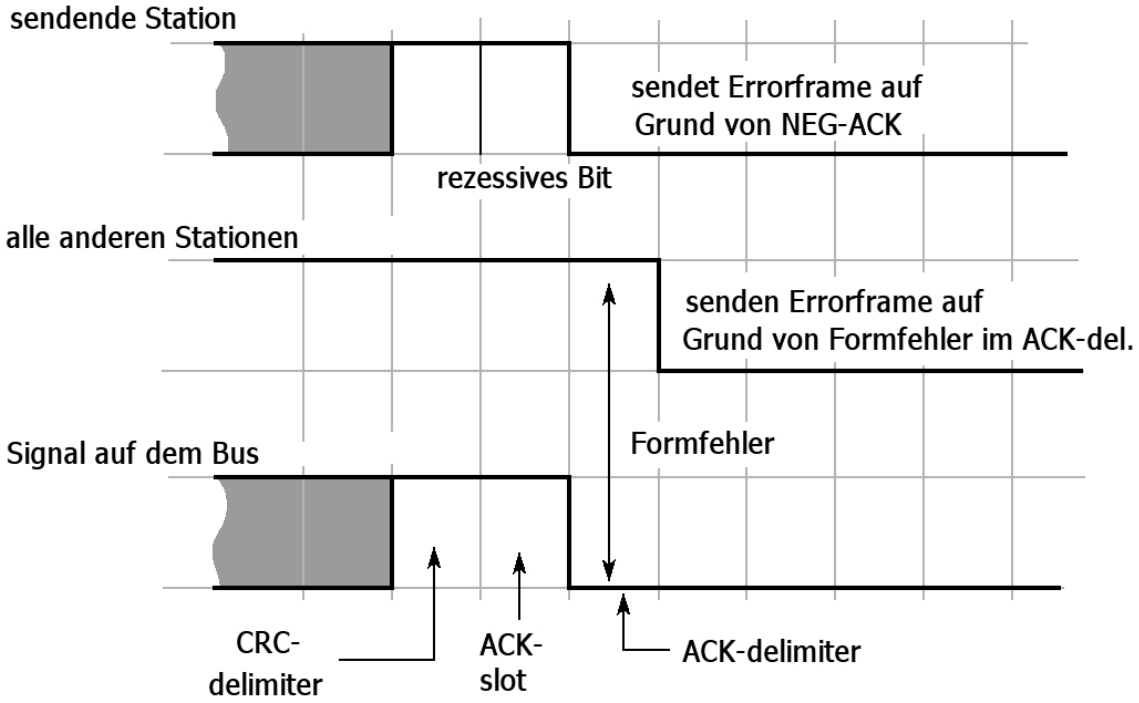


Abbildung 2.20: Negative Acknowledgement - NAK

Durch das Überschreiben von rezessiven durch dominate Bits kann es beim Negative-Acknowledgement zu einer Art Multicast kommen. Wenn eine Station lokal einen CRC-Fehler erkennt und deshalb ein rezessives Bit im ACK-Slot sendet, wird diese durch Stationen, die dieses Fehler nicht gesehen haben, überstimmt. Die Station wird daher nach dem ACK-Delimiter anfangen ein Errorframe zu senden. Dies ist in Abbildung 2.21 dargestellt.

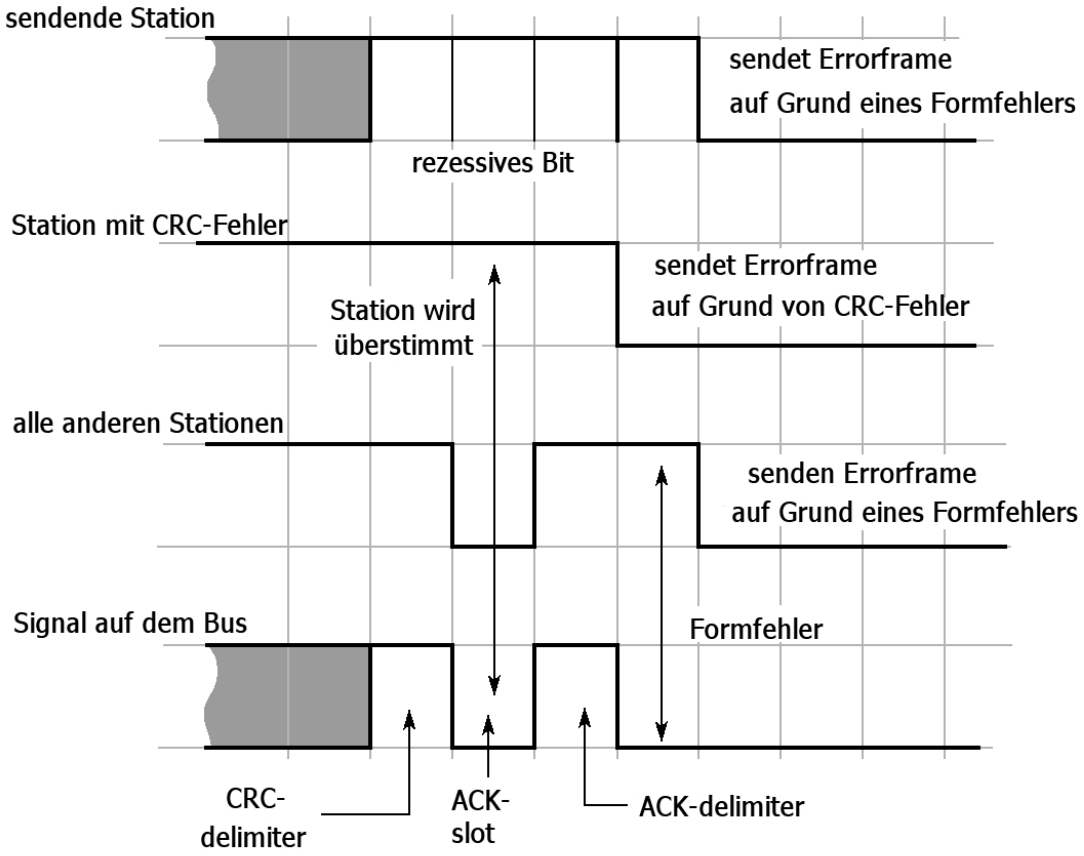


Abbildung 2.21: Überstimmung eines Negative-Acknowledgement

2.6.14 Gültigkeit einer Nachricht

Der Zeitpunkt an dem eine Nachricht als gültig empfangen und als gültig versandt gilt ist für Sender und Empfänger unterschiedlich. Für den Sender der Nachricht ist diese gültig wenn kein Fehler bis zum Ende des End-of-Frame-fields, also am Ende des siebten Bits des End-of-Frame-fields, aufgetreten ist. Für Empfänger gilt eine Nachricht als gültig wenn bis zum Ende des vorletzten Bits des End-of-Frame-field, also das Ende des sechsten Bits, kein Fehler aufgetreten ist. Abbildung 2.22 fasst dies noch einmal zusammen. RX-OK steht dabei für Reception-OK. TX-OK für Transmission-OK.

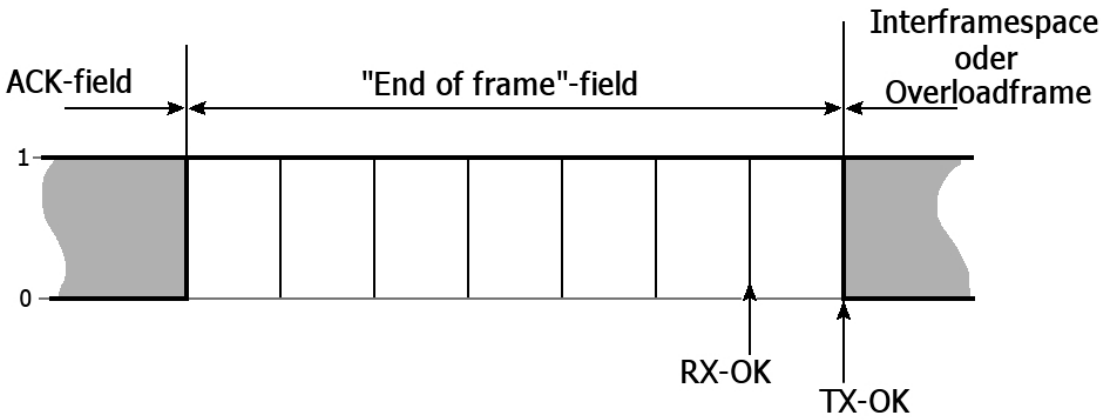


Abbildung 2.22: Gültigkeit einer Nachricht

2.6.15 Fehlererkennung

CAN nutzt mehrere Mechanismen, um Fehler während der Übertragung zu erkennen. Diese sollen in diesem Abschnitt erklärt werden.

Bitmonitoring

Wann immer eine Station ein Bit sendet überprüft diese, ob das Bit in dieser Form auf dem Bus angekommen ist. Eine Abweichung vom gesendeten Bit wird als **Bit-Error** gewertet. Hierbei gilt:

Anmerkung: 0 ist dominant, 1 ist rezessiv

empfangenes Bit	gesendetes Bit	Resultat
dominant	dominant	kein Fehler
dominant	rezessiv	möglicher Bit-Error
rezessiv	dominant	Bit-Error
rezessiv	rezessiv	kein Fehler

Für den Fall, dass ein rezessives Bit gesendet und ein dominantes Bit empfangen wurde gilt für Sender:

- Innerhalb des Arbitrationfields ist dies kein Fehler. Die Station muss hierauf Empfänger werden.
- Außerhalb des Arbitrationfields ist dies ein Bit-Error.
- Im ACK-Slot ist dies kein Bit-Error.

Wenn ein dominantes Bit gesendet wurden und ein rezessives Bit empfangen wurde so ist dies immer ein Bit-Error.

Bitstuffing

Bitstuffing(Bitstopfen) ist ein Mechanismus, bei dem nach einer bestimmten Anzahl von Bits gleicher Polarität ein Bit mit umgekehrter Polarität eingefügt wird. Die Anzahl Bits gleicher Polarität, die sogn. Stuff-Breite, beträgt bei CAN fünf Bit. Ein Sender wird die Nachricht während der Übertragung mit diesen zusätzlichen Bits versehen. Dies wird als **Stuffing** bezeichnet. Alle Empfänger werden diese Bits wieder aus dem Datenstrom entfernen, z.B. indem diese ihre zuständigen Automaten für die Dauer des Stuffbits anhalten. Dies wird als **Destuffing** bezeichnet. Abbildung 2.23 zeigt dieses.

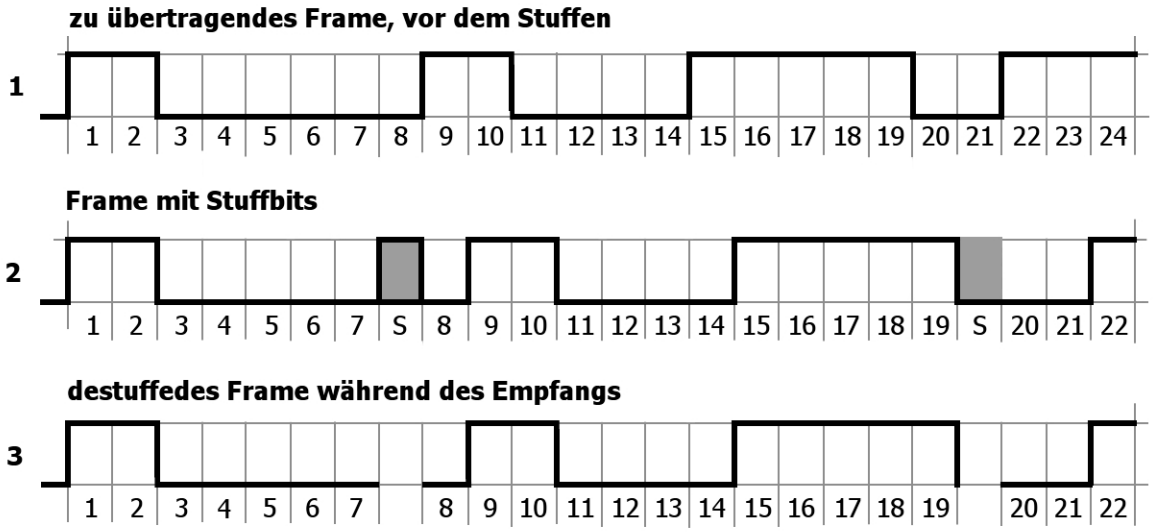


Abbildung 2.23: Bitstuffing

Durch diesen Mechanismus ist eine kontinuierliche Überprüfung der Übertragung sowohl von Sender- als auch von Empfängerseite möglich. Alle Empfänger erwarten nach einer Abfolge von fünf gleichwertigen Bits ein komplementäres Bit. Das Fehlen dieses Bits werten die Empfänger als **Stuff-Error**. Der Sender der Nachricht wird dies nicht als Stuff-Error interpretieren, da dieser dies durch das Bitmonitoring als **Bit-Error** erkennen wird. Das Bitstuffing dient außerdem dazu, zu verhindern, dass über eine zu lange Zeit kein Flankenwechsel auf dem Bus stattfindet. Das Bitstuffing wird nur beim Dataframe und beim Remoteframe verwendet. Das Bitstuffing wird mit Start-Of-Frame-bit gestartet und endet mit dem letzten Bit des CRC-fields. Der CRC-Delimiter ist bereits nicht mehr im Bereich des Bitstuffing enthalten. Abbildung 2.24 zeigt den Bereich in dem Bitstuffing eingesetzt wird.

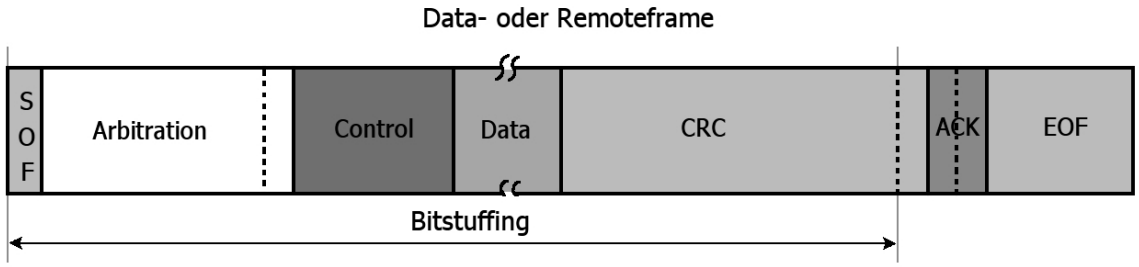


Abbildung 2.24: Einsatzbereich des Bitstuffings

Cyclic-Redundancy-Check

Cyclic Redundancy Check wird nur auf den Datenstrom ohne Stuffbits angewendet. CRC beruht auf Polynomdivision. Der Datenstrom wird dabei als Polynom betrachtet und durch ein Generatorpolynom geteilt. Das dabei entstandene Restpolynom bildet die CRC-Checksumme. Zur Berechnung des Restpolynoms wird folgendes Generatorpolynom verwendet:

$$Generatorpolynom := x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1 \tag{2.1}$$

CRC wird nur bei Data- und Remoteframe angewandt. Der Bereich zur Bildung der Checksumme ist in Abbildung 2.25 dargestellt.

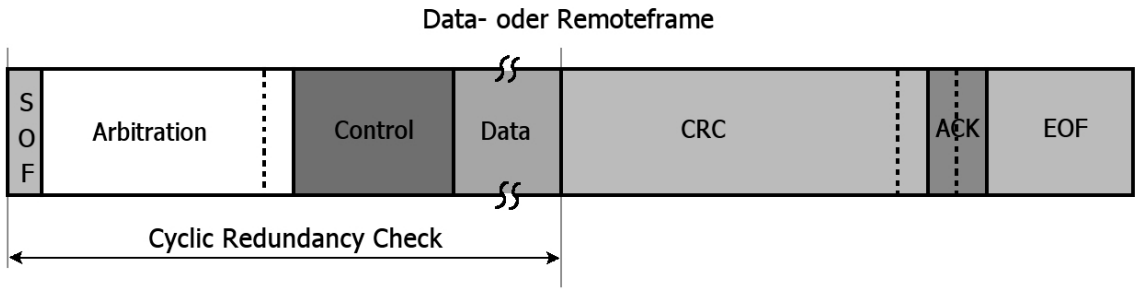


Abbildung 2.25: Einsatzbereich des Cyclic-Redundancy-Checks

Jede Station berechnet während der Übertragung ihre eigene CRC-Checksumme. Nur die Sendestation übermittelt diese allerdings. Alle Empfänger überprüfen **nach dem Empfang**

diese Summe, ob diese mit der errechneten Checksumme übereinstimmt. Es wäre auch denkbar die Überprüfung bereits während des Empfangs der Checksumme durchzuführen und so sofort einen Fehler zu melden. Dies wird bei CAN aber nicht getan. Ein Abweichen der empfangenen von der errechnete Checksumme wird als **CRC-Error** gewertet.

In der Spezifikation ist ein Pseudocode zur Implementierung dieses Verfahrens angegeben.

Frameformatcheck

Beim Frameformatcheck wird Überprüft, ob die Bits, für die eine feste Form im Frame definiert wurde, diese auch einhalten. In Data- sowie im Remoteframe sind dies die Bits:

- CRC-Delimiter
- ACK-Delimiter
- alle Bits des End-of-Frame-fields

Alle diese Bits müssen rezessiv sein. Ein Abweichen stellt einen **Form-Error** dar. Eine Ausnahme stellt in diesem Fall das letzte Bit des End-of-Frame-fields dar. Ein dominantes Bit an dieser Stelle wird von den Empfängerstationen nicht als Fehler gewertet, sondern als Beginn eines Overloadframes.

2.6.16 Fehlersignalisierung

In diesem Abschnitt soll erklärt werden, welchen Besonderheiten es bei der Fehlersignalisierung gibt.

Globale und lokale Fehler

Eine Station in einem CAN-Netz wird einen erkannten Fehler so schnell wie möglich an alle Stationen weitergeben. Fehler müssen dabei aber nicht von allen Stationen auch wahrgenommen werden. In diesem Zusammenhang müssen globale Fehler, also Fehler, die von allen Stationen gleichzeitig erkannt werden und lokale Fehler, also Fehler, die nur von einer Station gesehen werden unterschieden werden.

Lokale Fehler können viele Ursachen haben. Ein falsch eingestelltes Bittiming, ein defekter Bustreiber oder auch Umwelteinflüsse spielen dabei eine Rolle. Um zu Verhindern, dass eine Station, die stetig lokale Fehler meldet, den Bus permanent stört, wird bei CAN ein

Mechanismus angewandt der eine Unterscheidung zwischen lokalen und globalen Fehlern ermöglicht.

Dieser Mechanismus erfolgt durch eine Überlappung von Errorflags. Wenn eine Station ein Errorflag auf Grund eines lokalen Fehlers sendet, so werden die restlichen Stationen ebenfalls ein Errorflag senden, sobald diese eine Abweichung erkennen. Dies ist bestenfalls ein Bit später der Fall; schlechtesten Falls, durch Verletzung des Bitstuffings, 6 Bit später. Die Station, die als erste ein Errorflag gesendet hat sendet somit auch als erste eine Errordelimiter. Da der Errordelimiter rezessiv gesendet wird, können die anderen Stationen ungestört ihre Errorflags weiter senden. Bemerkt nun eine Station im ersten Bit des Errordelimiters ein dominantes Bit, so steht fest, dass diese einen lokalen Fehler gemeldet hat. Abbildung 2.26 zeigt diesen Vorgang.

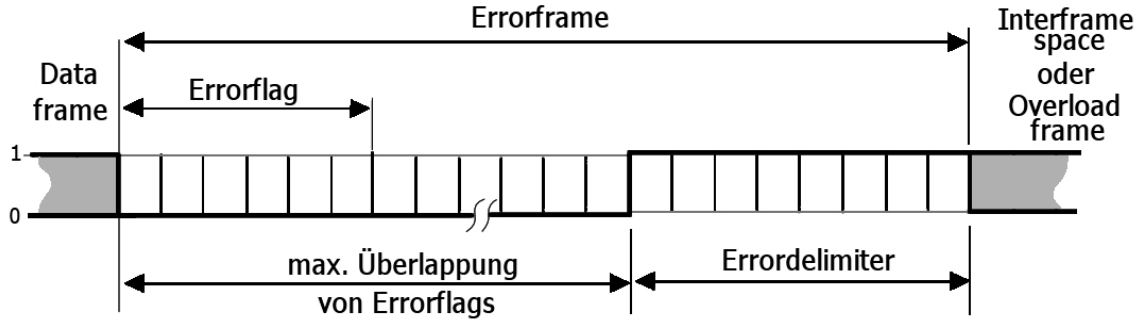


Abbildung 2.26: Überlappung von Errorflags

Auf Grund dieser Überlappung tolerieren alle Stationen eine Folge von bis zu 7 dominanten Bits zu Beginn des Errordelimiters.

Ein globaler Fehler hingegen wird von alle Stationen gleichzeitig bemerkt, woraufhin diese auch gleichzeitig das Errorflag senden um anschließend gleichzeitig mit dem Errordelimiter zu beginnen. In diesem Fall kann keine Station ein dominantes Bit im Errordelimiter vorfinden.

CRC-Fehler

Die Fehlersignalisierung nach dem Erkennen eines CRC-Fehlers weicht von Signalisierung anderer Fehler ab. Erkennt eine Station einen CRC-Fehler so macht sie dies durch ein negatives ACK kenntlich und wird nach dem ACK-Delimiter anfangen einer Errorflags zu senden, so denn der Sender nicht schon, auf Grund des Negative-ACK, einen Fehler meldet. Da dieser CRC-Fehler aber auch ein lokal erkannter Fehler sein kann ist es möglich, das andere Stationen ein positives ACK senden und der Sender den ACK-Delimiter nicht zerstört. In

diesem Fall sendet die Station ihr Errorflag nach dem ACK-Delimiter. Abbildung 2.27 zeigt dies.

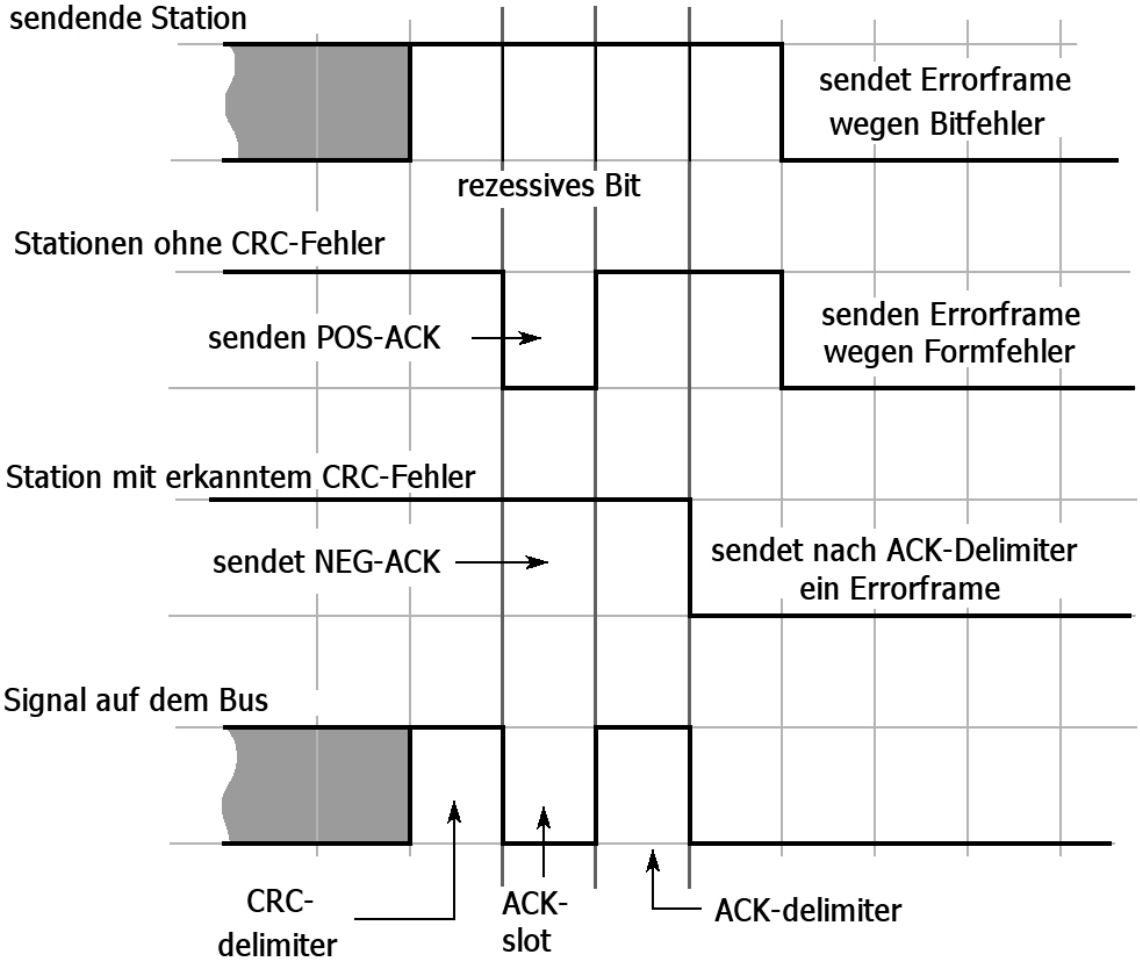


Abbildung 2.27: lokaler CRC-Fehler

2.6.17 Fehlerregeln für das Faultconfinement

Für das Inkrementieren und das Dekrementieren der Fehlerzähler, Transmit-Error-Count(TEC) und Receive-Error-Count(REC) sieht die Protokollspezifikation folgende Regeln vor:

Für **Sender** gilt:

- Wann immer ein Errorflag gesendet wird: erhöhe TEC um 8. **Ausnahme**: Wenn die Station Error-Passive ist und ein Errorframe auf Grund eines Negative-ACK sendet und in ihrem Passive-Errorflag kein dominantes Bit entdeckt: verändere den TEC nicht.

- Bit-Error während des Sendens von Active-Errorflag oder Overloadflag: erhöhe TEC um 8.
- Nach jeder Folge von 8 dominanten Bits im Error- oder Overload-Delimiter: erhöhe TEC um 8.
- Nach erfolgreichem Versenden einer Nachricht: verringere TEC um 1.

Für **Empfänger** gilt:

- Wann immer ein Errorflag gesendet wird: erhöhe REC um 8. **Ausnahme**: Bit-Error während ein Active-Error- oder Overload-flag gesendet wird.
- Erkennen eines dominanten Bits als erstes Bit im Errordelimiter: erhöhe REC um 8.
- Bit-Error beim senden von Active-Error- oder Overload-flag: erhöhe REC um 8.
- Nach jeder Folge von 8 dominanten Bits im Error- oder Overload-Delimiter: erhöhe REC um 8.
- Nach erfolgreichem Empfang einer Nachricht: verringere REC um 1.

2.6.18 Acceptance-Filtering

Die Feststellung, ob die Nachricht vom Controller an die CPU des Systems weitergegeben, also akzeptiert, werden soll, erfolgt auf dem gesamten Identifier der Nachricht. Die Spezifikation erlaubt hierfür den Einsatz von Mask-Registern. In diesen Registern können Teile des Identifiers maskiert, also als von Interesse bzw. nicht von Interesse markieren, werden. Wenn solche Mask-Register verwendet werden, so schreibt die Spezifikation vor, dass dann jedes Bit dieser Register programmierbar sein muss.

2.6.19 Bittiming

In diesem Abschnitt soll das Bittiming erklärt werden, sowie die Regeln die für die Synchronisierung spezifiziert sind.

Aufbau eines Bits

Bei CAN ist die Dauer eines Bit, die sogn. nominale Bitzeit, in vier Segmente unterteilt. Diese Segmente bestehen aus einem ganzzahligen Vielfachen von sogn. Zeitquanten(Timequanta - TQ). Ein TQ stellt die kleinste Zeiteinheit im Bittiming dar. Die Dauer eines TQ ist vom Systemtakt abgeleitet und kann einen oder mehrere Takte umfassen. Um diese Dauer festzulegen kann ein Vorteiler(Prescaler) verwendet werden. Dies ist in der Spezifikation aber

nicht vorgeschrieben. Die Dauer des TQ kann sich während des Betriebs, von Oszillatorschwankungen einmal abgesehen, nicht ändern.

Die Bitsegmente erfüllen unterschiedliche Aufgaben und haben verschiedene Längen:

- **Synchronization Segment - SYNC_SEG**: Ist immer immer ein TQ lang und wird für die Synchronisierung verwendet. Diese Segment kann nicht in der Dauer eingestellt werden.
- **Propagation Delay Segment - PROP_SEG**: Ist in der Länge einstellbar. Es wird verwendet um die Signallaufzeit auf dem Bus zu kompensieren.
- **Phase Buffer Segment 1 - PHASE_SEG1**: Ist in der Länge einstellbar und wird zum Ausgleich von Phasenfehlern genutzt.
- **Phase Buffer Segment 2 - PHASE_SEG2**: Ist in der Länge einstellbar und wird zum Ausgleich von Phasenfehlern genutzt.

Der Zeitpunkt an dem der Buspegel übernommen wird, wird als Samplepoint bezeichnet und befindet sich zwischen PHASE_SEG1 und PHASE_SEG2. Der Samplepoint kann durch einstellen der Werte für PHASE_SEG1 und PHASE_SEG2 verschoben werden.

Die Werte für PROP_SEG, PHASE_SEG1 und PHASE_SEG2 müssen vom Anwender eingestellt werden. Ebenso muss der Wert für den Prescaler eingestellt werden, wenn ein solcher verwendet wird. Diese Werte sind für Controller verschiedener Hersteller unterschiedlich und werden meist in den Datenblättern zu diesen angegeben.

Abbildung 2.28 zeigt den Aufbau eines Bits. In diesem Beispiel beträgt die Dauer eines TQ drei Systemtakte(F_{Sys})

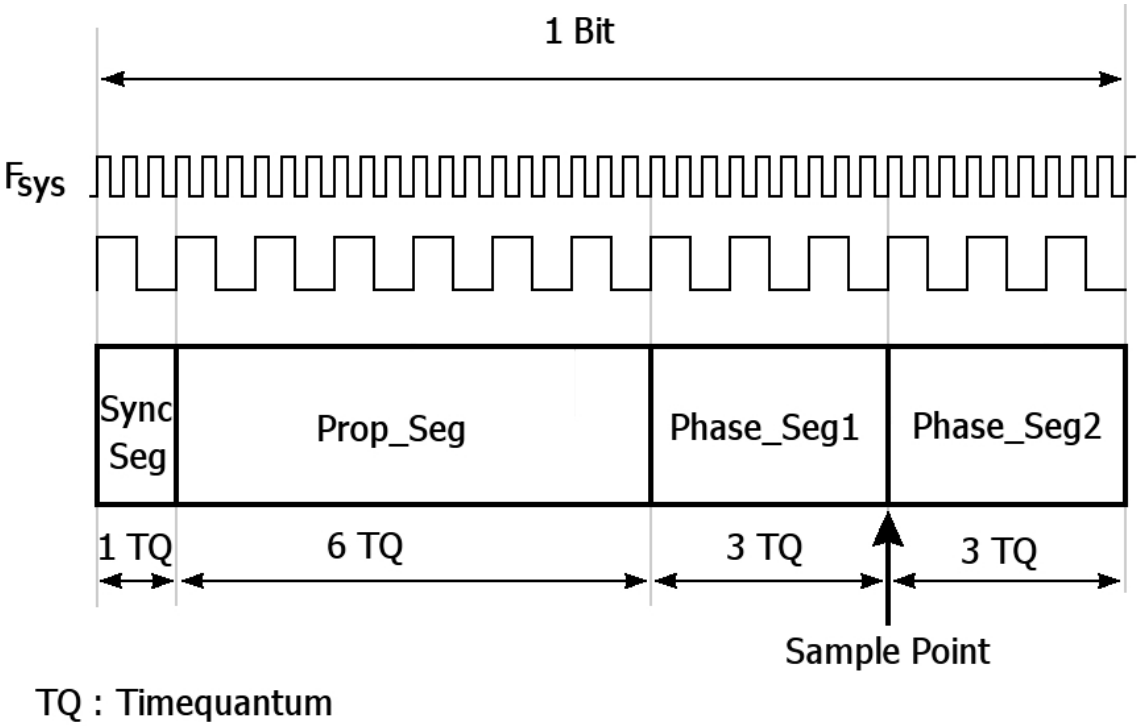


Abbildung 2.28: Unterteilung eines Bits

Einen weiteren wichtigen Wert stellt die **Information Processing Time**, also die Zeitdauer, die nach Samplen des Buspegels benötigt wird um diesen Wert zu verarbeiten und ggf. ein Nachfolgebit festzulegen, dar. Diese Zeit darf maximal 2 TQ betragen.

Die Spezifikation schreibt weiterhin vor, dass die Gesamtdauer eines Bits mindestens auf einen Wert zwischen 8 und 25 TQ einstellbar sein muss. Die Spezifikation verbietet nicht, dass diese Dauer mehr als 25 TQ umfassen darf. Für einen ggf. verwendeten Prescaler muss der Wert mindestens zwischen 1 und 32 einstellbar sein. Auch hier sind größere Werte nicht verboten.

Weiterhin schreibt die Spezifikation für die Längen der einzelnen Segment vor:

- **SYNC_SEG**: 1 TQ lang.
- **PROP_SEG**: zwischen 1 bis 8 TQ einstellbar.
- **PHASE_SEG1**: zwischen 1 bis 8 TQ einstellbar.
- **PHASE_SEG2**: Maximum aus PHASE SEG1 und Information Processing Time.
- **Information Processing Time**: 0 bis 2 TQ.

Synchronisierung

Für die Synchronisierung werden bei CAN, seid des Spezifikation 2.0, nur die Flanken verwendet, die aus einem Wechsel von rezessivem zu dominantem Buspegel herrühren. Dabei muss eine solche Flanke innerhalb der SYNC_SEG liegen damit die Bittiming synchron ist. Andernfalls liegt ein Phasenfehler vor. Dieser Phasenfehler kann entweder positiv oder negativ sein. Dies hängt vom Samplepoint ab. Tritt die Flanke nach dem SYNC_SEG und vor dem Samplepoint auf so ist der Phasenfehler positiv. Tritt die Flanke zwischen Samplepoint und SYNC SEG auf so ist der Phasenfehler negativ.

CAN verwendet zwei Arten der Synchronisierung. Zum einen die sogn. Harte Synchronisierung und zum anderen die sogn. Nachsynchronisierung.

Bei der Harten Synchronisierung wird das Bittiming beim Auftreten einer Flanke mit dem SYNC_SEG neu gestartet. Abbildung 2.29 zeigt eine solche harte Synchronisierung.

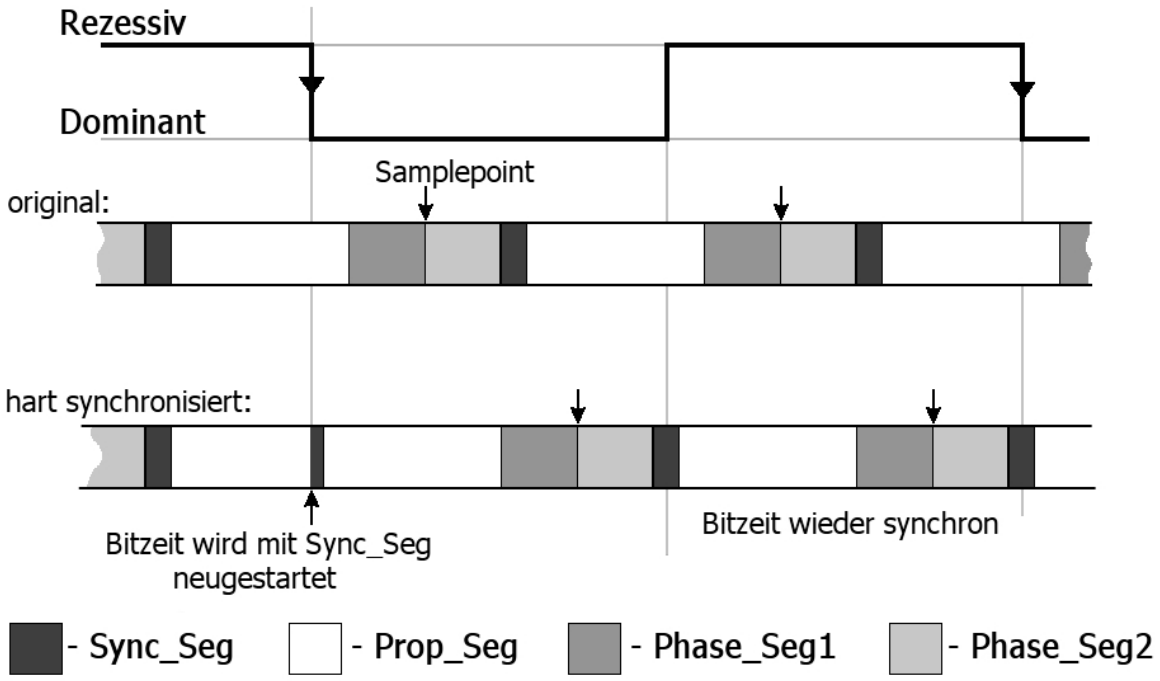


Abbildung 2.29: Harte Synchronisierung

Bei der Nachsynchronisierung wird bei positivem Phasenfehler das PHASE_ SEG1 verlängert; bei negativem Phasenfehler wird das PHASE_SEG2 verkürzt. Die Anzahl an TQ um die PHASE_SEG1 und PHASE_SEG2 verlängert bzw. verkürzt werden dürfen hängt dabei von der sogn. **Resynchronization Jump Width - RJW**, in der Literatur wird diese auch als

Synchronization Jump Width - SJW bezeichnet, ab. Diese sollte laut Spezifikation zwischen 1 und dem Minimum von PHASE_SEG1 und 4 einstellbar sein.

Abbildung 2.30 zeigt die Nachsynchronisierung bei einer sogn. frühen Flanke. Ein frühe Flanke liegt dann vor, wenn diese vor dem SYNC_SEG liegt. In diesem Fall ist der Phasenfehler negativ und das PHASE_SEG2 wird verkürzt. Das PHASE_SEG2 wird dabei um den Betrag des Phasenfehlers verkürzt. Ist dieser größer als die RJW so wird dieses Segment um den Betrag von RJW verkürzt. Ist der Betrag des Phasefehlers kleiner oder gleich RJW so wird der Rest dieses Segments sowie das nachfolgende SYNC_SEG ausgelassen.

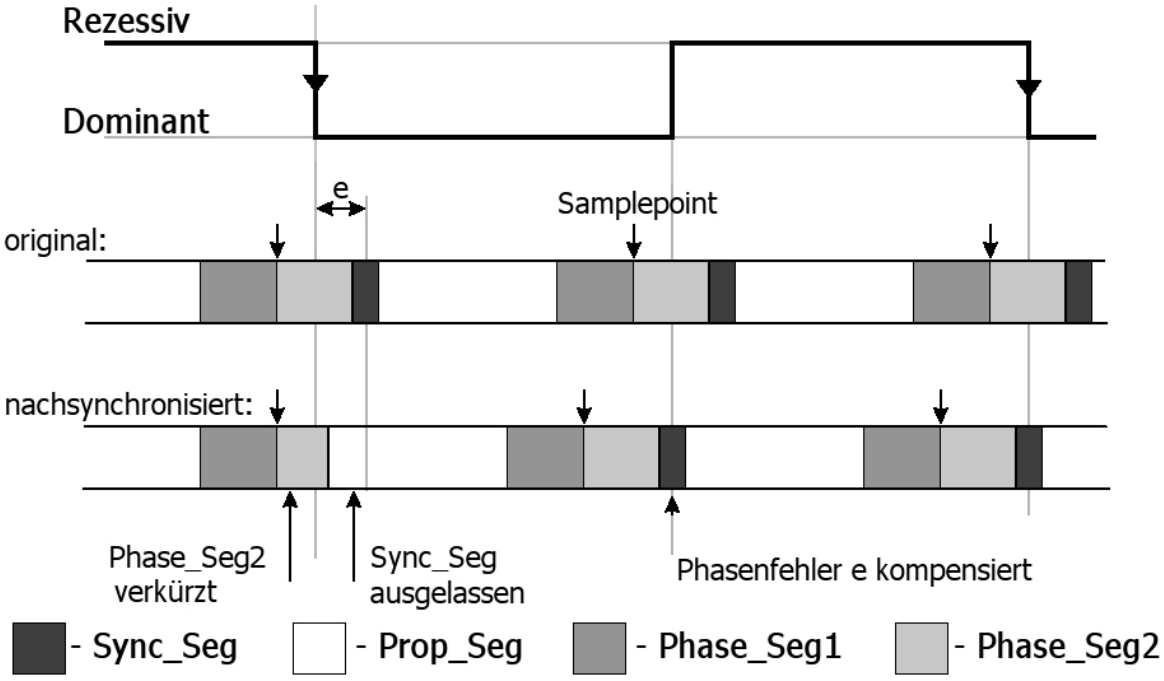


Abbildung 2.30: Nachsynchronisierung - frühe Flanke

Im Falle eines positiven Phasenfehlers spricht man von einer späten Flanke. In diesem Fall wird das PHASE_SEG1 verlängert. Abbildung 2.31 zeigt dies. Ist der Phasenfehler kleiner als RJW so wird das PHASE_SEG1 um diesen Wert verlängert. Ist der Phasenfehler größer als RJW so wird PHASE_SEG1 um RJW verlängert.

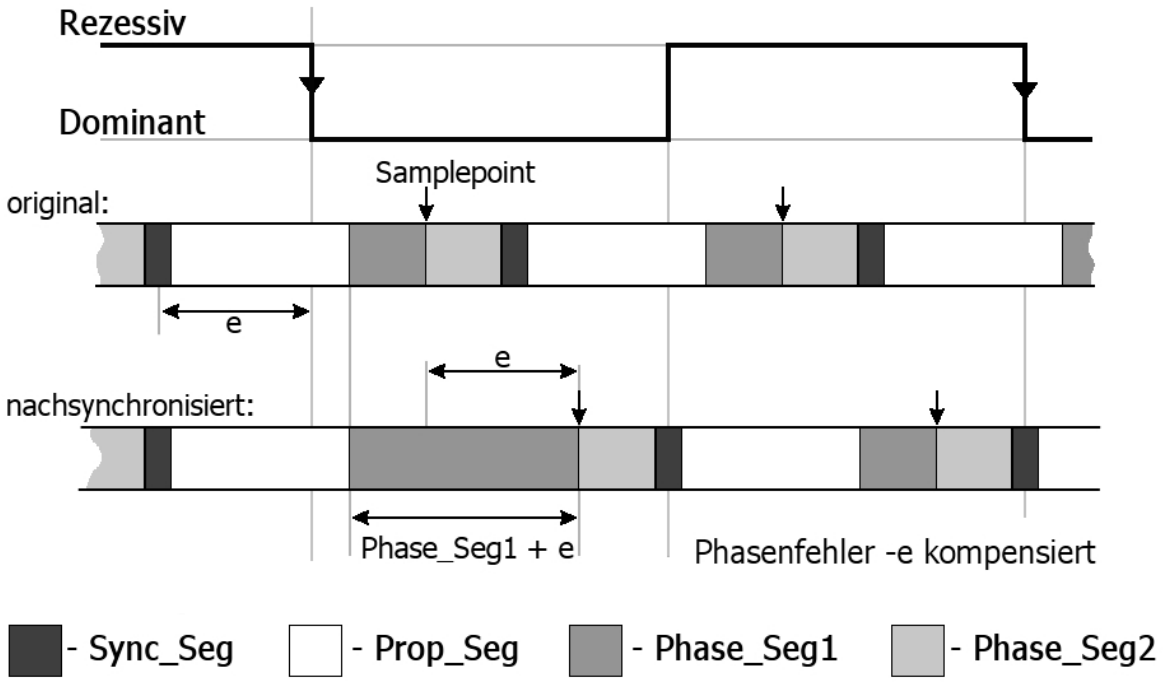


Abbildung 2.31: Nachsynchronisierung - späte Flanke

Für harte Synchronisierung und Nachsynchronisierung gelten folgende Regeln:

- Es ist nur eine Synchronisierung während einer Bitzeit erlaubt.
- Eine Flanke wird nur dann zur Synchronisierung verwendet, wenn der Buspegel direkt nach dieser Flanke von dem abweicht, der während des letzten Samplepoint gelesen wurde.
- Wenn ein Flankenwechsel von rezessiv zu dominant stattfindet während der Bus IDLE ist, so wird ein harte Synchronisierung vorgenommen.
- Für alle anderen Flanken von rezessiv zu dominant wird Nachsynchronisierung verwendet. Mit der Ausnahme, dass eine Station, die selber ein dominantes Bit sendet sich nicht selber darauf nachsynchronisieren darf.

2.7 Weitere Funktionen eines CAN-Controllers

In diesem Abschnitt wird analysiert welche weiteren Funktionen gängige CAN-Controller umfassen.

Es gibt zwei Begriffe, die im Zusammenhang mit CAN des öfteren auftauchen. Diese sind Full-CAN und Basic-CAN. Diese Begriffe stammen aus der Zeit der ersten CAN-Controller

und sind nicht spezifiziert. Full-CAN und Basic-CAN stehen für unterschiedliche Konzepte für die Annahme und den Versand von Nachrichten, beschreiben jedoch nicht die Mächtigkeit des Controllers. Auch wenn der Ausdruck Full-CAN den Eindruck erwecken sollte, dass dies mehr vom eigentlichen CAN-Protokoll implementiert. Mit den eigentlichen Protokoll-Spezifikationen haben diese Begrifflichkeiten allerdings nichts zu tun und sollten auch nicht mehr verwendet werden.

Als **Basic-CAN** werden bzw. wurden Controller bezeichnet bei den relativ wenige Hardwareaufwand für den Versand und die Annahme von Nachrichten betrieben wurde. Diese Controller hatten meist nur einen Transmitbuffer, sowie einen Receivebuffer und überliessen der CPU das Acceptance-Filtering und das Antworten auf Remoteframes.

Full-CAN Controller dagegen verfügen meist über mehrere Hardwarekomponenten, die jeweils über Transmit- und Receivepuffer verfügen und auch mit einem eigenen Acceptance-Filtering versehen sind. Zudem verfügen diese meist über einen Automatismus für die Antwort auf Remoteframes, sogn. Auto-Reply. Ein Full-CAN-Controller verursacht so weniger Last auf der CPU.

Viele aktuelle CAN-Implementierungen bieten außerdem eine Unterstützung für Time-Triggered-CAN(TTCAN), nach ISO 11898-4 an. Da die ISO-Standards hier jedoch nicht zur Verfügung stehen und genaue Informationen zu einer solchen Hardwareunterstützung fehlen wird TTCAN hier nicht mehr weiter behandelt.

2.8 CAN Busankopplung

Die Busankopplung zu CAN ist in der Spezifikation ausgelassen. In der Praxis sind Tranceiver-Bausteine nach dem ISO-Standard, also den Standards 11898-2, 11898-3 und 11898-5, am geläufigsten. Firmen wie z.B. Texas-Instruments, Atmel, NXP, Intel bieten diese an.

3 Design

Nachdem im vorigen Kapitel analysiert wurde welche Vorgaben und welche Freiheiten für die Entwicklung eines CAN-Controllers existieren soll nun in diesem Kapitel der Entwurf dieses Controllers dargelegt werden.

3.1 Designentscheidungen

In diesem Abschnitt sollen die Designentscheidungen erörtert werden.

3.1.1 Wahl des Konzepts für Versand und Empfang von Nachrichten

Als erstes soll die Entscheidung getroffen werden, welche Möglichkeiten zum Versand und zum Empfang der Controller bereitstellen soll. Hierbei muss bedacht werden, dass der hier entwickelte Controller lediglich ein kleines Teilstück für diese SoC werden soll. Je mehr Hardware dieser Controller also benötigt, desto weniger Hardware bleibt für andere Teile des SoC. Weiterhin muss bedacht werden, dass die CPU, welche später im SoC verwendet werden soll, eine ARM9 CPU ist. ARM-CPUs zeichnen sich unter anderem dadurch aus, dass dieses relativ schnell sind. Eine genaue Taktfrequenz lässt sich zum jetzigen Zeitpunkt zwar noch nicht abschätzen, da diese noch nicht fertiggestellt ist und auch noch nicht feststeht in was für einem FPGA dieses SoC später implementiert werden soll. Daher ist die hier getroffene Abschätzung dazu rein spekulativ. Ausgegangen wird hier von einer Taktfrequenz von ca. 50 MHz.

Kritisch betrachtet werden sollen diese beiden Konzepte in Hinblick auf zwei Situationen, die in einem CAN-Bussystem auftreten können. Um die folgende Rechnung einfacher zu machen wird hier von einer Bitrate von 1 MBit ausgegangen, was einer Zeitspannen von 1 µs pro Bit entspricht. Die erste Situation entsteht durch die Anforderung eines Dataframes durch ein Remoteframe. Zwischen dem Zeitpunkt, an dem das Remoteframe als gültig bewertet wird(RX_OK), und dem Zeitpunkt, an dem das entsprechende Dataframe gesendet werden kann, liegen 4 Bit. Also 4 µs. Es soll nun abgeschätzt werden wie viel Zeit für die Abarbeitung einer entsprechenden Interruptserviceroutine benötigt wird. Dafür wird davon ausgegangen,

dass alle Register, die dabei gelesen/geschrieben werden, 32 Bit breit sein. Weiterhin soll das Dataframe, das gesendet werden soll 8 Byte Nutzdaten tragen. Außerdem wird davon ausgegangen, dass der Controller über eine Bridge mit dem Mikroprozessor verbunden ist. So ergibt sich, dass die CPU bei einem Interrupt zunächst ein Register lesen muss, um zu erkennen, dass der CAN-Controller diesen Interrupt geworfen hat. Danach muss im Controller gelesen werden, um welchen Interrupt es sich handelt. Danach muss der Identifier des empfangenen Frames eingelesen werden, um festzustellen, dass es sich um ein Remoteframe handelt. Anschließend muss der Datalengthcode des Frames eingelesen werden, um festzustellen, wie viele Bytes Nutzdaten angefordert wurden. Anschließend müssen der Identifier, der Datalengthcode und die Nutzdaten des Dataframes in Register geschrieben werden. Hierbei ist zu beachten, dass für die Nutzdaten, max. 8 Byte, 2 Register mit jeweils 32 geschrieben werden müssen. Für das setzen des Datalengthcode wird angenommen, das dieser das Register nicht exklusiv nutzt, also noch andere Einstellungen in diesem Register codiert sind. So muss dieses Register vor dem Schreiben zunächst eingelesen werden, der originale Inhalt des Registers mit dem neuen Wert des DLC verodert werden und anschließend wieder in das Register geschrieben werden. Zuletzt muss in einem Register noch ein Bit gesetzt werden, damit der Controller sendet. Hier wird analog zum Setzen des DLC davon ausgegangen, dass dieses Register mehrere Steuerbits enthält und daher erst gelesen und dann geschrieben werden muss. Es ergibt sich so, 11 Register gelesen/geschrieben werden müssen. Jetzt wird davon ausgegangen, dass für jedes Schreiben/Lesen eines Registers die CPU 20 Takte benötigt, um z.B. den Registerinhalt in einen Array zu übertragen. So ergibt sich, dass 231 CPU-Takte für das Abarbeiten dieser ISR benötigt wird. Auf einen Zeitspanne von 4 µs umgerechnet ergibt sich so eine Zeit von ca. 0,173 µs pro Instruktion, was einem CPU-Takt von 57,75 MHz entspricht. Ein Controller nach dem FullCAN-Konzept kann dies in Hardware, durch eine Auto-Reply-Funktion, lösen und hat hier eindeutig Vorteile. Allerdings ist bei CAN nicht definiert wie viel Zeit zwischen der Anforderung eines Dataframes und dem Senden des selben vergehen darf. Auch wenn das Dataframe direkt gesendet wird, also ohne, dass der Bus dazwischen idle wird, ist durch die Priorisierung der Nachrichten nicht sichergestellt, dass dieses Frame auch tatsächlich zu Ende gesendet wird. So ist diese Zeit nicht so kritisch.

Die andere kritische Situation ist die, dass wenn ein Frame empfangen wurde, dieses rechtzeitig aus den Registern des Controllers gelesen werden muss, bevor diese durch den Empfang eines anderen Frames wieder überschrieben werden. Es soll nun abgeschätzt werden wie viel Zeit benötigt wird um die Register auszulesen bevor diese überschrieben werden. Angenommen wird, dass das Frame ein Dataframe mit 8 Byte Nutzdaten ist. Aus dieses Frame folgt nun ohne, dass der Bus idle wird, ein weiteres Frame. Von diesem wird angenommen, dass es die kürzt mögliche Form hat. Dies entspricht einem Dataframe mit 0 Nutzdaten oder einem Remoteframe. Weiterhin soll dieses Frame ein Standardformat-frame sein. So ergibt sich aus der Summe von SOF, 11 Bit Identifier, RTR-Bit, 6 Bit Controlfield, 16 Bit CRC-field, 2 Bit ACK-field und 7 Bit EOF-field eine Länge von 44 Bit. Weiterhin soll dieses

Frame so aufgebaut sein, dass durch seine Form keine Stuffbits benötigt werden. Auf diese Summe muss nun noch die Dauer des Intermissionfields von 3 Bit drauf gerechnet werden; da die Zeitpunkte des RX_OK bei beiden Frames 1 Bit vor dem Ende des Frames liegen müssen diese nicht von der Summe abgezogen werden. Es verbleiben also insgesamt 47 Bit. Bei einer Bitrate von 1 MBit entspricht dies einer Zeitspanne von 47 µs. Hier wird nun davon ausgegangen, dass zunächst das Register der Bridge gelesen werden muss, danach das Register des Controllers welches den Interrupt angibt, dann ein Register für den Identifier, eines für den Datalengthcode und 2 Register für Nutzdaten. Insgesamt also 7 Register gelesen werden müssen. Auch hier wird davon ausgegangen, dass die CPU pro Registeroperation weitere 20 Takte benötigt. So ergeben sich 147 Takte für die Abarbeitung der ISR. Dies entspricht 0,319 µs oder einem CPU-Takt von 3,127 MHz. In dieser Situation hat auch der FullCAN-Controller wieder Vorteile, da diese meist über Empfangspuffer verfügen. Allerdings ist diese Situation nicht als sehr kritisch anzusehen, da diese bereits ab errechneten 3,127 MHz nicht mehr auftreten kann.

Ein weiterer Aspekt, der beachtet werden sollte, ist der Komfort für den Anwender des Controllers sowie der daraus resultierende Hardwareaufwand. Controller, die nach dem FullCAN-Konzept arbeiten bieten einen großen Komfort für den Anwender, da diese sehr viel durch Hardware lösen, was sonst durch eine Interrupt-Service-Routine(ISR) gelöst werden muss. Durch diesen Mehraufwand steigt allerdings auch der Verbrauch an Hardware, die der FPGA zur Verfügung stellt. Controller nach dem BasicCAN-Konzept hingegen bieten weniger Komfort für den Anwender, da diese über sehr wenig bis keine Hardwareunterstützung für die Annahme von Nachrichten bieten. Dafür ist der zusätzliche Verbrauch an Hardware im FPGA allerdings geringer.

Es ist also festzustellen, dass der Mehraufwand für das FullCAN-Konzept in Verbindung mit einer schnellen CPU nicht nötigt ist. Ein reiner BasicCAN-Controller aber zu wenig Komfort bietet. Daher fällt die Entscheidung hier darauf, einen Mittelweg zwischen diesen beiden Konzepten zu designen. Dazu wird der Controller mit mehreren Einheiten für das Acceptance-Filtering ausgestattet. Dadurch ist es möglich, dass nicht bei jeder Nachricht durch eine ISR festgestellt werden muss, ob diese von Belang ist. Weiterhin wird nur ein einfacher Transmit-Puffer implementiert. Diese Lösung lässt sich mit wenig Aufwand implementieren und bietet trotzdem noch ein wenig Komfort für den Anwender.

3.2 Partitionierung des Controllers

Der Controller ist unterteilt in:

- **CAN-CORE**: Dieser umfasst alles, was in der Spezifikation definiert ist. Mit Ausnahme des Acceptance-Filterings.

- **Acceptance-Filter**: Diese übernehmen die Annahme von Nachrichten.
- **Transmit-Logik**: Diese Logik ist für den Versandt von Nachrichten zuständig.
- **Mikroprozessor-Interface**: Hierin sind die Register und deren Mapping definiert zum Mikroprozessor des SoC definiert.

Abbildung 3.1 zeigt die Aufteilung des Controllers.

Durch die Abkopplung des CAN-CORE vom Rest des eigentlichen Controllers ist es möglich den Controller anzupassen oder komplett zu verändern. Ebenso ist es möglich den CAN-CORE als eigenständigen Controller zu verwenden. So wäre es denkbar einen FPGA für spezielle Filteroperationen zu nutzen und die Ergebnisse direkt über CAN zu versenden. In diesem Fall wäre ein kompletter CAN-Controller unnötig.

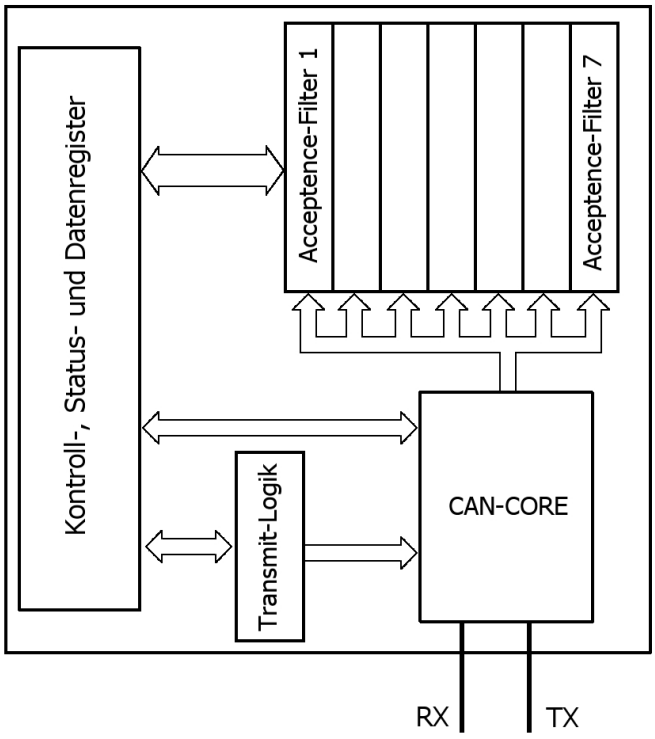


Abbildung 3.1: Aufbau des CAN-Controllers

Der CAN-CORE ist wiederum aufgeteilt in:

- **Baudrate-Prescaler**: erzeugt lediglich das Clock-Enable für die Bittiming-Logik. Der Vorteiler für diese Komponente ist einstellbar.

- **Bittiming-Logik**: erzeugt das Clock-Enable für den Bitstream-Prozessor, legt zum richtigen Zeitpunkt das, vom Bitstream-Prozessor vorgegebene, Bit auf den Bus und gibt das gesamplete Bit an den Bitstream-Prozessor zurück.
- **Bitstream-Prozessor**: zuständig für das Framing, das Serialisieren und Deserialisieren von Nachrichten, das Faultconfinement usw.; diese Einheit umfasst den Data-Link-Layer aus der CAN-Spezifikation.

Abbildung 3.2 zeigt diese Unterteilung.

Baudrate-Prescaler und Bittiming-Logik stellen zusammen den Physical-Layer aus der CAN-Spezifikation dar. Die Funktionen des Data-Link-Layers übernimmt der Bitstream-Prozessor.

Der Baudrate-Prescaler ist als Extra-Komponente ausgegliedert damit dieser bei Bedarf ganz weggelassen werden kann oder durch einen nicht frei einstellbaren Prescaler ersetzt werden kann.

Ebenso ist die Bittiming-Logik vom Bitstream-Prozessor abgegrenzt. Diese ist in diesem Fall frei einstellbar. Für den Fall, dass aber nun der Controller kein frei einstellbares Bittiming benötigt kann diese durch eine angepasste Bittiming-Logik ersetzt werden.

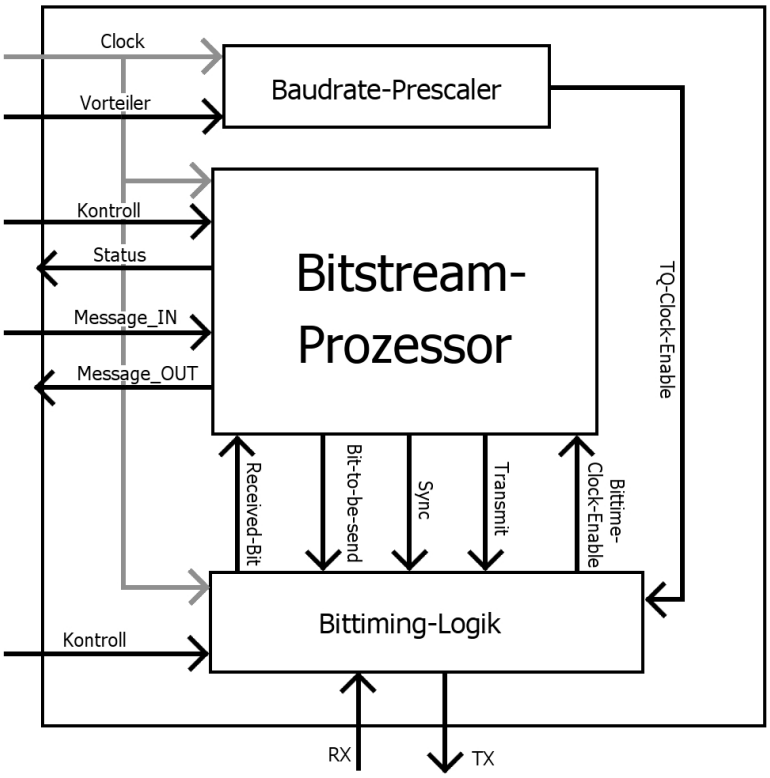


Abbildung 3.2: Aufbau des CAN-Cores

3.3 Bitstreamprozessor

In diesem Abschnitt soll der Aufbau des Bitstream-Prozessors erläutert werden.

In der Analyse hat sich gezeigt, dass das innerhalb des CAN-Protokolls Funktionen existieren die als einzelne Einheiten realisiert werden müssen. Diese sind das Fault-Confinement, das Bitstuffing und die Berechnung der CRC-Sequenz. Für diese Aufgaben werden daher einzelne Komponenten entworfen.

Der Rest des Protokolls ließe sich prinzipiell in einem großen Zustandsautomaten zusammenfassen. Dieser Automat würde allerdings sehr unübersichtlich und würde die Pflege und Erweiterung des Codes verkomplizieren.

Statt dessen wird hier das Protokoll in kleine und überschaubare Automaten zerlegt.

3.3.1 Zerlegung des CAN-Protokolls

Zerlegt wird das Protokoll in eine Einheit, die die Auswahl trifft welches Frame gerade gesendet bzw. empfangen wird, sowie Einheiten, die den Ablauf der einzelnen Frames behandeln.

Dabei zeigt sich, dass das Overloadframe und das Active-Errorframe die gleiche Form besitzen. Lediglich die Auswirkungen auf das Faultconfinement sind bei diesen Frames unterschiedlich. Overloadframe und Active-Errorframe lassen sich aber noch weiter in ihr jeweiliges Flag und den Delimiter zerlegen. Hierdurch lässt sich der Delimiter auch noch für das Passive-Errorframe verwenden. Für das Passive-Errorframe ist so lediglich noch eine Einheit für das Passive-Errorflag nötig.

Der Interframespace lässt sich in eine Einheit für das Intermissionfield und eine Einheit für das Suspended-Transmissionfield zerlegen. Für das Bus-Idle-field ist keine extra Einheit nötig.

Dataframe und Remoteframe haben beide, bis auf kleine Abweichungen, die gleiche Form und werden deshalb zu einer Einheit zusammengefasst. Diese Einheit hat die Kontrolle über die Einheit für das Bitstuffing und das CRC. Diese Einheit kann sowohl Senden als auch Empfangen.

Weiterhin wird eine Einheit benötigt, mit dem der Controller nach dem Reset oder nach dem Einschalten erkennt wann der Bus Idle ist.

Es ergeben sich folgende Einheiten, hier Units genannt, mit entsprechenden Eigenschaften:

Faultconfinement-Unit Diese Unit regelt die Fehlerbehandlung des CAN-Controllers. Sie beinhaltet die Fehlerzähler, den Automaten für die Fehlerzustände und das Regelwerk, um die Fehlerzähler zu setzen. Diese Unit erhält von der Streamcontrol-Unit die Information ob ein Fehler vorliegt und was dies für ein Fehler ist. Dies ist nötig da im Regelwerk, neben der Unterscheidung von Sende- und Empfangsfehlern, Ausnahmen für einige Fehler existieren. Weiterhin erhält diese Unit von der Streamcontrol-Unit die Information, ob eine Nachricht erfolgreich versendet oder empfangen wurde. Die Faultconfinement-Unit gibt an die Streamcontrol-Unit nur den Fehlerzustand(Faultstate) weiter. Zum Interface hin gibt sie die Werte der Fehlerzähler, TEC und REC, weiter. Einstellbar von Außen ist an dieser Unit nur, ob diese durch den Empfang von 128*11 Bit wieder aus dem Bus-Off-Zustand zurückkehren darf.

Streamcontrol-Unit Diese Unit steuert den Ablauf der Frames. Diese Unit legt fest welches Frame gerade auf dem Bus liegt, schaltet die entsprechende Unit für dieses ein und

nimmt die Status-Meldungen dieser Unit entgegen, um diese an die Faultconfinement-Unit weiterzuleiten sowie um daraus, und dem Fehlerzustand der Faultconfinement-Unit, das Folgeframe zu berechnen. Weiterhin legt diese Unit für die Bittiming-Logik fest wann diese welchen Synchronisation-Algorithmus benutzen soll. Diese Unit speichert in einem Register, ob der Controller Sender oder Empfänger ist. Im Falle eines Data- oder Remoteframes übergibt sie allerdings die Kontrolle über dieses Register an die Data-Remote-frame-Unit. Weiterhin übergibt sie bei Data- und Remoteframe die Kontrolle über das zu sendende Bit an die Data-Remote-frame-Unit. Bei allen anderen Units entfällt diese Übergabe, da diese alle eine definierte Form haben.

Active-Error-Overloadflag-Unit Diese Unit bildet die Form des Active-Errorflags und das Overloadflags ab. Diese Unit besteht aus einem einfachen Zustandsautomaten, der 6 dominante, aufeinander folgende Bits erkennt. Diese Unit meldet an die Streamcontrol-Unit ob sie ihren Endzustand erreicht hat oder einen Fehler entdeckt hat. Letzteres gilt für den Empfang eines rezessiven Bits.

Passive-Errorflag-Unit Diese Unit bildet die Form des Passive-Errorflags nach. Diese Unit erkennt entweder eine Folge von sechs aufeinander folgenden dominanten oder rezessiven Bits. Diese Unit teilt der Streamcontrol-Unit mit, dass sie ihren Endzustand erreicht hat und ob ein dominantes Bit empfangen wurde. Letzteres dient einer der Ausnahmen im Faultconfinment.

Error-Overloaddelimiter-Unit Diese Unit gibt die Form des Errordelimiters, sowohl Active- als auch Passive-Errordelimiter, an. Diese Unit erreicht ihren Endzustand nach einer Folge von Acht rezessiven Bits. Das Erreichen ihres Endzustandes teilt diese der Streamcontrol-Unit mit. Weiterhin zählt diese Einheit auch aufeinander folgende dominante Bits. Dabei meldet sie beim Empfang eines achten dominanten Bits dieses an die Faultconfinement-Unit. Ebenso teilt sie der Streamcontrol-Unit mit, ob das erste empfangene Bit ein dominantes Bit ist. Wenn nach dem Empfang eines rezessiven Bits ein dominantes Bit empfangen wird teilt sie die ebenfalls der Streamcontrol-Unit als Fehler mit. Mit der Ausnahme: wenn dies im achten rezessiven Bit geschieht, teilt sie dies der Streamcontrol-Unit nicht als Fehler sondern als Erkennen eines Overloadflags mit.

Suspended-Transmissionfield-Unit Diese Unit bildet die Form des Suspended-Transmission-fields nach. Diese Unit besteht aus einem Zustandsautomaten mit Acht Zuständen, die jeweils für ein rezessives Bit stehen. Diese Unit teilt der Streamcontrol-Unit entweder das Erreichen ihres Endzustandes oder, nach dem Empfang eines dominanten Bits, das Erkennen eines Start-Of-Frame mit.

Intermissionfield-Unit Diese Unit bildet das Intermissionfield nach. Diese Unit besteht aus einem Zustandsautomaten mit drei Zuständen. Diese Zuständen stehen jeweils für ein Bit. Sie teilt der Streamcontrol-Unit beim Empfang eines dominanten Bits im ersten oder zweiten Bit das Erkennen eines Overloadflags mit; den Empfang eines dominan-

ten Bits im dritten Bit teilt sie dies der Streamcontrol-Unit als Start-Of-Frame mit. Beim Empfang eines rezessiven Bit dritten Bit teilt sie das Erreichen ihres Endzustandes mit.

Bus-Unknown-Unit Diese Unit bildet kein Frame bzw. Teilframe nach und ist auch nicht sofort aus der Spezifikation des CAN-Protokolls ersichtlich. Sie bildet eine Eigenschaft des CAN-Protokolls nach, nachdem der Bus nach dem Empfang von 11 aufeinander folgenden rezessiven Bit idle ist. Diese 11 Bit entsprechen der Summe aus entweder ACK-Delimiter, End-of-Frame-field und Intermission-field oder Active-, Passive- oder Overloaddlimiter und Intermission-field. Diese Unit erwartet eine Folge von 11 rezessiven Bits. Beim Empfang des elften rezessiven Bits teilt sie dies der Streamcontrol-Unit als Erreichen ihres Endzustandes mit. Diese Unit wird für den Einsprung in die Kommunikation nach dem Reset sowie zur Erkennung von 128*11 Bit für die Rückkehr aus dem Bus-Off verwendet.

Data-Remote-frame-Unit Diese Unit bildet das Dataframe und das Remoteframe nach und ist für deren Empfang und Versand zuständig. Diese Unit teilt der Streamcontrol-Unit mit, ob und welche Fehler aufgetreten sind, das Erreichen ihres Endzustandes, den erfolgreichen Versand oder Empfang einer Nachricht sowie das Bit, das gerade gesendet werden soll. Zudem gibt diese Unit die Interrupts für Empfang von Versand von Nachrichten an. Diese Unit übernimmt die Kontrolle über das Register, welches angibt, ob der Controller gerade im Sende- oder Empfangsmodus ist. Weiterhin steuert diese Unit wann die Bitstuffing-Unit und die CRC-Unit aktiviert sind. Diese Unit hängt aber auch von der Bitstuffing-Unit ab. Wenn die Bitstuffing-Unit mitteilt, dass das nächste Bit ein Stuffbit sein muss, hält diese Unit ihren aktuellen Zustand und nimmt das zu sendende Stuffbit entgegen. Diese Unit hat ebenfalls die Kontrolle über die Schieberegister, die für den Empfang und den Versand von Nachrichten zuständig sind.

Bitstuffing-Unit Diese Unit ist für die Einhaltung der Bitstuffingregel zuständig. Diese Unit wird beim Empfang eines fünften gleichförmigen Bits der Data-Remote-frame-Unit mitteilen, dass als nächstes Bit ein Stuffbit zu erfolgen hat, sowie das Ensprechende Bit, das als Stuffbit dient. Den Verstoß gegen die Stuffing-Regel teilt sie ebenfalls der Data-Remote-frame-Unit mit.

CRC-Unit Diese Unit berechnet die CRC-Sequenz. Diese Sequenz berechnet sie mit Hilfe eines Schieberegisters. Die errechnete Sequenz teilt sie der Data-Remote-frame-Unit mit. Die Data-Remote-frame-Unit kann dieser Unit mitteilen, dass sie entweder ihre Berechnung anhalten soll, z.B. bei einem Stuffbit, oder dass diese Unit das Register weitershiften soll. Letztes wird für den Versand der CRC-Sequenz benötigt.

Abbildung 3.3 zeigt die Einzelkomponenten des Bitstream-Prozessors und deren Verbindung untereinander. Hierbei ist zu beachten, dass das empfangene Bit (Received-Bit) nicht durch die Streamcontrol-Unit geleitet wird, sondern direkt an alle Units verteilt wird. Gleiches gilt für die Clock und das Clock-Enable. Diese wurden in der Abbildung weggelassen.

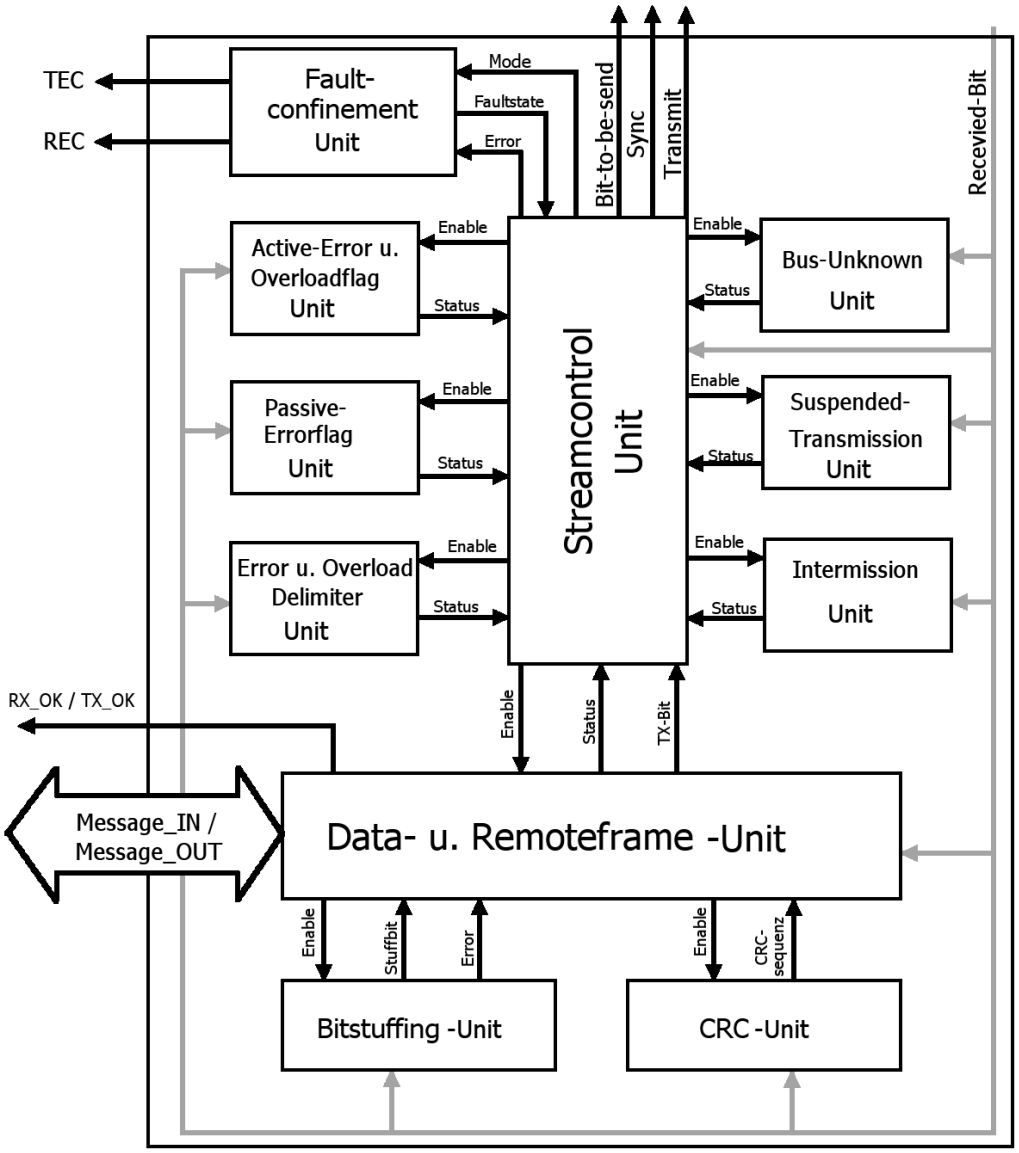


Abbildung 3.3: Aufbau des Bitstreamprozessors

3.3.2 Automat der Streamcontrol-Unit

Durch die, im vorigen Abschnitt erwähnte, Mehrfachverwendung der Units, z.B bei Error- und Overload-Delimiter, ergibt sich ein Problem. Dies soll hier kurz Anhand des Overloadframes erklärt werden.

Es wäre möglich nur einen Zustand zu nutzen und in diesem dann zuerst die Active-Error-Overloadflag-Unit zu enablen und, nach dem dieses ihren Endzustand erreicht hat, die Error-

Overloaddelimiter-Unit zu enablen. Dadurch wird der Automat der Streamcontrol-Unit weniger komplex, da dieser weniger Zustände enthalten würde. Allerdings macht dies das Debugging unübersichtlicher, da in diesem Fall jeweils zwei unterschiedliche Units überwacht werden müssen. Daher werden diese zusammengesetzten Frames in Teilzustände zerlegt, in denen nur jeweils eine Unit enabled wird. Hierdurch wird der Automat zwar komplexer aber das Debugging einfacher.

Ein weiteres Problem ergibt sich beim Start-Of-Frame. Dieses kann entweder selber gesendet werden oder durch eine andere Unit erkannt werden, z.B im Intermissionfield. Dadurch müsste die Data-Remote-frame-Unit zwei Startzustände haben. Dieses liesse sich wahrscheinlich sogar in Code umsetzen, allerdings macht dies zum einen das Debugging wieder unnötig kompliziert und zum anderen widerspricht dies der Definition von Zustandsautomaten. Da diese nur einen Startzustand haben dürfen. Eine andere Möglichkeit wäre die Data-Remote-frame-Unit in den Zuständen, in denen ein Start-Of-Frame erkannt werden kann, mitlaufen zu lassen. Dadurch wären dann aber pro Zustand wieder mehrere Units aktiv und dies soll hier vermieden werden. Das Start-Of-Frame wird daher aus der Data-Remoteframe-Unit ausgegliedert.

Weiterhin könnten in diesem Automat für Overloadframe und Active-Errorframe jeweils die gleichen Zustände verwendet werden. Der einzige Unterschied zwischen diesen Frames besteht im Setzen der Fehlerzähler wenn diese gesendet werden. Hierdurch wird der Automat weniger komplex. Dies wirkt sich aber auch wieder nachteilig auf das Debugging aus, da hierbei dann erst nachgeschaut werden muss wie sich der Stand der Fehlerzähler geändert hat, um zwischen einem Overloadframe und einem Active-Errorframe zu unterscheiden. Daher werden sowohl für die Teile des Active-Errorframes als auch für die des Overloadframes Zustände eingefügt.

3.3.3 Datenpfad der Data-Remote-frame-Unit

Hier soll der Aufbau des Datenpfades der Data-Remote-frame-Unit erklärt werden. Dieser Datenpfad ist von zentraler Bedeutung, da dieser für das Serialisieren der zu versendenden Nachricht sowie für das De-serialisieren der empfangenen Nachricht zuständig ist.

Für das Serialisieren und De-serialisieren werden hier Schiebenregister verwendet. Jedes dieser Schieberegister wird dabei für nur ein Teilstück von Data- oder Remoteframe verwendet. Insgesamt werden Zwölf Schieberegister verwendet:

- 1 Register für den Base-Identifier, das SRR- bzw. RTR-Bit und das IDE-Bit.
- 1 Register für den Extended-Identifier und das Extended-RTR-Bit.
- 1 Register für das R1-Bit + und den Datalengthcode.

- 8 Register für die Bytes der Nutzdaten.
- 1 Register für die empfangene CRC-Sequenz.

Diese Register werden durch den Steuerautomaten, entsprechend des Abschnitts des Frames, ausgewählt und weitergeshiftet. Diese Register werden immer nach Links geshiftet. Für den Fall, dass ein Stuffbit erwartet wird oder gesendet werden soll, wird das Shiften ausgesetzt. Wenn eines dieser Register selektiert ist, gibt es sein MSB an den Steuerautomaten aus. Dieser hat dann die Wahl dieses Bit oder ein Stuffbit zu senden, wenn der Automat im Sende-Modus ist. Wenn das empfangene Bit kein Stuffbit ist wird dieses immer an Stelle des LSB in das ausgewählte Register eingetragen, unabhängig davon, ob dieser Automat gerade Sender oder Empfänger ist. Eine Sonderregelung betrifft allerdings die Register für die CRC-Sequenzen. Das zusendende Bit kommt hierbei aus der CRC-Unit, während das empfangene Bit immer in das Register des Datenpfades gespeichert wird. Abbildung 3.4 zeigt diesen Aufbau sehr vereinfacht.

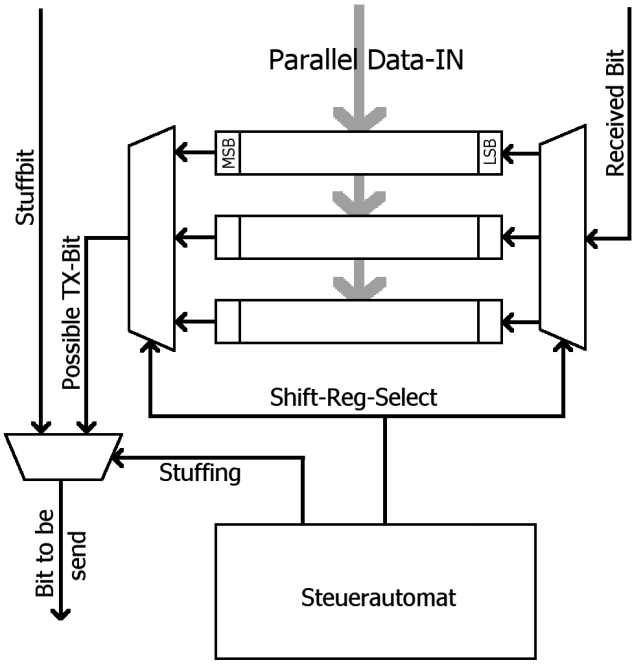


Abbildung 3.4: vereinfachter Datenpfad der Data-Remote-frame-Unit

3.4 Baudrateprescaler

In diesem Abschnitt soll der Baudrateprescaler erläutert werden. Der Baudrateprescaler besteht aus einen einfachen Aufwärtszähler. Dieser Zähler kann durch eine Vergleichswert zurückgesetzt werden. Dieser Vergleichswert ist die Prescale-Rate. Wenn der Zähler und die

Prescale-Rate den gleichen Wert haben, wird der Zähler ein Enable-Signal ausgeben und im nächsten Takt wieder bei Null beginnen. Das so erzeugte Enable wird für die Bittiminglogik verwendet und stellt die Dauer eines TQ dar.

3.5 Bittiminglogik

In diesem Abschnitt soll die Bittiminglogik erläutert werden. Die Bittiming-Logik besteht aus zwei Teilen. Einer Logik, die das eigentliche Bittiming ausführt, und einer, die die Flanken auf dem Bus detektiert, Bus-Monitor genannt.

Die Logik für das Bittiming ist ein Zustandsautomat mit vier Zuständen. Diese Zustände repräsentieren die Segmente aus dem Bittiming der CAN-Spezifikation (vergleiche Kapitel 2.6.19). Dabei wird der Übergang zwischen diesen Zuständen durch einen Counter berechnet. Dieser Counter gibt an wie viele TQ abgelaufen sind und taktet mit dem Enable-Signal des Baudrateprescalers.

Um festzustellen, wann ein Zustandswechsel erfolgt wird für die einzelnen Zustände jeweils ein Vergleichswert genutzt. Dieser gibt an nach wie vielen TQ der Zustand verlassen wird. Während des Übergangs von Phase1-Buffer-Segment zu Phase2-Buffer-Segment wird ein Enable-Signal generiert. Dieses Signal enabled das Register in dem der aktuelle Bus-Pegel gespeichert wird.

Wenn eine Flanke vom Bus-Monitor gemeldet wird, berechnet der Zustandsautomat aus dem aktuellen Zustand und dem Stand des Counters für die Zeitquanten die Distanz zwischen Flanke und SYNC_SEG. Mit dieser Distanz wird dann entweder der Vergleichswert für das Phase1-Buffer-Segment verlängert oder das Phase2-Buffer-Segment verkürzt. Dies erfolgt nach den Synchronisierungs-Regeln aus der CAN-Spezifikation.

Der Bus-Monitor nutzt ein Register in welches er in jedem Takt einmal den Bus-Pegel übernimmt. Diesen Wert vergleicht er mit dem Wert, der beim letzten Samplepoint eingelesen wurde. Der Bus-Monitor meldet eine Flanke wenn der, von ihm festgestellte, Wert dominant und der des letzten Samplepoints rezessiv ist.

Weiterhin bietet die Bittiming-Logik an, den Bus-Pegel mittels einer Mehrheitsentscheidung zu festzustellen. Dafür wird bei jedem TQ der aktuelle Bus-Pegel in ein 3 Bit Schieberegister übernommen. Der Bus-Pegel wird dann aus diesen Bits bestimmt. Dabei ergibt die Mehrheit dieser Bits den Wert des Pegels an, welcher dann übernommen und an den Bitstream-Prozessor weitergeleitet wird. Um diese Funktion zu nutzen muss diese aktiviert werden.

Die Bittiminglogik stellt außerdem das Clock-Enable für den Bitstreamprozessor. Dieses wird einen Takt nach dem Samplepoint generiert. Diese Zeit stellt die Information-Processing-Time dar. Diese darf nach der CAN-Spezifikation kleiner oder gleich 2 TQ sein. Hier beträgt

diese höchstens 1 TQ wenn der Baudrateprescaler auf einen Wert von Null eingestellt ist. Ansonsten ist die Zeit kürzer.

3.6 Acceptance-Filtering und Transmission-Gluelogic

In diesem Abschnitt sollen die Transmission-Gluelogic und die Logik für das Accpetance-Filtering erläutert werden. Diese bestehen jeweils aus Schaltnetzen, da die Register, die diese nutzen, mit dem Generator des Mikroprozessor-Interfaces erzeugt werden.

3.6.1 Transmission-Gluelogic

Die Transmission-Gluelogic nimmt den Inhalt von Registern des Mikroprozessor-Interfaces entgegen und leitet diesen an den CAN-Core weiter. Dies tut sie wenn dazu in einem Register das TX-Request-Bit gesetzt ist. Anderen Falls setzt sie die Ausgänge zum CAN-Core auf 0. Das TX-Request-Bit leitet sie ebenfalls an den CAN-Core weiter. Wenn der CAN-Core ein TX_OK, also das erfolgreiche Versenden der Nachricht, meldet setzt die Transmission-Gluelogic das TX-Request-Bit wieder zurück.

3.6.2 Acceptance-Filtering

Die Einheiten des Acceptance-Filterings nehmen die Nachrichten vom CAN-Core entgegen und speichern diese in Registern des Interfaces. Wenn diese Einheiten eine Nachricht akzeptieren setzen sie einen entsprechenden Interrupt. Diese Einheiten greifen für das Filtern der Nachricht jeweils auf mehrere Register zu:

- **ID-TAG-Register**: In diesem Register kann der Identifier der akzeptierten Nachricht angegeben werden.
- **ID-MASK-Register**: In diesem Register werden die Bits des Identifiers angegeben auf die gefiltert werden soll.
- **DLC-Tag-Register**: In diesem Register kann angegeben werden wie viele Bytes die Nachricht hat, die akzeptiert werden soll.
- **DLC-MASK-Register**: In diesem Register werden die Bits angegeben, die beim Filtern des Datalengthcode beachtet werden sollen.

Das Filtern erfolgt dabei durch eine logische UND-Verknüpfung zwischen TAG- und MASK-Regsiter, sowie zwischen MASK-Register und dem entsprechenden Teil der Nachricht. Sind die Ergebnisse dieser beiden Verknüpfungen identisch so gilt die Nachricht als akzeptiert. Die Nachricht wird allerdings erst gespeichert wenn der CAN-Core ein RX_OK, also den erfolgreichen Empfang einer Nachricht, meldet. Eine Acceptance-Filtering-Einheit, die eine Nachricht akzeptiert hat, wird dann diese Nachricht in ihren Registern im Interface speichern und in einen Register ein RX_OK-Bit setzen. Dieses Register löst dadurch einen Interrupt aus.

3.7 Mikroprozessor-Interface

Beim Entwurf des Interfaces kann hier nur Einfluss darauf genommen werden, welche und wie viele Register verwendet werden. Dabei soll hier vor allem geachtet werden, dass Kontrollregister, Statusregister und Datenregister von einander getrennt sind.

Die Register des Mikroprozessor-Interfaces sind alle 32 Bit breit. Aber nicht alle Einstellungen, die der Controller benötigt, sind 32 Bit breit. Dies ist beim Datalengthcode der Fall. Dieser ist nur 4 Bit breit. Die Acceptance-Filtering-Units und auch die Transmission-Gluelogic benötigen aber die Angabe dieses Datalengthcodes. Um aber Register einzusparen sollen sich die Acceptance-Filtering-Units und die Transmission-Gluelogic hierfür Register teilen.

Der Controller erhält folgenden Register:

- **CAN_STI**: CAN-Status-Interrupt: Dieses Register zeigt alle Interrupts an, die der Controller ausgegeben hat.
- **CAN_GIE**: CAN-General-Interrupt-Enable: In diesem Register können die Interrupts des Controllers aktiviert/deaktiviert werden.
- **CAN_EN**: CAN-Enable: In diesem Register können Teile oder Funktionen des Controllers aktiviert/deaktiviert werden.
- **CAN_BT**: CAN-BitTiming: In diesem Register können die Werte für die Bittiming-Logik eingestellt werden.
- **CAN_ERR**: CAN-Error: In diesem Register können die Fehlerzähler des Faultconfinements ausgelesen werden.
- **CAN_TX_ID**: CAN-Transmission-Identifier: In diesem Register wird der Identifier der zu übertragenden Nachricht eingegeben.
- **CAN_TX_MSG_LOW**: CAN-Transmission-Message-Low: In diesem Register können die unteren vier Byte der zu übertragenen Nachricht angegeben werden.

- **CAN_TX_MSG_HIGH**: CAN-Transmission-Message-High: In diesem Register können die oberen vier Byte des zu übermittelnden Nachricht angegeben werden.
- **CAN_DLC_TAG**: CAN-DataLengthCode-Tag: In diesem Register könne die DLC-Tags für die Acceptance-Filtering-Units sowie der DLC für die zu übertragenden Nachricht angegeben werden.
- **CAN_DLC_MASK**: CAN-DataLengthCode-Mask: Dieses Register beinhaltet die DLC-Masken für die Acceptance-Filtering-Units.
- **CAN_DLC_OUT**: CAN-DataLengthCode-Out: In diesem Register werden die Datalengthcodes der akzeptierten Nachrichten ausgegeben.
- **CAN_ACC_ID_TAG***: CAN-Acceptance-Filtering-Identifier-Tag: Diese Register beinhalteten den Identifier-Tag für die entsprechenden Acceptance-Filtering-Units.
- **CAN_ACC_ID_MASK***: CAN-Acceptance-Filtering-Idenitifier-Mask: Diese Register beinhalteten die Identifier-Masken für die entsprechenden Acceptance-Filtering-Units.
- **CAN_ACC_ID_OUT***: CAN-Acceptance-Filtering-Identifier-Out: Diese Register geben den Identifier der Nachricht aus, die von der entsprechenden Acceptance-Filtering-Unit akzeptiert wurde.
- **CAN_ACC_MSG_LOW_OUT***: CAN-Acceptance-Filtering-Message-Low-Out: Diese Register geben die unteren vier Byte der, von der entsprechenden Acceptance-Filtering-Unit akzeptierten, Nachricht aus.
- **CAN_ACC_MSG_HIGH_OUT***: CAN-Acceptance-Filtering-Message-High-Out: Diese Register geben die oberen vier Byte der, von der entsprechenden Acceptance-Filtering-Unit akzeptierten, Nachricht aus.

Die Register für die Acceptance-Filtering-Unit sind jeweils in siebenfacher Ausführung vorhanden. Diese Anzahl der Acceptance-Filtering-Units ergab sich daraus, dass alle Register des Mikroprozessor-Interfaces 32 Bit breit sind und eine Acceptance-Filtering-Unit jeweils 4 Bit für das Einstellen des zu filternden Datalengthcodes benötigt. Ebenso benötigt die Transmission-Gluelogic 4 Bit zum Einstellen des zu sendenden Datalengthcodes. Da der Controller nur über Einheit zum Senden verfügen soll, blieben so noch 28 ungenutzte Bits in einem Register übrig. Es erschien daher sinnvoll dieses Register sowohl für den Datalengthcode der Transmission-Gluelogic als auch für die Tags der Acceptance-Filtering-Units zu verwenden. So ergab sich eine Gesamtanzahl von sieben Acceptance-Filtering-Units.

3.8 Erweiterungen am Mikroprozessor-Interface

In diesem Abschnitt sollen die Änderungen am Mikroprozessor-Interface erläutert werden, die nötig sind um einige Funktionen zu ermöglichen, die vom Controller benötigt werden.

Der CAN-Controller liefert nicht bei jedem Takt etwas das gespeichert werden soll. Die Register des Mikroprozessor-Interfaces speichern allerdings bei jedem Takt. Damit keine falschen Ergebnisse in die Register übernommen werden wurde dazu ein neuer Registertyp entworfen, der über ein peripherieseitiges Write-Enable verfügt.

Damit mehrere Einheiten sich ein Register für ihre Ausgabe teilen können, wurde ein Registertyp entworfen, der über ein peripherieseitiges, bitweises Write-Enable verfügt.

Zudem wurde ein Registertyp entworfen, der über ein peripherieseitiges, bitweise Reset-On-Read verfügt. Dieser Registertyp ermöglicht es, dass Teile des Controllers Bits zurücksetzen können ohne den Rest des Registers zurückzusetzen. Dies ist z.B. beim Zurücksetzen des TX-Request-Bits durch die Transmission-Gluelogic von Bedeutung.

Diese neuen Registertypen zogen Änderungen am Generator des Mikroprozessor-Interface nach sich.

4 Realisierung

Nachdem im vorigen Kapitel das Design des Controllers erklärt wurde, soll hier nun dargelegt werden, wie dieses realisiert wurde. In diesem Kapitel wird erklärt welche Codingkonventionen verwendet wurden, wie der Code in Dateien aufgeteilt ist und ein Codebeispiel angegeben. Zudem wird erläutert welche Werkzeuge verwendet wurden und wie der Controller getestet wurde.

4.1 Implementierungsrichtlinien

In diesem Abschnitt wird erklärt, welche Koventionen für die Umsetzung in VHDL verwendet wurden.

4.1.1 Prozesse

In VHDL können Aktionen sowohl parallel als auch sequentiell ausgeführt werden. Code der nur aus parallelen Anweisungen besteht ist meist sehr unübersichtlich und schwer verständlich, um dies zu umgehen gibt es in VHDL die Möglichkeit Prozesse zu verwenden. Prozesse laufen ebenfalls parallel aber die Anweisungen innerhalb eines Prozesses laufen sequentiell ab.

Bei der Verwendung von Prozessen zur Modellierung von Abläufen sollte allerdings darauf geachtet werden, dass kombinatorische Logik und sequentielle Logik von einander getrennt sind. Als kombinatorische Logik wird Logik bezeichnet, die keine internen Zustände besitzt. Logik also die ein reines Schaltnetz beschreibt. Als sequentielle Logik wird taktabhängige Logik, wie z.B. Register oder Flip-Flops, bezeichnet.

Im Code des CAN-Controllers werden kombinatorische und sequentielle Logik stets von einander getrennt.

4.1.2 Variablen

Jedes Signal, dass in einem Prozess verwendet wird, wird zu Beginn des Prozesses auf eine Variable übertragen. Zuweisungen von Variablen auf Signale werden am Ende des Prozesses vorgenommen.

Bei der Verwendung von Zahlentypen, z.B. Integer, als Variablen, wird dabei immer eine Range, also der Zahlenbereich, den diese annehmen kann, mit angegeben. Dies ermöglicht dem Synthesetool später Hardware einzusparen.

4.1.3 Signal- und Variablennamen

Die Namen von Signalen und Variablen sollen weitgehend selbsterklärend sein. Um die Lesbarkeit des Codes zu erhöhen tragen alle Signale und Variablen eine Endung, welche mit einem Unterstrich vom eigentlichen Namen abgetrennt ist. Dabei werden für Signale folgende Endungen verwendet:

- **_cs**: Steht für **c**locked **s**ignal. Diese Endung tragen abgetaktete Signale, also Signale, die aus einem Flip-Flop oder Register kommen.
- **_ns**: Steht für **n**ew **s**ignal. Diese Endung tragen Signale, die erst noch abgetaktet werden sollen. Also Signale, die in ein Flip-Flop oder ein Register hinein laufen.
- **_s**: Steht für **s**ignal. Diese Endung tragen Signale, die nicht abgetaktet wurden oder nicht abgetaktet werden sollen.

Abbildung 4.1 zeigt dies am Beispiel eines D-Flip-Flops.

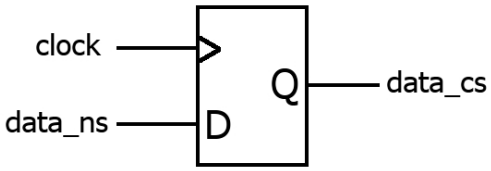


Abbildung 4.1: D-Flip-Flop

Variablen erhalten als Endung immer **_v**, um zu verdeutlichen, dass es sich um eine **V**ariable handelt.

4.1.4 Aufteilung in Dateien

Für jede Entity wird eine eigenen Datei angelegt. Diese Datei erhält als Namen den Namen der Entity, welche darin beschrieben ist. So ist z.B. die Entity Baudrateprescaler in der Datei baudrateprescaler.vhd definiert. Weiterhin werden Entities niemals in einer Datei instanziert, in der auch andere Logik in Prozessen beschrieben wird. So ergibt sich eine Trennung zwischen den Definitionen der Entities und deren Instanziierungen.

In diesem Projekt werden Entities nur in drei Dateien instanziiert und mit einander verbunden. Diese Dateien sind:

- **bitstream_processor.vhd**: hier werden die Entities der einzelnen Units instanziiert und verbunden.
- **can_core.vhd**: instanziert die Entities Bitstreamprocessor, Baudrateprescaler und Bittiminglogic.
- **can_controller.vhd** instanziiert die Entity Can_Core, die Entities Acceptance_filter, diese sieben mal, die Entity Transmission sowie das Mikroprozessor-Interface.

4.1.5 Codebeispiel

Um den Einstieg in den eigentlichen Code zu erleichtern soll dieser hier nun an einem Beispiel erklärt werden. Hierfür wurde der Code der Data-Remote-frame-Unit gewählt, da dieser alle Konventionen enthält, die in diesem Projekt verwendet wurden. Dieser Code wird allerdings nur in Auszügen erklärt.

Diese Unit ist als ein Zustandsautomat implementiert. Um den Code aber möglichst kurz und übersichtlich zu halten wurde nicht für jedes Bit dieses Automaten ein Zustand entworfen, sondern nur Zustände für Abschnitte des Frames. Die Anzahl an Bits, für die der Automat dabei einem Zustand verweilt wird durch einen Counter berechnet.

Listing 4.1 zeigt den Code der sequentiellen Logik der Data-Remote-frame-Unit. Hier sind sowohl das Zustandsregister(state_cs), als auch das Register des Indexcounters(index_cs) beschrieben. Diese Register haben beide einen synchronen Reset. Dies ist im gesamten Design des Controllers zu. Getaktet werden diese Register auf steigende Flanke. Dazu wird das Makro **rising_edge(clk)** verwendet. Dies erhöht die Lesbarkeit des Codes und macht die Simulation sicherer, da dieses Makro überprüft, ob das Clocksignal zwischen den steigenden Flanken tatsächlich abgefallen ist. Statt rising_edge() zu verwenden wäre es auch möglich, dies mit clk = '1' **AND** clk'EVENT abzufragen. Dies ist aber, nach persönlicher Meinung, weniger gut lesbar. Zudem wird hier nicht überprüft, ob das Clocksignal tatsächlich den Pegel geändert hat.

```
160 --Zustandsregister und Indexregister:
161   SEQ_LOGIK : PROCESS(clk)
162   BEGIN
163     IF(rising_edge(clk))THEN
164       IF(nreset = '0')THEN
165         state_cs <= BASE_ID;
166         index_cs <= (OTHERS => '0');
167       ELSIF(enable = '1' AND clk_en = '1')THEN
168         IF(state_en_s = '1')THEN
169           state_cs <= state_ns;
170         END IF;
171         IF(cnt_en_s = '1')THEN
172           index_cs <= index_ns;
173         END IF;
174       END IF;
175     END IF;
176   END PROCESS SEQ_LOGIK;
```

Listing 4.1: sequenzielle Logik aus data_remote_frame.vhd

Die Übernahme der neuen Werte in das Zustandsregister sowie in das Registers des Indexzähler hängt von Enablesignalen ab. Da wären zunächst das Enable für den Controller und das Clock-Enablesignal, das die Bitiming-Logik generiert, sowie die Enablesignale, die von der kombinatorischen Logik der Data-Remote-frame-Unit generiert werden ab. Diese sind zum einen **state_en_s** als Enablesignal für das Zustandsregister und zum anderen **cnt_en_s** als Enablesignal für das Indexregister.

Die kombinatorische Logik der Data-Remote-frame-Unit berechnet den Folgezustand des Zustandsautomaten. Diese Logik stellt damit das Übergangsschaltnetz dieses Automaten dar. Dazu verwendet diese den gespeicherten Zustand aus dem Zustandsregister und den Wert des Indexzählers. Listing 4.2 zeigt die Berechnung des Folgezustands. Dieses Listing ist nur ein kleiner Auszug aus dem Code der kombinatorischen Logik, wie an den Zeilennummern des Listings zu sehen ist. Die Registerwerte werden dazu zunächst auf Variablen übertragen, hier **state_v**(für den aktuellen Zustand), **next_state_v**(für den Folgezustand) und **index_v** für den Indexwert. Bei index_v ist zu beachten, dass diese eine Integervariable ist. Daher wird hier der Wert nicht direkt aus dem Register übernommen, sondern erst durch **usuv2int()** in einen Zahlenwert umgerechnet.

```
102   CONSTANT BASE_ID_MAX_INDEX    : integer := 10;
109   CONSTANT BASE_ID_REGISTER      : std_ulogic_vector(3 DOWNTO 0) := "0001";
123   TYPE STATE_TYPE IS (BASE_ID, SRR_RTR, IDE_BIT, EXT_ID, EXT_RTR, R1_BIT, R0_BIT,
          DLC,
299     state_v             := state_cs;
300     next_state_v        := state_cs;
301     index_v             := usuv2int(index_cs);
346     CASE state_v IS
```

```
347         WHEN BASE_ID =>
348           -- 11 Bit
349           limit_v       := BASE_ID_MAX_INDEX;
350           next_state_v  := SRR_RTR;
351           arb_zone_v    := '1';
352           crc_zone_v    := '1';
353           stuff_zone_v  := '1';
354           s_wrd_v       := '1';
355           reg_select_v  := BASE_ID_REGISTER;
356         WHEN SRR_RTR =>
357           --  1 Bit
358           limit_v       := 0;
359           next_state_v  := IDE_BIT;
360           arb_zone_v    := '1';
361           crc_zone_v    := '1';
362           stuff_zone_v  := '1';
363           s_wrd_v       := '1';
364           reg_select_v  := BASE_ID_REGISTER;
566       END CASE;
```

Listing 4.2: kombinatorische Logik aus data_remote_frame.vhd - Berechnung des Folgezustands

Die Auswahl des Folgezustands erfolgt durch eine CASE-Anweisung. In dieser wird für den aktuellen Zustand immer der Folgezustand und ein Limit für den Wert des Indexzählers, der in diesem Zustand angenommen werden kann, gesetzt. Diese werden in **next_state_v** und **limit_v** angegeben. Weiterhin werden hier auch die Enablesignale für die CRC-Unit, in **crc_zone_v**, und die Bitstuffing-Unit, in **stuff_zone_v**, gesetzt, es wird festgelegt, ob dieser Abschnitt des Frames zum Arbitrationfield gehört, in **arb_zone_v**, das entsprechende Register im Datenpfads, in **reg_select_v**, selektiert und darauf eingestellt immer das empfangene Bit wegzuspeichern, die erfolgt durch **s_wrd_v**.

Nach der Auswahl des Folgezustandes wird verglichen, ob der Indexzähler bereits das Limit erreicht hat. Ist dies der Fall, so wird das Enablesignal für das Zusatndsregister gesetzt und der Indexzähler auf 0 zurückgesetzt. Ist das Limit noch nicht erreicht, so wird der Indexzähler um 1 erhöht. Dies ist in Listing 4.3 zu sehen.

```
568       IF(index_v = limit_v)THEN
569         state_en_v  := '1';
570         cnt_en_v    := '1';
571         new_index_v := 0;
572       ELSE
573         cnt_en_v    := '1';
574         new_index_v := index_v + 1;
575       END IF;
576
577       IF(stuff_zone_v = '1')THEN
```

```
578       IF(stuff_bit_v = '1')THEN
579         s_wrd_v            := '0';
580         crc_zone_v         := '0';
583       ELSIF(next_is_stuff_v = '1')THEN
584         state_en_v := '0';
585         cnt_en_v   := '0';
586       END IF;
587     END IF;
```

Listing 4.3: kombinatorische Logik aus data_remote_frame.vhd - Aussetzen des Zustandswechsels

Anschließend wird überprüft, ob das nächste Bit ein Stuffbit ist oder ob aktuell ein Stuffbit auf dem Bus liegt. Dies wird durch zwei Signale von der Bitstuffing-Logik mitgeteilt. Diese sind in die Variablen **next_is_stuff_v** und **stuffbit_v** umgesetzt. Im Fall, dass das nächste Bit ein Stuffbit ist, werden die Enablesignale für das Zustandsregister und den Indexzähler zurückgenommen. Der Zustandsautomat wird also in diesem Moment angehalten. Wenn das aktuelle Bit ein Stuffbit ist, werden die Enablesignale für das Speichern des Bits sowie das Enablesignal für die CRC-Unit zurückgenommen. So werden Stuffbits, für die interne Weiterverarbeitung, aus dem Datenstrom entfernt. Die Rücknahme dieser Enablesignale ist möglich, da diese erst am Ende des Prozesses auf Signale übertragen werden und vorher nur Variablen sind. Da die Abarbeitung innerhalb eines Prozesses sequentiell verläuft hat das Setzen dies Variablen erst Effekt wenn diese am Ende auf Signale umgesetzt werden. Listing A.2 zeigt die Zuweisung von Variablen auf Signale am Ende des Prozesses. Hierbei ist zu beachten, dass Variablen, die intern als Integer benutzt werden, zurück gewandelt wenn diese auf Signale zugewiesen werden. Dies ist in Zeile 644 von Listing A.2 zu sehen.

```
642     state_en_s         <= state_en_v;
643     state_ns           <= next_state_v;
644     index_ns           <= int2usuv(new_index_v, 5);
645     reg_select         <= reg_select_v;
646     cnt_en_s           <= cnt_en_v;
647     s_wrd              <= s_wrd_v;
666   END PROCESS DF_FSM;
```

Listing 4.4: kombinatorische Logik aus data_remote_frame.vhd - Zuweisung von Variablen auf Signale

4.2 Verwendete Werkzeuge

Für die Simulation wird ModelSim SE-64 in der Version 6.3J von Mentor Graphics eingesetzt. Für die Synthese wird ISE WebPACK 10.1.03 der Firma Xilinx verwendet. Eingesetzt werden beide Tools unter Ubuntu 9.04 für X86-64-Architektur .

4.3 VHDL-Simulation und Timingsimulation des Controllers

Für die Simulation des Controllers wurden zwei Testbenches entworfen. Eine Testbench für die Bittiminglogik und den Baudrateprescaler sowie eine Testbench in welcher der komplette Controller getestet wurde. In der Testbench für den Controller wurden zwei Controller instanziiert. Ein Controller als DUT(Device Under Test), welcher den zu testenden Controller darstellt und ein Controller, der als Gegenstelle für das Acknowledgement sorgt. Damit diese beiden Controller kommunizieren können wurde ein Busmedium und entsprechende Tranceiver für diese Controller simuliert. Wo bei hier angemerkt werden muss, dass sowohl bei den Tranceivern als bei dem simulierten Busmedium keine physikalisch korrekte Simulation vorgenommen wurde. Die Laufzeiten der Signale durch die Tranceiver, sowie die auf dem Busmedium wurden hierbei nicht berücksichtigt. Weiterhin wurde ein Testprozess für das DUT entworfen. In diesem Prozess wird für jeden Testfall zunächst ein Printout in der Konsole des Simulators gemacht, welches beschreibt um welchen Test es sich handelt und zu welchem Zeitpunkt in der Simulation dieser stattfindet.

Getestet wurde der Controller indem dieser, also das DUT, darauf eingestellt wurde Daten zu versenden oder zu empfangen. Dabei wurde getestet ob, die Frames des DUT die richtige Form auf dem Bus haben indem der Testprozess ein gleichförmiges Frame, welches händisch generiert wurde, sendet sobald das DUT anfängt zu sende. In diesem Fall kann das DUT nicht erkennen, dass diese Frame auf dem Bus liegt, da beide Frames gleich aussehen müssen. Ein Fehlverhalten des Controllers lässt sich so sehr einfach erkennen. Diese händisch generierten Frames wurden entweder aus Lehrbüchern entnommen oder es wurden Frames gebildet die mit einem Oszilloskop von einem anderen CAN-Controller abgegriffen wurden.

Neben den händisch, Bit für Bit, generierten Frames wurden Frames mit Prozeduren erzeugt und der Controller mit diesen getestet. Außerdem wurde der zweite Controller dazu verwendet Frames zu senden, die das DUT dann erkennen musste. Bei diesen Tests wurde getestet, ob das DUT richtig mit unterschiedlichen Prioritäten umgeht und ob es Fehler, die in die Frames eingebaut wurden, erkennt und diese entsprechend den Fehlerregeln erkennt und behandelt. Um dies in der Simulation leichter zu erkennen wurde eine Prozedur geschrieben, die eine simple ISR nachbildet. In dieser werden die Register des DUT ausgelesen und deren Inhalt in der Konsole ausgegeben und der Zeitpunkt vermerkt an dem dies passiert ist. Abbildung 4.2 zeigt das Printout eines dieser Testfälle.

```
#####################################################################    TESTCASE 4    ##########################
# BIT-ERROR_2:  Bit-Error ausserhalb der Arbitrierungsphase@228968750 ps
# ***********************************    ISR START     ************************************************
# DUT: CAN-Controller: CAN_STI:  INTERRUPT(s): 00000000000000010000100000000000 @271843750 ps
# START-OF-FRAME
# BIT-ERROR
# TEC: 8   REC: 1
# ***********************************    ISR ENDE   @272000000 ps ****************************************
#####################################################################################################################
```

Abbildung 4.2: Printout der Testbench

Weiterhin wurde getestet, ob die Einzelkomponenten des Controllers richtig zusammenarbeiten. Dies wurde anhand der Signale im Wave-Window des Simulators erledigt. Abbildung 4.3 zeigt einen kleinen Ausschnitt aus dem Wave-Window. In diesem ist dargestellt, wie der Automat der Data-Remote-frame-Unit während eines Stuffbits ausgesetzt wird. Dieses Stuffbit tritt am Ende des CRC-fields auf und ist somit das letztmögliche Stuffbit im Frame. Es ist zu erkennen wie der Wert des Indexzählers(index_cs) zunächst in regelmäßigen Abständen inkrementiert wird. Als der Indexzähler einen Wert von 14 angenommen hat wird dieser jedoch für einen längeren Zeitraum beibehalten. Das Signal next_is_stuff nimmt zu diesem Zeitpunkt einen Wert von 1 an. Das Signal cnt_en_s in der Data-Remote-frame-Unit geht daraufhin auf 0. Der Indexzähler wird damit für den Zeitpunkt des Stuffbits angehalten.

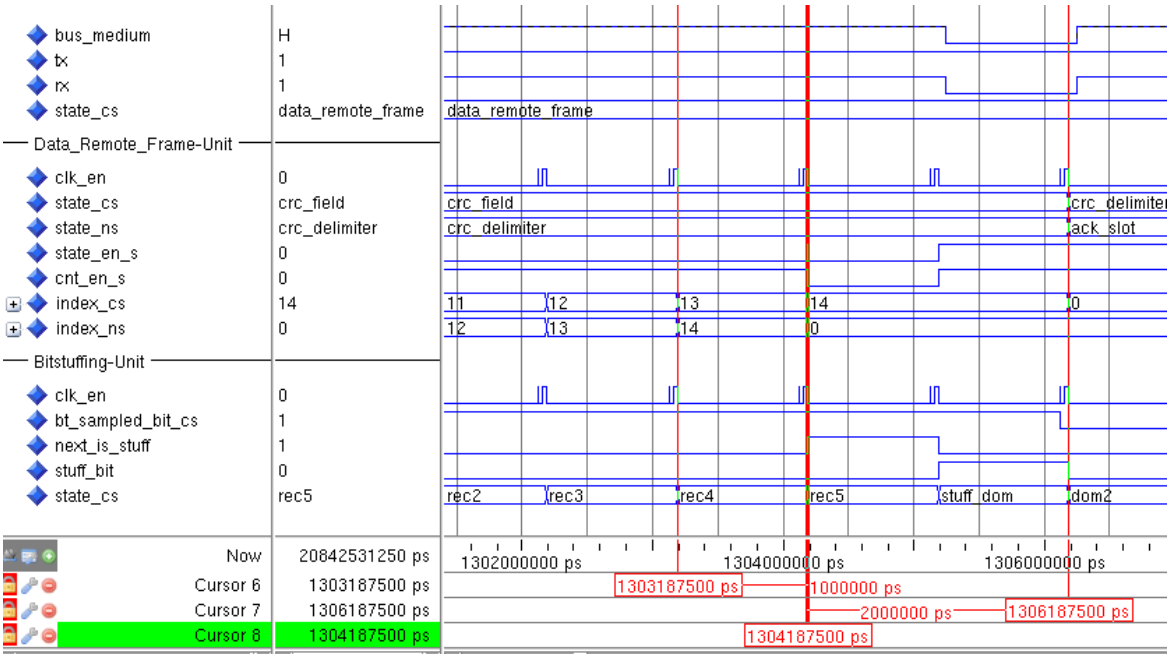


Abbildung 4.3: Simulation des Controllers

Diese Testbench wurde ebenfalls für die Timingsimulation des Controllers verwendet. Abbildung 4.4 zeigt wieder die Situation des Anhaltens der Data-Remote-frame-Unit im Falle eines Stuffbits. Hierbei fällt auf, dass weit weniger Signale vorhanden sind, da durch die

Synthese sehr viele Signale entweder weg optimiert wurden oder durch Signale ersetzt wurden die einen anderen Namen tragen. Aber auch hier ist noch zu erkennen wie der Wert des Indexzählers gesetzt wird.

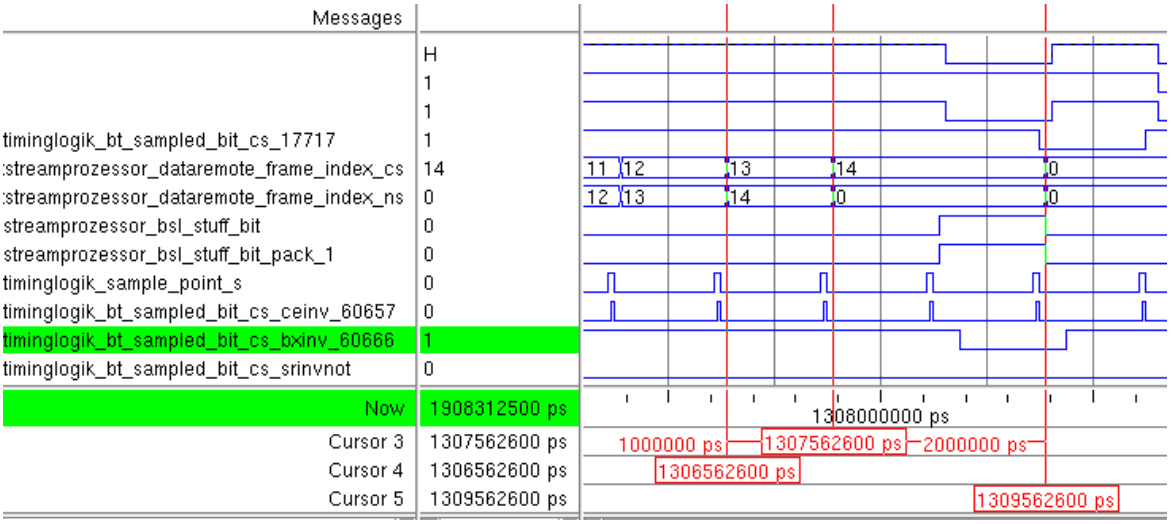


Abbildung 4.4: Timingsimulation des Controllers

Es ist noch zu erwähnen, dass die Tests, die hier durchgeführt wurden, keinesfalls zur der Aussage führen, dass der Controller fehlerfrei ist. Zu einem hinreichenden Test sind deutlich mehr Tests erforderlich. Zu dem bietet Bosch ein VHDL-Referenzmodell zum Testen auf Konformität an. Dieses stand hier allerdings nicht zur Verfügung. Weiterhin wurden das Testen hier, bedingt dadurch, dass dies eben eine Bachelorarbeit ist, vom Entwickler selbst übernommen, was den Regeln guten Testens widerspricht, bei dem Entwicklung und Test eines Systems nicht von der selben Person übernommen werden soll.

4.4 Ergebnis der Synthese

Nachdem auf die Simulation des Controllers eingegangen wurde, soll hier das Ergebnis der Synthese erörtert werden.

Die Synthese ergab eine max. Taktfrequenz von 53,724 MHz für den Controller. Abbildung 4.5 zeigt den entsprechenden Auszug aus dem Synthesereport.

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 18.614ns (Maximum Frequency: 53.724MHz)
   Minimum input arrival time before clock: 5.646ns
   Maximum output required time after clock: 15.584ns
   Maximum combinational path delay: 7.904ns
```

Abbildung 4.5: Auszug aus dem Synthesereport: Timing

Der Verbauch an Hardware im FPGA ist eher moderat. Abbildung 4.6 zeigt den entsprechenden Auszug aus dem Synthesereport dazu. Hierbei ist zu beachten, dass hier für die Synthese der Controller, der Timingsimulation wegen, mit einem Interface versehen wurde, dessen Signale im UCF-File nach Außen weitergegeben wurden. Dies ist im Eintrag "bonded IOBsßu sehen. Dieser hat hier einen Wert von 43. Wenn der Controller später in einem SoC verwendet wird müssen nur die Signale für RX und TX nach Außen geführt werden.

```
Device utilization summary:
---------------------------

Selected Device : 3s1200efg320-4

 Number of Slices:                     1610  out of   8672    18%
 Number of Slice Flip Flops:           1813  out of  17344    10%
 Number of 4 input LUTs:               2289  out of  17344    13%
 Number of IOs:                          43
 Number of bonded IOBs:                  43  out of    250    17%
 Number of GCLKs:                         1  out of     24     4%
```

Abbildung 4.6: Auszug aus dem Synthesereport: Hardwareverbrauch

Bei der Betrachtung der Logikblöcke, die das Synthesetool aus dem Design erstellt hat fällt auf, dass insgesamt 11 Addierer/Subtrahierer erstellt wurden. Siehe Abbildung 4.7. Addierer/Subtrahierer wirken sich aber nachteilig auf die Länge von Schaltnetzen und so auf die max. Taktfrequenz aus.

```
=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 11
 2-bit adder carry out                                 : 1
 4-bit adder                                           : 2
 4-bit adder carry out                                 : 2
 4-bit subtractor                                      : 1
 5-bit adder                                           : 1
 5-bit addsub                                          : 1
 8-bit adder                                           : 1
 8-bit addsub                                          : 1
 9-bit addsub                                          : 1
# Registers                                            : 1920
 Flip-Flops                                            : 1920
# Comparators                                          : 105
 15-bit comparator not equal                           : 1
 31-bit comparator equal                               : 7
 31-bit comparator not equal                           : 7
 32-bit comparator greatequal                          : 1
 32-bit comparator lessequal                           : 1
 4-bit comparator equal                                : 8
 4-bit comparator greatequal                           : 56
 4-bit comparator greater                              : 1
 4-bit comparator not equal                            : 7
 5-bit comparator equal                                : 1
 5-bit comparator greater                              : 1
 8-bit comparator equal                                : 1
 8-bit comparator greatequal                           : 1
 8-bit comparator greater                              : 3
 8-bit comparator less                                 : 2
 9-bit comparator greatequal                           : 1
 9-bit comparator greater                              : 4
 9-bit comparator less                                 : 2
# Multiplexers                                         : 10
 1-bit 16-to-1 multiplexer                             : 2
 1-bit 8-to-1 multiplexer                              : 7
 32-bit 4-to-1 multiplexer                             : 1
# Xors                                                 : 1
 1-bit xor2                                            : 1

=========================================================================
```

Abbildung 4.7: Auszug aus dem Synthesereport: Einzelkomponenten

Eine genauere Betrachtung zeigt, dass von die Addierern/Subtrahierern zwei auf die Faultconfinement-Unit, einer auf die Data-Remote-Frame-Unit, einer auf den Baudratepres-caler und sieben auf die Bittiminglogik entfallen.

Der Synthesereport zeigt bei der Bittiminglogik, dass von diesen sieben Addierern/Subtra-hierern vier für die Berechnung der Distanz des Phasefehlers zum Synchronization-Segment verwendet werden. Dies ist in Abbildung 4.8 zu sehen.

```
Related source file is "/home/simon/ISEWork/can/Modelsim/mit_debug_interface/btl.vhd".
Found finite state machine <FSM_0> for signal <bt_state_cs>.
-----------------------------------------------------------------------
| States             | 4                                              |
| Transitions        | 18                                             |
| Inputs             | 4                                              |
| Outputs            | 11                                             |
| Clock              | clk (rising_edge)                              |
| Clock enable       | bt_state_cs$not0000 (positive)                 |
| Reset              | nreset (negative)                              |
| Reset type         | synchronous                                    |
| Reset State        | syn_seg                                        |
| Power Up State     | syn_seg                                        |
| Encoding           | automatic                                      |
| Implementation     | LUT                                            |
-----------------------------------------------------------------------
Found 4-bit register for signal <bt_phase1_cs>.
Found 3-bit register for signal <bt_phase2_cs>.
Found 1-bit register for signal <bt_resync_cs>.
Found 1-bit register for signal <bt_sampled_bit_cs>.
Found 4-bit register for signal <bt_tq_count_cs>.
Found 4-bit adder for signal <bt_tq_count_ns$addsub0000> created at line 323.
Found 4-bit comparator equal for signal <bt_tq_count_ns$cmp_eq0000> created at line 317.
Found 1-bit register for signal <bus_value_cs>.
Found 1-bit register for signal <clk_en_cs>.
Found 5-bit addsub for signal <distance_v$addsub0001>.
Found 2-bit adder carry out for signal <distance_v$addsub0003> created at line 210.
Found 4-bit adder carry out for signal <distance_v$addsub0004> created at line 244.
Found 4-bit adder carry out for signal <distance_v$addsub0005> created at line 251.
Found 5-bit comparator greater for signal <distance_v$cmp_gt0000> created at line 284.
Found 1-bit register for signal <enable_cs>.
Found 3-bit register for signal <multi_bus_value_cs>.
Found 4-bit adder for signal <new_phase1_v$addsub0000> created at line 296.
Found 4-bit subtractor for signal <new_phase2_v$addsub0000> created at line 292.
Found 1-bit register for signal <tx_bit_cs>.
Summary:
    inferred   1 Finite State Machine(s).
    inferred  20 D-type flip-flop(s).
    inferred   7 Adder/Subtractor(s).
    inferred   2 Comparator(s).
```

Abbildung 4.8: Auszug aus dem Synthesereport: Hardwareverbrauch der Bittiminglogik

4.5 Hardwaretest

Für den Test in Hardware wurde nicht der gesamte CAN-Controller verwendet sondern nur der CAN-CORE. Der Rest des Controllers, also das Mikroprozessor-Interface und die dazugehörigen Register, sowie die Transmission-Gluelogic und die Acceptance-Filter wurden nicht in Hardware getestet. Das Mikroprozessor-Interface und die darin enthaltenen Register wurden mit dem, von Herrn Pautz entwickelten, Mikroprozessor-Interface-Baukasten erzeugt. Herr Pautz hat im Rahmen der Entwicklung dieses Baukastens verschiedene Umsetzungen, die mit diesem Baukastens erzeugt wurden, in Hardware getestet. Zudem steht die

CPU für das SoC noch nicht zu Verfügung. Ein Test in Hardware wäre so nur möglich gewesen, wenn dafür entweder einen andere CPU mit im FPGA verwendet wird oder indem ein Mikrocontroller extern an das FPGA-Board angeschlossen wird und dieser als CPU verwendet wird. Eine andere CPU mit passendem Interface steht allerdings nicht zur Verfügung und im Falle, dass ein Mikrocontroller als CPU verwendet wird, wären Änderungen am Interface nötig. Auf einen Test der konkreten Umsetzung des Interfaces des Controllers in Hardware wurde hier deshalb verzichtet.

Zum Testen des CAN-CORE wurde das NEXYS2 FPGA-Board der Firma Digilent(Nexys2) verwendet. Der verwendete FPGA auf diesem ist ein Spartan-3E mit 1200K Gates von Xilinx. Für die Busankopplung wurde der CAN-Tranceiverbaustein SN65HVD231 von Texas Instruments(SN65HVD231) verwendet.

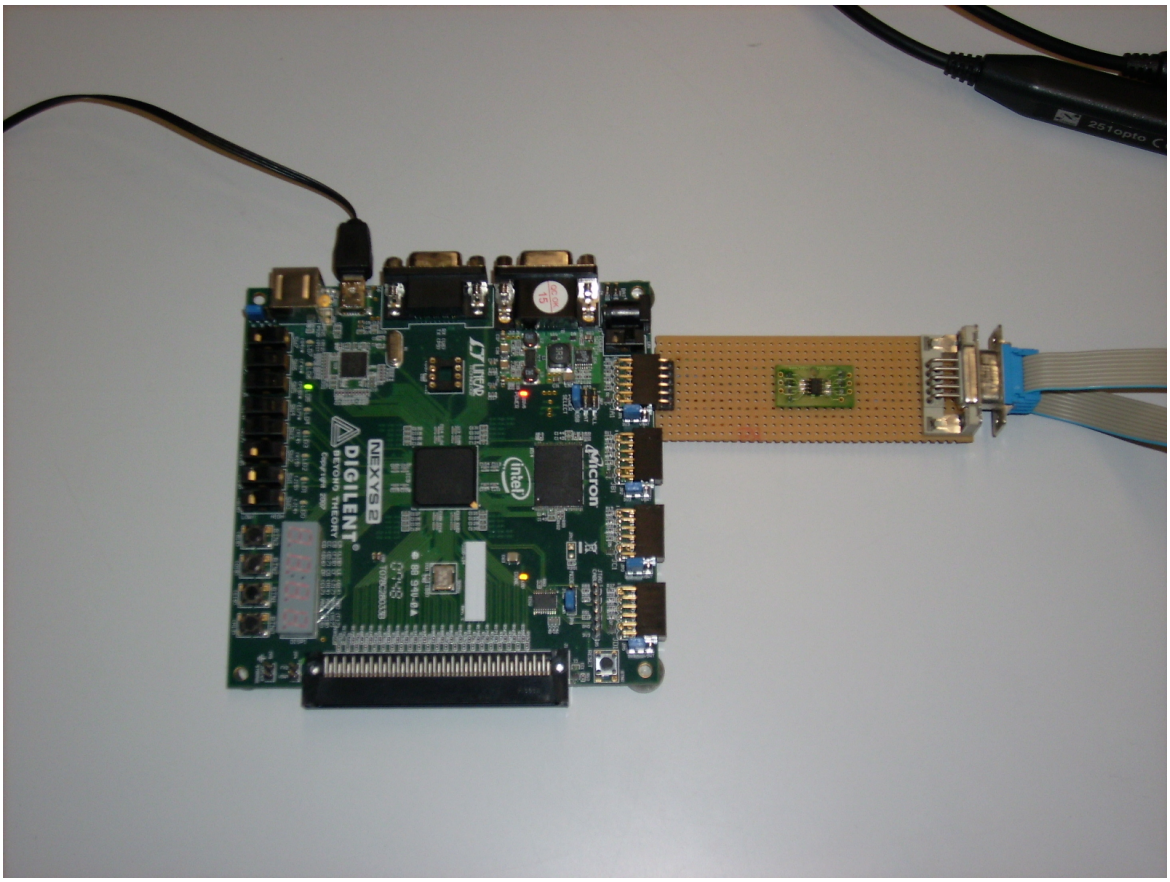


Abbildung 4.9: Digilent Nexys 2 mit angeschlossenem CAN-Tranceiver

Als Gegenstelle zum CAN-CORE wurde ein Laborboard aus der HAW verwendet. Dieses nutzt einen AT90CAN128 Mikrocontroller von Atmel und wird auch in Praktika an der HAW eingesetzt und verfügt bereits über einen CAN-Tranceiver. Weiterhin ist dieses Board mit

einer seriellen Schnittstelle und einem passenden Tranceiver ausgestattet wodurch eine einfache Terminalverbindung mit diesem Board möglich ist. Die Konfiguration dieses Boards wurde nicht verändert. Abbildung 4.10 zeigt das verwendete Laborboard.

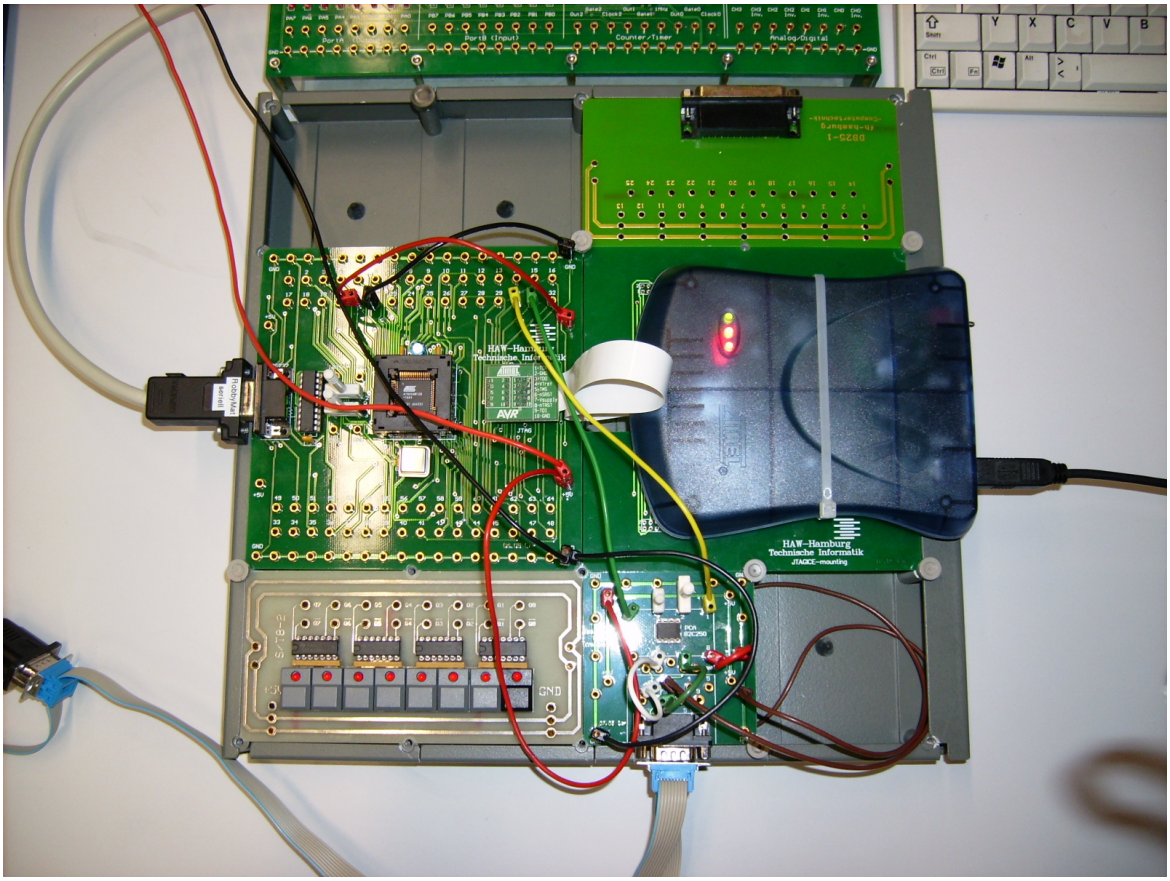


Abbildung 4.10: Laborboard mit Atmel AT90CAN128

Zur Überprüfung des Busverkehrs sowie um weitere Teilnehmer im CAN-Bus zu simulieren wurde ein Laptop mit einem CANalyzer der Firma Vector(CANalyser) verwendet. Mit diesem ist es möglich den Datenverkehr auf dem Bus zu analysieren und gezielt Daten zu versenden. Dieser Laptop ist in Abbildung 4.11 zu sehen.

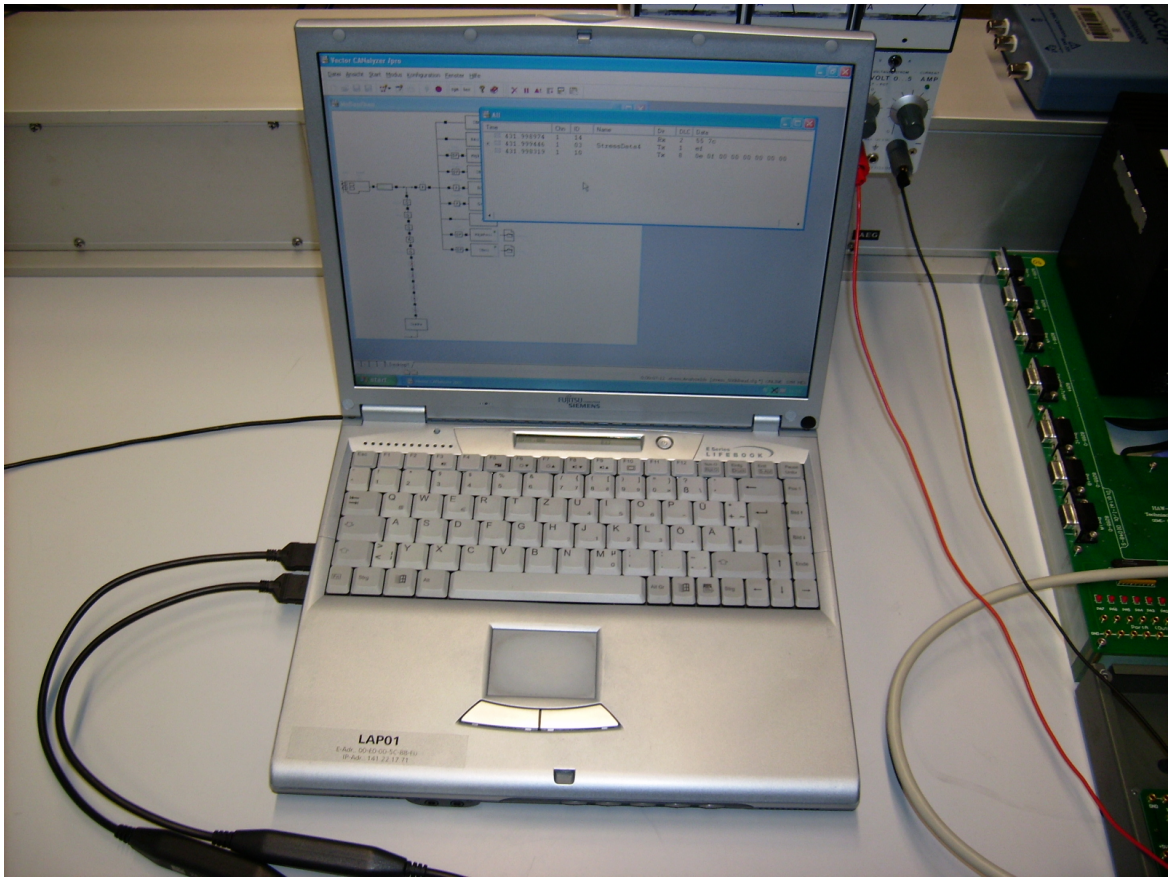


Abbildung 4.11: Canalyser

Der komplette Versuchsaufbau ist noch einmal in Abbildung 4.12 dargestellt.

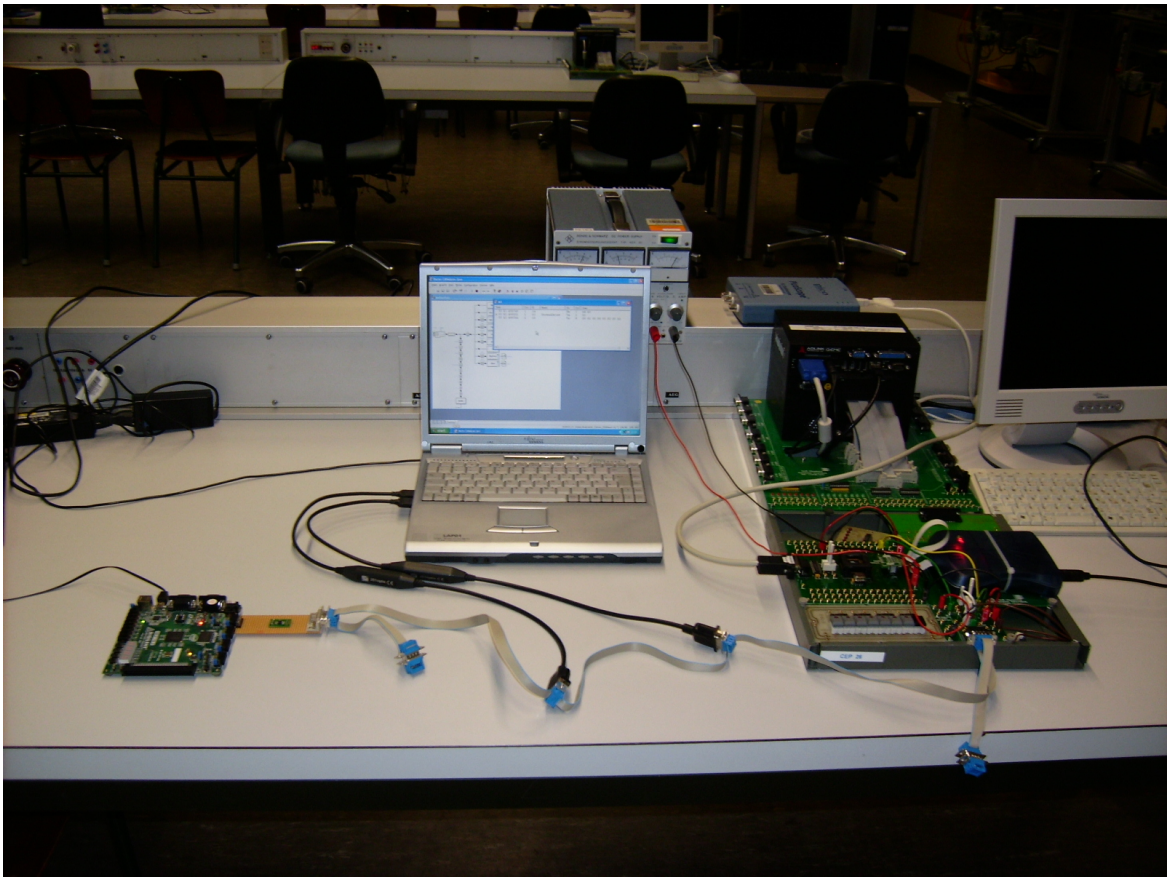


Abbildung 4.12: Versuchsaufbau

Der CAN-CORE wurde zum Testen mit drei verschiedenen Betriebsmodi und vier einstellbaren Bittimings ausgestattet:

- **Modus 1**: Definierte und unveränderliche Daten. Dieser Modus dient dazu um mit einem Oszilloskop o.ä. festzustellen, ob das gesendete Frame der Definition entspricht.
- **Modus 2**: Es wird eine Sequenz von 16-Bit Pseudozufallszahlen gesendet. Dieser Modus dient dazu, hinreichend unterschiedliche Daten zum Versand zu erzeugen, welche von einer Gegenstelle auf Korrektheit überprüft werden können. Dazu muss an einer Gegenstelle jeweils die Pseudozufallszahl errechnet werden, von der erwartet wird, das der CAN-CORE diese sendet.
- **Modus 3**: Es wird eine Sequenz von 7-Bit Pseudozufallszahlen gesendet. Dieser Modus dient dazu, einen überprüfbaren Datenstrom zu erzeugen ohne dass die Gegenstelle die Pseudozufallszahl berechnen muss.

Der CAN-CORE wurde mit einen Takt von 50 MHz, was dem Takt des auf dem FPGA-Board verbauten Oszillators entspricht, getestet. Alle Tests wurden bei einer Bitrate von

125 kbit durchgeführt. Weiterhin wurde der CAN-CORE mit einem festen, also nicht einstellbaren Identifier für die Nachrichten versehen. Für die Tests mit Zufallszahlen wurde der AT90CAN128 als Gegenstelle für den CAN-CORE genutzt, während der CANalyser dazu genutzt wurde die Kommunikation anderer Busteilnehmer zu simulieren. Für die Erzeugung der Pseudozufallszahlensequenzen(Pseudo Random Bit Sequence - **PRBS**) wurden im FPGA zwei Linear-Feedback-Shift-Register(LFSR) verwendet. Zur Erzeugung der Pseudozufallszahlensequenzen wurden folgende Feedbackpolynome verwendet:

$$PRBS7 : x^7 + x^6 \tag{4.1}$$

Abbildung 4.13: Feedbackpolynom für 7 Bit Pseudozufallszahlensequenz

$$PRBS16 : x^{16} + x^{14} + x^{13} + x^{11} \tag{4.2}$$

Abbildung 4.14: Feedbackpolynom für 16 Bit Pseudozufallszahlensequenz

Diese Polynome erzeugen jeweils ein Pseudozufallszahlensequenz maximaler Länge. Also entsprechend Bitbreite eine Anzahl von $2^{Bitbreite} - 1$ Pseudozufallszahlen. Diese Polynome wurden aus Table of Linear Feedback Shift Register vom Department of Physics der Universität von Otago in Neuseeland(LFSR-Table) entnommen.

Vor jedem dieser Tests führte der AT90CAN128 ein Capture-Phase durch. In dieser Phase empfing dieser 1000 Nachrichten und übernahm die letzte dieser Nachrichten bevor der eigentliche Test durchgeführt wurde. Durch diese Phase wurde dafür gesorgt, dass der Empfangspuffer des AVR keine alten Daten mehr enthielt.

4.5.1 7-Bit Pseudozufallszahlensequenz

Beim Test mit einer 7 Bit Pseudozufallszahlensequenz ergibt sich, wenn als Startwert für das LFSR ein Wert ungleich Null gewählt wurde, ein Zyklus von den Zahlenwerten 1 bis 127. In diesem Test wird überprüft, ob diese Zyklen über einen längeren Zeitraum eingehalten werden. Dazu wird der AVR genutzt um immer eine Zufallszahl vom Bus entgegen zu nehmen und diese als Index für einen Array zu verwenden und diesen an dieser Stelle um 1 zu inkrementieren. In jedem Zyklus darf ein Zahlenwert nur einmal vorkommen anderen Falls wird dies als Fehler vermerkt. Ebenfalls wird als Fehler vermerkt, wenn eine Null empfangen wurde. Dieser Test endet nachdem 128 Zyklen empfangen wurden. Danach kann einfach überprüft werden, ob Fehler aufgetreten sind, indem dieser Array durchlaufen wird und für jeden Indexwert überprüft wird, ob der Array an dieser Stelle einen Wert ungleich 128 hat. Die Anzahl von 128 Zyklen wurde hierbei willkürlich gewählt.

Es konnten keine Fehler bei diesem Test festgestellt werden.

4.5.2 16-Bit Pseudozufallszahlensequenz

Bei diesem Test berechnet der AVR analog zum CAN-CORE eine 16-Bit Zufallszahlsequenz. Als Startwert nimmt er dafür die zuletzt gesendete Zufallszahl. Danach nimmt der AVR immer eine Zufallszahl vom CAN-CORE entgegen und vergleicht, ob diese mit der übereinstimmt die dieser selbst errechnet hat. Ein Abweichen wird als Fehler gewertet. Ein Übereinstimmen wird als erfolgreich vermerkt. Der AVR gibt dabei immer nach 10000 empfangenen Nachrichten aus, wie viele davon erfolgreich und wie viele fehlerhaft waren. Abbildung 4.15 zeigt einen Ausschnitt aus diesem Printout.

Dieser Test wurde über einen Zeitraum von ca. einer Stunde ausgeführt. Dabei wurden keinen fehlerhaften Nachrichten entdeckt. Während dieses Tests wurde mit dem CANalyser weitere Kommunikation auf dem Bus simuliert.

Insgesamt wurden während dieses Zeitraumes 500 Printouts aufgezeichnet. Diese waren alle fehlerfrei. Aus diesen 500 Printouts und den entsprechenden 10000 Nachrichten für die jeder einzelne steht sowie, pro Nachricht 16 Bit, also 2 Byte, versandt wurden ergibt sich ein versandte Datenmenge von 9,536 MByte.

```
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
OK:  10000,   Err: 0
```

Abbildung 4.15: Printout des AVR

5 Persönliches Fazit

Hier soll nun kurz erläutert werden, was mir, im Rückblick betrachtet, an diesem Projekt positiv und was eher negativ auffällt.

In der Auswertung des Syntheseergebnisses hat sich gezeigt, dass die Bittiminglogik sehr viele Addierer/Subtrahierer verbraucht. Dies ist mit relativ wenig Aufwand optimierbar und wird sich auch positiv auf das Syntheseergebnis, sowohl in Hinblick auf die max. Taktfrequenz als auch auf den Hardwareverbrauch, auswirken.

Ebenso sollte überdacht werden, ob die Verwendung eines Counters in der Data-Remoteframe-Unit für den Zustandswechsel sinnvoll ist oder dieser Counter besser aus dieser Unit entfernt wird und stattdessen für jedes Bit ein eigener Zustand entworfen wird.

Ebenso missfällt mir inzwischen die Idee, das Protokoll in viele kleine Units zerlegt zu haben. Dies habe ich getan, um den Code möglichst übersichtlich zu halten. Dieses rächt sich aber in der Synthese. Jede dieser Units ist als ein Zustandsautomat mit eigenem Zustandsregister implementiert. Wären diese Unit zu einem Automaten zusammengefasst gäbe es nur ein Zustandsregister dafür. In der Synthese wird so eine mögliche Optimierung eingeschränkt.

Auch gefällt mir das Mikroprozessor-Interface so nicht mehr. Dies sollte vor allem sehr simpel gehalten werden um so keine Hardware zu verschwenden. Allerdings bin ich mir nicht mehr sicher, ob dies eine gute Entscheidung war und das Interface nicht doch um Funktionen wie Auto-Reply erweitert werden sollte oder ganz ersetzt werden sollte.

Ebenso hätte ich noch gerne eine Unterstützung für TTCAN eingebaut. Allerdings war dies auf Grund fehlender Informationen und zu knapper Zeit, sowohl zum Implementieren als auch im Besonderen zum Testen des selbigen, nicht möglich.

6 Aussichten

Der CAN-Controller ist einsatzbereit. Jedoch ist dieser nicht auf Konformanz zum ISO-Standard 11898-1 getestet, da dieser nicht zur Verfügung stand. Der Controller bietet durch seinen modularen Aufbau die Möglichkeit diesen leicht anzupassen. Hier böte sich vor allem an, diesen um eine Unterstützung für TTCAN zu erweitern, da dies eine sehr beliebte und mächtige Erweiterung ist. Auch sollte noch am Design gearbeitet werden, um dieses schlanker und effizienter zu machen. Ebenso wurden noch keine Bibliotheken für die Verwendung dieses Controllers in Sprachen wie z.B. C entwickelt.

Literaturverzeichnis

[CAN 2.0] ROBERT BOSCH GMBH (Hrsg.): *CAN Specification Version 2.0*. – Stand: 1991-09 http://www.semiconductors.bosch.de/pdf/can2spec.pdf - Abruf: 2009-09-17

[CAN Bit Timings] PORT GMBH (Hrsg.): *Request form for CANopen Bit Timings*. – http://www.port.de/pages/misc/bittimings.php?lang=en - Abruf: 2009-12-08

[CANalyser] VECTOR INFORMATIK GMBH (Hrsg.): *Vector [CANalyzer]*. – http://www.vector.com/vi_canalyzer_de,,2816.html - Abruf: 2009-12-08

[CAST] CAST INC. (Hrsg.): *CAN IP Core - CAN CAN-specification 2.0B Bus Controller from CAST, Inc.*. – http://www.cast-inc.com/cores/can/index.shtml - Abruf: 2009-10-01

[CiA 2008] CAN IN AUTOMATION (Hrsg.): *Manufacturing*. – http://www.can-cia.org/index.php?id=131 - Abruf: 2008-12-08

[DCAN] ROBERT BOSCH GMBH (Hrsg.): *D CAN for FPGA*. – http://www.semiconductors.bosch.de/en/20/can/products/dcan-netlist.asp - Abruf: 2008-12-08

[Engels 2002] ENGELS, Horst ; ENGELS, Horst (Hrsg.): *CAN-Bus-Technik einfach, anschaulich und praxisnah vorgestellt. 2. überarbeitete Aufl.* Poing : Franzis, 2002. – ISBN 3-7723-5146-8

[Etschberger 2000] ETSCHBERGER, Konrad: Vorwort. In: ETSCHBERGER, Konrad (Hrsg.): *Controller-Area-Network - Grundlagen, Protokolle, Bausteine, Anwendungen*. München : Hanser, 2000. – ISBN 3-446-19431-2 S. V–VII

[Farsi 1999] FARSI, M. ; RATCLIFF, K. ; BARBOSA, Manuel: An overview of Controller Area Network. In: *COMPUTING & CONTROL ENGINEERING JOURNAL* 10 (1999), june, Nr. 3, S. 113 – 120

[Höltkemeier 2008] HÖLTKEMEIER, Karl-Ullrich: BMW: Internet Protocol fürs Auto. In: *Konstruktionspraxis.de*. – Stand: 2008-03-17 http://www.konstruktionspraxis.vogel.de/themen/automatisierung/hmi/bedienen/articles/117037/ - Abruf: 2008-12-08

[Leen 2002] LEEN, Gabriel ; HEFFERNAN, Donal: Expanding Automotive Electronic Systems. In: *IEEE Computer* 35 (2002), Januar, Nr. 1, S. 88–93

[LFSR-Table] DEPARTMENT OF PHYSICS, UNIVERSITY OF OTAGO, NEW ZEALAND (Hrsg.): *Table of Linear Feedback Shift Register Taps*. – http://www.physics.otago.ac.nz/px/research/electronics/papers/technical-reports/lfsr_table.pdf/view - Abruf: 2009-11-29

[Nexys2] DIGILENT, INC. (Hrsg.): *Nexys2 FPGA Board*. – http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,789&Prod=NEXYS2 - Abruf: 2009-12-10

[OpenCores] OPENCORES.ORG (Hrsg.): *www.opencores.org*. – http://www.opencores.org/ - Abruf: 2008-10-17

[Paret 2007] PARET, Dominique: *Multiplexed Networks for Embedded Systems*. WILEY, 2007. – ISBN 978-0-470-03416-3

[SN65HVD231] TEXAS INSTRUMENTS INCORPORATED (Hrsg.): *Interface - CAN - SN65HVD231 - TI.com*. – http://focus.ti.com/docs/prod/folders/print/sn65hvd231.html - Abruf: 2009-08-29

[Xilinx] XILINX, INC. (Hrsg.): *CAN*. – http://www.xilinx.com/products/ipcenter/DO-DI-CAN.htm - Abruf: 2009-05-20

A Interface des Controllers

Hier soll eine kurze Erklärung zum Interface des Controllers und die Belegung dessen Register gegeben werden.

A.1 Anordnung der Register

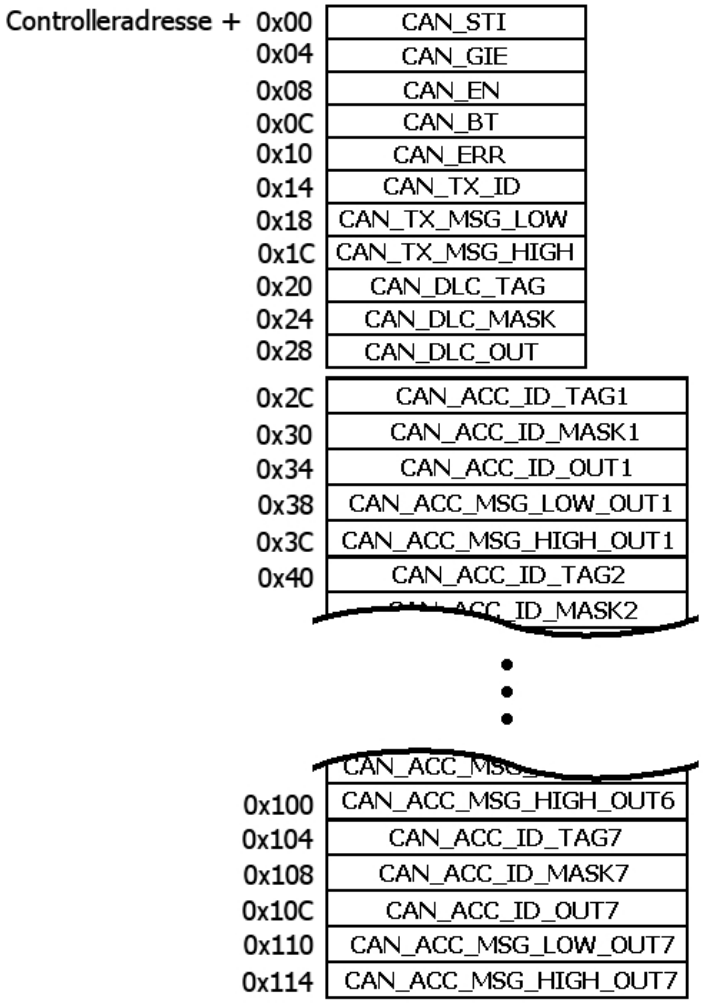


Abbildung A.1: Anordnung der Register

A.2 CAN_STI: CAN Status Interrupt

Reset-On-Read Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
--	--	--	--	--	--	--	--
--	--	--	--	--	DLCWARN	ARB-LOST	BITERR
FORMERR	ACKERR	CRCERR	STUFFERR	SOF	EOF	BUS-OFF	ERRWARN
TXOK	ACC7_RXOK	ACC6_RXOK	ACC5_RXOK	ACC4_RXOK	ACC3_RXOK	ACC2_RXOK	ACC1_RXOK
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.2: CAN Status Interrupt Register

In diesem Register wird ausgegeben welche Interrupts aufgetreten sind. Eine 1 bedeutet dabei, dass ein entsprechender Interrupt aufgetreten ist. Dieses Register wird beim Lesen automatisch resettet und der Interrupt zurückgenommen.

DLCWARN(Datalengthcode Warning) wird ausgelöst, wenn der empfangene DLC grösser als 8 Byte ist.

ARB-LOST(Arbitration Lost) wird ausgelöst, wenn der Controller die Arbitrierung verliert.

BITERR(Bit Error) wird ausgelöst, wenn ein Bitfehler aufgetreten ist.

FORMERR(Form Error) wird bei aufgetretenem Formfehler ausgelöst.

ACKERR(Acknowledgement Error) wird ausgelöst, wenn ein Acknowledgementfehler aufgetreten ist.

CRCERR(CRC Error) wird ausgelöst, wenn ein CRC-fehler aufgetreten ist.

STUFFERR(Stuffing Error) wird ausgelöst, wenn ein Fehler beim Bitstuffing aufgetreten ist.

SOF(Start Of Frame) wird ausgelöst, wenn ein Start of Frame Bit erkannt wurde.

EOF(End Of Frame) wird bei erkanntem End of Frame ausgelöst.

BUS-OFF wird ausgelöst, wenn der Controller in den Bus Off Zustand übergeht.

ERRWARN(Error Warning) wird ausgelöst wenn einer der Fehlerzähler einen Wert größer oder gleich 96 annimmt.

TXOK(Transmission Okay) wird ausgelöst, wenn der Controller erfolgreich eine Nachricht übertragen hat.

ACC1_RXOK - ACC7_RXOK(Acceptance-Filter 1-7 Reception Okay) wird ausgelöst, wenn einer der Acceptance-Filter eine Nachricht akzeptiert hat.

A.3 CAN_GIE: CAN General Interrupt Enable

Read and Write Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
--	--	--	--	--	--	--	--
--	--	--	--	--	DLCWARN	ARB-LOST	BITERR
FORMERR	ACKERR	CRCERR	STUFFERR	SOF	EOF	BUS-OFF	ERRWARN
TXOK	ACC7_RXOK	ACC6_RXOK	ACC5_RXOK	ACC4_RXOK	ACC3_RXOK	ACC2_RXOK	ACC1_RXOK
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.3: CAN General Interrupt Enable Register

In diesem Register können die Enables für die Interrupts gesetzt werden. Die Interrupts in CAN_STI werden nur gesetzt wenn in diesem Register die entsprechenden Enables gesetzt wurden.

- 0 := Interrupt disabled
- 1 := Interrupt enabled

A.4 CAN_EN: CAN Enable

Read and Write Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
Enable	SingleShot	Bus-Off-Rtrn	--	--	--	--	--
--	--	--	--	--	--	--	--
--	ACC7_OV	ACC6_OV	ACC5_OV	ACC4_OV	ACC3_OV	ACC2_OV	ACC1_OV
TX-Req	ACC7_EN	ACC6_EN	ACC5_EN	ACC4_EN	ACC3_EN	ACC2_EN	ACC1_EN
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.4: CAN Enable Register

Enable: generelles Enable für den Controller

SingleShot: Wenn dieses Bit gesetzt ist, wird der Controller die zu sendende Nachricht verwerfen, sobald dieser entweder die Arbitrierung verloren hat oder die Nachricht während der Übertragung zerstört wurde. Der Controller wird nicht versuchen diese Nachricht erneut zu senden.

Bus-Off-Rtrn(Bus-Off-Return): Wenn dieses Bit gesetzt ist, wird der Controller nach 128 * 11 rezessiven Bit aus dem Bus-Off-Zustand in den Error-Active-Zustand zurückkehren.

ACC1_OV - ACC7_OV(Acceptance-Filter Overloadrequest): Wenn einer der Acceptance-Filter eine Nachricht akzeptiert hat, wird der Controller im Anschluss ein Overloadframe senden, wenn das entsprechende Bit gesetzt ist.

TX_Req(Transmit Request): Wenn dieses Bit gesetzt ist, wird der Controller den Inhalt aus den Transmitregistern senden. Bei erfolgreicher Übertragung wird dieses Bit automatisch zurückgesetzt. Dies gilt auch, wenn SingleShot aktiviert wurde und die Nachricht zerstört wurde oder bei der Arbitrierung unterlegen ist.

ACC1_EN - ACC7_EN(Acceptance-Filter Enable): Enable für die Acceptance-Filtering-Einheiten des Controllers.

A.5 CAN_BT: CAN Bittiming

Read and Write Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
BRP Rate 7	BRP Rate 6	BRP Rate 5	BRP Rate 4	BRP Rate 3	BRP Rate 2	BRP Rate 1	BRP Rate 0
--	--	--	--	--	--	--	--
--	PROP 3	PROP 1	PROP 0	--	PHASE1 2	PHASE1 1	PHASE1 0
--	PHASE2 2	PHASE2 1	PHASE2 0	--	MultiSample	SJW 1	SJW 0
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.5: CAN Bittiming Register

BRP Rate 7 - BRP Rate 0(Baudrateprescaler Rate): Wert für den Baudrateprescaler.

PROP 3 - PROP 0: Wert für das Propgationdelay Segment.

PHASE1 2 - PHASE1 0: Wert für das Phase1-Buffersegment.

PHASE2 2 - PHASE2 0: Wert für das Phase2-Buffersegment.

Multisample: Wenn dieses Bit gesetzt ist, wird der Buspegel 3 mal gesampled und aus einer Mehrheitsentscheidung darauf bestimmt. Der so errechnete Wert wird am Samplepoint übernommen.

- 0 := Singlesampling. Der Buspegel wird am Samplepoint direkt übernommen.
- 1 := Multisampling. Der Buspegel wird aus der Mehrheit von 3 gesampleten Wert errechnet.

SJW 1 - SJW 0(Synchronization Jump Width): Wert der Sprungweite für das Synchronisieren.

Die Werte für BRP Rate, PROP, PHASE1, PHASE2 und SJW sind alle als max. Index für diese Werte zu verstehen und müssen daher um 1 vermindert angegeben werden. Ein Wert von 7 für die BRP Rate bedeutet also, dass der Prescaler den Takt durch 8 teilt. Die Bittiminglogik dieses Controllers ist kompatibel zu den Timings eines Atmel CANary. Diese Timings können auf der Website der port GmbH(CAN Bit Timings) entnommen werden.

A.6 CAN_ERR: CAN Error

Read-Only Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
--	--	--	--	--	--	--	--
--	--	--	--	--	--	--	TEC8
TEC 7	TEC 6	TEC 5	TEC 4	TEC 3	TEC 2	TEC 1	TEC 0
REC 7	REC 6	REC 5	REC 4	REC 3	REC 2	REC 1	REC 0
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.6: CAN Error Register

In diesem Register können die Werte der Fehlerzähler ausgelesen werden.

A.7 CAN_TX_ID: CAN Transmit Identifier

Read and Write Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
--	BaseID 10	BaseID 9	BaseID 8	BaseID 7	BaseID 6	BaseID 5	BaseID 4
BaseID 3	BaseID 2	BaseID 1	BaseID 0	ExtID 17	ExtID 16	ExtID 15	ExtID 14
ExtID 13	ExtID 12	ExtID 11	ExtID 10	ExtID 9	ExtID 8	ExtID 7	ExtID 6
ExtID 5	ExtID 4	ExtID 3	ExtID 2	ExtID 1	ExtID 0	IDE	RTR
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.7: CAN Transmit Identifier Register

In diesem Register wird der Identifier der zu übertragenden Nachricht angegeben.

BaseID 10 - 0: Base-Identifer der Nachricht.

ExtID 17 - 0: Extended-Identifier der Nachricht.

IDE(Identifier Extension):

- 0 := Standardformat
- 1 := Extendedformat

RTR(Remote Transmission Request):

- 0 := Dataframe
- 1 := Remoteframe

A.8 CAN_TX_MSG_LOW: CAN Transmit Message Low

Read and Write Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
Byte3 - 7	Byte3 - 6	Byte3 - 5	Byte3 - 4	Byte3 - 3	Byte3 - 2	Byte3 - 1	Byte3 - 0
Byte2 - 7	Byte2 - 6	Byte2 - 5	Byte2 - 4	Byte2 - 3	Byte2 - 2	Byte2 - 1	Byte2 - 0
Byte1 - 7	Byte1 - 6	Byte1 - 5	Byte1 - 4	Byte1 - 3	Byte1 - 2	Byte1 - 1	Byte1 - 0
Byte0 - 7	Byte0 - 6	Byte0 - 5	Byte0 - 4	Byte0 - 3	Byte0 - 2	Byte0 - 1	Byte0 - 0
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.8: CAN Transmit Message Low Register

In diesem Register werden Byte0 - Byte3 der zu sendenden Nachricht angegeben.

A.9 CAN_TX_MSG_HIGH: CAN Transmit Message High

Read and Write Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
Byte7 - 7	Byte7 - 6	Byte7 - 5	Byte7 - 4	Byte7 - 3	Byte7 - 2	Byte7 - 1	Byte7 - 0
Byte6 - 7	Byte6 - 6	Byte6 - 5	Byte6 - 4	Byte6 - 3	Byte6 - 2	Byte6 - 1	Byte6 - 0
Byte5 - 7	Byte5 - 6	Byte5 - 5	Byte5 - 4	Byte5 - 3	Byte5 - 2	Byte5 - 1	Byte5 - 0
Byte4 - 7	Byte4 - 6	Byte4 - 5	Byte4 - 4	Byte4 - 3	Byte4 - 2	Byte4 - 1	Byte4 - 0
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.9: CAN Transmit Message High Register

In diesem Register werden Byte4 - Byte7 der zu sendenden Nachricht angegeben.

A.10 CAN_DLC_TAG: CAN Datalengthcode and Tag

Read and Write Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
TX_DLC 3	TX_DLC 2	TX_DLC 1	TX_DLC 0	Acc7DLCT 3	Acc7DLCT 2	Acc7DLCT 1	Acc7DLCT 0
Acc6DLCT 3	Acc6DLCT 2	Acc6DLCT 1	Acc6DLCT 0	Acc5DLCT 3	Acc5DLCT 2	Acc5DLCT 1	Acc5DLCT 0
Acc4DLCT 3	Acc4DLCT 2	Acc4DLCT 1	Acc4DLCT 0	Acc3DLCT 3	Acc3DLCT 2	Acc3DLCT 1	Acc3DLCT 0
Acc2DLCT 3	Acc2DLCT 2	Acc2DLCT 1	Acc2DLCT 0	Acc1DLCT 3	Acc1DLCT 2	Acc1DLCT 1	Acc1DLCT 0
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.10: CAN Datalengthcode and Tag Register

In diesem Register kann zum einen der Datalengthcode der zu sendenden Nachricht angegeben werden und zum anderen die Datalengthcode-Tags für die Acceptance-Filter.

TX_DLC 3 - 0(Transmit Datalengthcode): Datalengthcode der zu übertragenden Nachricht. Gültig hierfür sind die Werte 0 - 15. Dabei ist zu beachten, dass aber max. 8 Byte übertragen werden können. Der Datalengthcode wird allerdings so übertragen wie er in diesem Register angegeben ist.

Acc*DLCT 3 - 0(Acceptance-Filter Datalengthcode Tag): Wert für den Datalengthcodetag des entsprechenden Acceptance-Filters.

A.11 CAN_DLC_MASK: CAN Datalengthcode Mask

Read and Write Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
--	--	--	--	Acc7DLCM 3	Acc7DLCM 2	Acc7DLCM 1	Acc7DLCM 0
Acc6DLCM 3	Acc6DLCM 2	Acc6DLCM 1	Acc6DLCM 0	Acc5DLCM 3	Acc5DLCM 2	Acc5DLCM 1	Acc5DLCM 0
Acc4DLCM 3	Acc4DLCM 2	Acc4DLCM 1	Acc4DLCM 0	Acc3DLCM 3	Acc3DLCM 2	Acc3DLCM 1	Acc3DLCM 0
Acc2DLCM 3	Acc2DLCM 2	Acc2DLCM 1	Acc2DLCM 0	Acc1DLCM 3	Acc1DLCM 2	Acc1DLCM 1	Acc1DLCM 0
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.11: CAN Datalengthcode Mask Register

In diesem Register werden die Masken für den Datalengthcode für die entprechenden Acceptance-Filter angegeben. Durch diese Masken ist ein Filtern auf Bereiche des Datalengthcode möglich.

- 0 := empfangenes Bit an dieser Stelle wird beim Akzeptanzfiltern ignoriert.
- 1 := empfangenes Bit an dieser Stelle muss beim Akzeptanzfiltern mit dem DLC-TAG an dieser Stelle übereinstimmen.

A.12 CAN_DLC_OUT: CAN Datalengthcode Output

Read-Only Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
--	--	--	--	DLC7_OUT 3	DLC7_OUT 2	DLC7_OUT 1	DLC7_OUT 0
DLC6_OUT 3	DLC6_OUT 2	DLC6_OUT 1	DLC6_OUT 0	DLC5_OUT 3	DLC5_OUT 2	DLC5_OUT 1	DLC5_OUT 0
DLC4_OUT 3	DLC4_OUT 2	DLC4_OUT 1	DLC4_OUT 0	DLC3_OUT 3	DLC3_OUT 2	DLC3_OUT 1	DLC3_OUT 0
DLC2_OUT 3	DLC2_OUT 2	DLC2_OUT 1	DLC2_OUT 0	DLC1_OUT 3	DLC1_OUT 2	DLC1_OUT 1	DLC1_OUT 0
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.12: CAN Datalengthcode Output Register

In diesem Register werden die Datalengthcodes der akzeptierten Nachrichten ausgegeben.

A.13 CAN_ACC_ID_TAG*: CAN Acceptance-Filter Identifier Tag *

Read and Write Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
--	BaseID 10	BaseID 9	BaseID 8	BaseID 7	BaseID 6	BaseID 5	BaseID 4
BaseID 3	BaseID 2	BaseID 1	BaseID 0	ExtID 17	ExtID 16	ExtID 15	ExtID 14
ExtID 13	ExtID 12	ExtID 11	ExtID 10	ExtID 9	ExtID 8	ExtID 7	ExtID 6
ExtID 5	ExtID 4	ExtID 3	ExtID 2	ExtID 1	ExtID 0	IDE	RTR
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.13: CAN Acceptance-Filter Identifier Tag Register *

In diesen Registern wird, für den entsprechenden Acceptance-Filter, der Identifier-Tag für des Filtern angegeben.

A.14 CAN_ACC_ID_MASK*: CAN Acceptance-Filter Identifier Mask *

Read and Write Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
--	BaseID 10	BaseID 9	BaseID 8	BaseID 7	BaseID 6	BaseID 5	BaseID 4
BaseID 3	BaseID 2	BaseID 1	BaseID 0	ExtID 17	ExtID 16	ExtID 15	ExtID 14
ExtID 13	ExtID 12	ExtID 11	ExtID 10	ExtID 9	ExtID 8	ExtID 7	ExtID 6
ExtID 5	ExtID 4	ExtID 3	ExtID 2	ExtID 1	ExtID 0	IDE	RTR
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.14: CAN Acceptance-Filter Identifier Mask Register *

In diesen Registern kann angegeben werden welche Bits beim Filtern der Nachricht berücksichtigt werden.

- 0 := empfangenes Bit an dieser Stelle wird beim Filtern nicht berücksichtigt
- 1 := empfangenes Bit an dieser Stelle muss mit dem des Identifier-Tags übereinstimmen.

A.15 CAN_ACC_ID_OUT*: CAN Acceptance-Filter Identifier Output *

Read-Only Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
--	--	--	--	DLC7_OUT 3	DLC7_OUT 2	DLC7_OUT 1	DLC7_OUT 0
DLC6_OUT 3	DLC6_OUT 2	DLC6_OUT 1	DLC6_OUT 0	DLC5_OUT 3	DLC5_OUT 2	DLC5_OUT 1	DLC5_OUT 0
DLC4_OUT 3	DLC4_OUT 2	DLC4_OUT 1	DLC4_OUT 0	DLC3_OUT 3	DLC3_OUT 2	DLC3_OUT 1	DLC3_OUT 0
DLC2_OUT 3	DLC2_OUT 2	DLC2_OUT 1	DLC2_OUT 0	DLC1_OUT 3	DLC1_OUT 2	DLC1_OUT 1	DLC1_OUT 0
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.15: CAN Acceptance-Filter Identifier Output Register *

In diesen Registern wird der Identifier der akzeptierten Nachricht ausgegeben.

A.16 CAN_ACC_MSG_LOW_OUT*: CAN Acceptance-Filter Message Low Output *

Read-Only Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
Byte3 - 7	Byte3 - 6	Byte3 - 5	Byte3 - 4	Byte3 - 3	Byte3 - 2	Byte3 - 1	Byte3 - 0
Byte2 - 7	Byte2 - 6	Byte2 - 5	Byte2 - 4	Byte2 - 3	Byte2 - 2	Byte2 - 1	Byte2 - 0
Byte1 - 7	Byte1 - 6	Byte1 - 5	Byte1 - 4	Byte1 - 3	Byte1 - 2	Byte1 - 1	Byte1 - 0
Byte0 - 7	Byte0 - 6	Byte0 - 5	Byte0 - 4	Byte0 - 3	Byte0 - 2	Byte0 - 1	Byte0 - 0
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.16: CAN Acceptance-Filter Message Low Output Register *

In diesem Register werden die Byte 0-3 der akzeptierten Nachricht ausgegeben.

A.17 CAN_ACC_MSG_HIGH_OUT*: CAN Acceptance-Filter Message HIGH Output *

Read-Only Register

31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16
Byte7 - 7	Byte7 - 6	Byte7 - 5	Byte7 - 4	Byte7 - 3	Byte7 - 2	Byte7 - 1	Byte7 - 0
Byte6 - 7	Byte6 - 6	Byte6 - 5	Byte6 - 4	Byte6 - 3	Byte6 - 2	Byte6 - 1	Byte6 - 0
Byte5 - 7	Byte5 - 6	Byte5 - 5	Byte5 - 4	Byte5 - 3	Byte5 - 2	Byte5 - 1	Byte5 - 0
Byte4 - 7	Byte4 - 6	Byte4 - 5	Byte4 - 4	Byte4 - 3	Byte4 - 2	Byte4 - 1	Byte4 - 0
15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0

Abbildung A.17: CAN Acceptance-Filter Message High Output Register *
In diesem Register werden die Byte 4-7 der akzeptierten Nachricht ausgegeben.

A.18 Pseudocode

Hier soll nun ein Pseudocode zur Initialisierung des Controllers sowie zu Pseudocode zu einer Interruptserviceroutine gegeben werden.

```
1  //Initialisierung des Controllers
2  int init(void){
3  /*zuerst das Bittiming einstellen:
4   das Timing wurde von http://www.port.de/pages/misc/bittimings.php?lang=en
        uebernommen
5   1 MBit bei 50 MHz Samplepoint bei 68%
6   Prescalerate := 0x01
7   Propdelay := 0x07
8   Phase1 := 0x07
9   Phase2 := 0x07
10  SJW    := 0x03
11  Multisampling aktiviert */
12 CAN_BT = 0x07007777;
13
14 /*Acceptance-Filter 1 so einstellen, dass alle Remoteframes angenommen
15   werden: */
16 CAN_ACC_ID_TAG1  = 0x00000001;
```

```
17 CAN_ACC_ID_MASK1 = 0x00000001;
18
19 /*Acceptance-Filter 2 so einstellen, dass
20  nur Dataframes mit einem Base-Identifier zwischen 0 und 127
21  sowie einem beliebigen Extended-Identifier angenommen werden. */
22 CAN_ACC_ID_TAG2  = 0x00000000;
23 CAN_ACC_ID_MASK2 = 0xf8000001;
24 /* Interrupts enablen:
25  Interrupt wenn Acceptance-Filter1 eine Nachricht angenommen hat.
26  Interrupt wenn Acceptance-Filter2 eine Nachricht angenommen hat.
27  Interrupt wenn der Controller in BUS-OFF geht. */
28 CAN_GIE = 0x00000203;
29 //der Controller und Acceptance-Filter 1 und 2 sollen enabled werden.
30 CAN_EN  = 0x80000003;
31
32 //Initialisierung abgeschlossen.
33 return 0;
34 }
```

Listing A.1: Pseudocode zur Initialisierung des Controllers

```
1  struct frame {
2      int identifier;
3      int data_high;
4      int data_low;
5      int dlc;
6  };
7
8  void ISR(){
9  int IRQ;
10 frame dataframe;
11 //zuerst ueberpruefen, welcher Interrupt aufgetreten ist:
12 IRQ = CAN_STI;
13 //Acceptance-Filter1 :
14 if((IRQ & 0x00000001) == 0x00000001){
15
16     // Lege empfangenen Identifier in Puffer
17     // fuer Remoteframes
18     put_buff(&remotebuff, CAN_ACC_ID_OUT1);
19
20 }
21 //Acceptance-Filter2 :
22 if((IRQ & 0x00000002) == 0x00000002{
23     //lese zuerst den Datalengthcode ein:
24     dataframe->dlc = CAN_DLC_OUT & 0x000000f0;
25     dataframe->dlc = dataframe->dlc >> 4;
26
27     //lege empfangenen Identifier in struct
28     dataframe->identifier = CAN_ACC_ID_OUT2;
```

```
29
30      //wenn Daten empfangen wurden lege diese in das struct:
31      if(dataframe->dlc > 0){
32          dataframe->data_low = CAN_ACC_MSG_LOW_OUT2;
33
34          //wenn mehr als 4 Byte empfangen wurden:
35          if(dataframe->dlc > 4){
36              dataframe->data_high = CAN_ACC_MSG_HIGH_OUT2;
37          }
38      }
39      //lege struct im Puffer fuer Dataframes ab:
40      put_buff(&databuff, dataframe);
41  }
42  //BUS-OFF:
43  if((IRQ & 0x00000200) == 0x00000200){
44
45      //irgendeine Fehlerbehandlung
46  }
47
48
49  }
```

Listing A.2: Pseudocode für eine Interruptserviceroutine

B Inhalt der CD

Hier soll einer kurzer Überblick über den Inhalt der CD gegeben werden. Die CD enthält vier Ordner und die Bachelorarbeit in elektronischer Form als PDF. Die Ordner und deren Inhalt werden im Folgenden erklärt.

B.1 Literatur

In diesem Ordner befinden sich alle Dokumenten die für diese Bachelorarbeit verwendet wurden und in elektronischer Form vorliegen.

B.2 Quellcode

Dieser Ordner beinhaltet zwei Unterordner. Zum einen einen Ordner für VHDL-Code und zum anderen einen Ordner für Code der in C verfasst wurde.

B.2.1 VHDL-Code

Dieser Ordner ist unterteilt in vier Ordner:

- **CAN-Controller**: Enthält den Quellcode des CAN-Controller ohne Testbench und µPI.
- **HW-Test**: Enthält den kompletten Quellcode des Controllers, der für den Hardwaretest verwendet wurde. Inklusive einer Testbencht.
- **Simulation**: Enthält den kompletten Quellcode des Controllers zur VHDL- und Timingsimulation. Inklusive einer Testbencht.
- **µPI**: Enthält den Quellcode zum µPI, wie dieses für den Controller verwendet wurde

B.2.2 C-Code

Dieser Ordner beinhaltet zwei Ordner:

- **AVR-Code**: Enthält den Quellcode für den AVR, der für den Test in Hardware verwendet wurde.
- **PRBS-Test**: Enthält den Quellcode zum Testen des Polynoms für den PRBS16-Test.

B.3 Synthese

Dieser Ordner beinhaltet alle Ausgaben des Synthesetools. Dieser Ordner enthält zwei Ordner:

- **HW-Test**: Beinhaltet die Ausgaben des Synthesetools zum Hardwaretest.
- **Timing-Simulation**: Beinhaltet die Ausgaben des Synthesetools für die Timingsimulation.

B.4 Sonstiges

In diesem Ordner ist alles abgelegt, was für diese Bachelorarbeit verwendet wurde und keinen direkten Bezug zu CAN oder dem entwickelten CAN-Controller hat. Dieser Ordner enthält auch das Printout des AVR aus dem Hardwaretest.

Glossar

ACK Acknowledgement - Quittung, Bestätigung

CAN Controller Area Network

CRC Cyclic Redundancy Check

CSMA Carrier Sense Multiple Access

DLC Datalengthcode

DUT Device Under Test

FPGA Field Programmable Gate Array

LFSR Linear-Feedback-Shift-Register

LLC Logical Link Control

LSB Least Significant Bit - niederwertigstes Bit

MAC Medium Access Control

MSB Most Significant Bit - höchstwertigstes Bit

NAK Negative Acknowledgement

NRZ Non Return to Zero

PRBS Pseudo Random Bit Sequence

REC Receive Error Count - Empfangsfehlerzähler

RX Receiver - Empfänger

RX_OK Reception OK - Empfang war erfolgreich

Samplepoint Zeitpunkt, an dem der Bus-Pegel übernommen wird

SoC System on (a) Chip

SOF Start of Frame

TEC Transmit Error Count - Sendefehlerzähler

TQ Time Quantum - Zeitquantum

TX Transmitter - Sender

TX_OK Transmission OK - Versand war erfolgreich

VHDL Very high speed intergrated circuit Hardware Description Language

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, 2. Januar 2010	
Ort, Datum	Unterschrift