Fachhochschule Dortmund
University of Applied Sciences and Arts

Bachelorthesis

Entwurf einer I2C zu CAN Brückenlogik in VHDL und Entwicklung einer Softwareumgebung zur Durchführung von Systemtests

Fachhochschule Dortmund

Fachbereich Elektrotechnik

Student: Armin Kuka

Erstprüfer: Prof. Dr.-Ing. Michael Karagounis

Zweitprüfer: M.Eng. Alexander Walsemann

Abgabedatum: 17.12.2019

Kurzzusammenfassung

Entwurf einer I2C zu CAN Brückenlogik in VHDL und Entwicklung einer Softwareumgebung zur Durchführung von Systemtests

In dieser Bachelorthesis wird die Inbetriebnahme eines Kommunikationssystems sowie die Entwicklung einzelner Bestandteile erläutert. Das System beinhaltet eine Softwareoberfläche, über die ein CAN-Interface sowie ein I2C-Master angesteuert werden können. Diese sind über das jeweilige Bussystem an einen CAN-Controller bzw. I2C-Slave angeschlossen, welche beide auf demselben FPGA Baustein implementiert sind. Der CAN-Controller sowie das I2C-Slave sind über eine Brückenlogik verschaltet, die es in toto ermöglicht, dass Daten zwischen den Bussystemen übertragen werden können. Im Rahmen dieser Arbeit wurde die Brückenlogik und die Softwareoberfläche entwickelt.

Abstract

Design of an I2C to CAN bridge logic in VHDL and development of a software environment for system tests

This bachelor thesis explains the commissioning of a communication system and the development of individual components. The system includes a software interface that can be used to control a CAN interface and an I2C-Master. These are connected with their respective bus system to a CAN controller respectively to an I2C-Slave, which are both implemented on the FPGA device. The CAN controller as well as the I2C slave are interconnected via a bridge logic, which allows altogether that data can be transferred between the bus systems. In this work, the bridge logic and software interface were developed.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass die von mir vorgelegte Prüfungsleistung selbstständig und ohne unzulässige fremde Hilfe erstellt worden ist. Alle verwendeten Quellen sind in der Arbeit so aufgeführt, dass Art und Umfang der Verwendung nachvollziehbar sind.

_____ _____

Ort, Datum Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

VHDL	Very High Speed Integrated Circuit Hardware Description Language
FPGA	Field Programmable Gate Array
ASIC	application-specific integrated circuit
I²C	Inter Integrated Circuit
SDA	Serial Data
SCL	Serial Clock
ACK	Acknowledge
NACK	Not-Acknowledge
Vdd	Versorgunsspannung
MSB	most significant bit
LSB	least significant bit
Z	hochohmig
SDI	Serial data input
SDO	Serial data output
USB	Universal Serial Bus
ILA	Intergrated Logic Analyzer
CAN	Controller Area Network
SOF	Start of Frame
RTR	Remote Transmission Request
IDE	Identifier-Extension
CRC	Cyclic Redundancy Check

1 Einleitung

1.1 DCS-Controller-Chip

Der Detector Control System Controller (DCS-Controller) ist ein sich noch in der Entwicklungsphase befindlicher Mikrochip, der im Zuge eines geplanten Upgrades am ATLAS-Pixeldetektor für die Kommunikation zwischen intelligenten Sensorknoten den sogenannten DCS-Chips und dem Kontrollzentrum entwickelt wurde. Dieser DCS-Controller kommuniziert über die seriellen Busprotokolle I2C und CAN jeweils mit den direkt an den Detektormodulen eingesetzten DCS-Chips und mit den DCS-Computern.

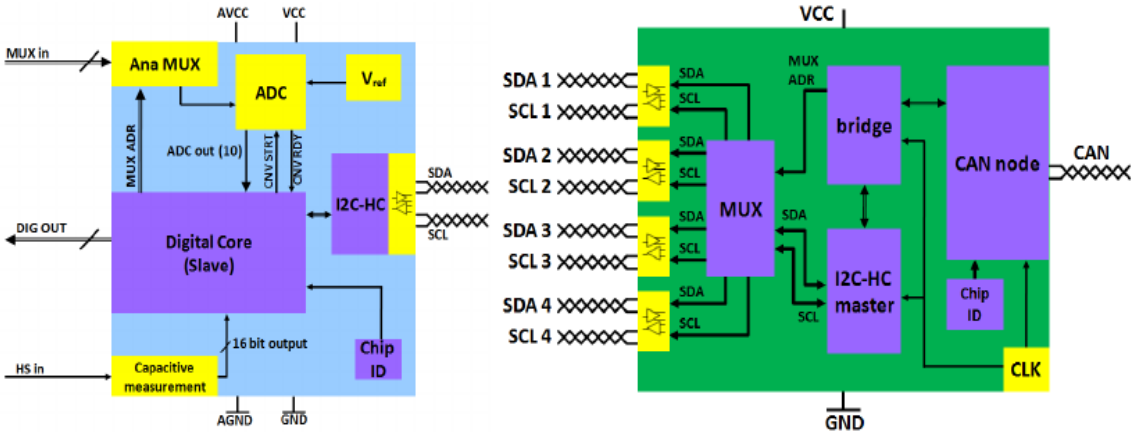

Abbildung 1: DCS-Chip mit DCS-Controller [1]

Der DCS-Controller erweitert die Kommunikationsreichweite des DCS Chips, durch die Umwandlung der I2C-Signale mittels eine integrierten Brückenlogik in CAN-Nachrichten, sodass die höhere Reichweite des CAN-Busses genutzt werden kann. Die Anzahl der benötigten Leitungen wird ebenfalls reduziert. Ein Prototyp mit ersten Strukturen des DSC-Controllers, wurde an der Fachhochschule Dortmund entwickelt und Anfang 2019 in TSMC 65nm CMOS Technologie in Produktion gegeben.

1.2 Testsystem

Im Zuge der erfolgreichen Nutzung von weitreichenden Simulationsmodellen, sollte als nächster Schritt ein Kommunikationssystem in richtiger Hardware realisiert werden. Dieses soll zu Testzwecken für zunächst einzelne Komponenten des DCS-Controllers verwendet werden. Es werden auf einem FPGA ein CAN-Controller, ein I2C-Slave und eine in dieser Arbeit entwickelte Bridge zusammen verschaltet, um die protokollübergreifende Kommunikation zu ermöglichen. Diese Schaltung wird an einem CAN- und I2C-Bus angeschlossen. Über eine Softwareoberfläche, die ebenfalls in dieser Arbeit entwickelt wurde, lassen sich ein I2C-Master und ein CAN-Inteface ansteuern. Dies entspricht dem in Abbildung 2 gezeigten schematischen Aufbau.

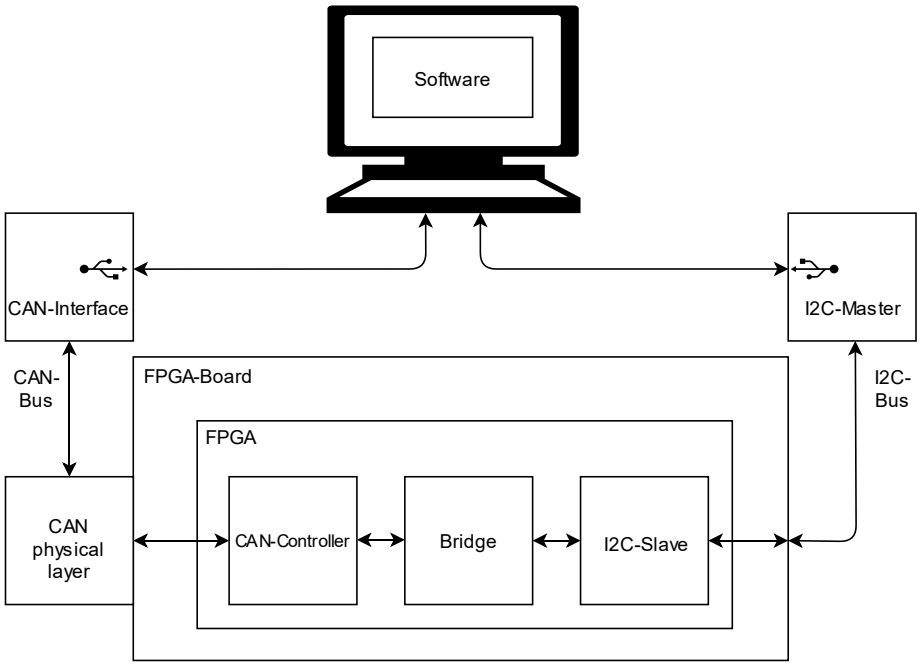

Abbildung 2: Das Testsystem.

Dieser Aufbau erlaubt es Daten über den I2C Bus in die Register des CAN Controllers im FPGA zu schreiben und oder daraus zu lesen. Dadurch kann der CAN Controller flexibel für Systemtests konfiguriert werden. Daten, die der CAN Controller empfangen hat, können auf diesem Weg ausgelesen und zum Vergleich mit den ursprünglichen Daten an den PC übergeben werden. Außerdem kann der PC Daten, die der CAN Controller im FPGA versenden soll, auf diese Weise in die Register des CAN Controllers übertragen. Im Folgenden werden die verwendeten Busprotokolle und die für die Kommunikationen genutzten Bausteine näher vorgestellt.

2 I²C-Protokoll

Das I²C-Protokoll ist ein Busprotokoll, welches 1982 von Philips Semiconductors (heute NXP Semiconductors) entwickelt wurde. Daten werden nach dem Master-Slave-Prinzip zwischen den hierarchisch übergeordneten Mastern und den untergeordneten Slaves seriell übertragen. Hierbei findet die Steuerung der Kommunikation durch den Master statt, dem Slave ist eine passive Rolle zugewiesen, dieser empfängt und sendet Daten nur auf Aufforderung eines Masters. Seriell bedeutet, dass die Datenpakete in Form von Bytes, Bit für Bit nacheinander übertragen werden. Es sind je nach Modus Übertragungsrate von bis zu 3,4 Mbit/s bidirektional und 5 Mbit/s unidirektional möglich.

2.1 Elektrische Spezifikation

Der I²C-Bus benötigt nur die zwei Leitungen SDA und SCL, wobei SDA für „Serial Data“ und SCL für „Serial Clock“ steht. Auf der SDA Leitung können Daten sowohl vom Master, als auch vom Slave gesendet und empfangen werden. Über die SCL Leitung wird das vom Master gesendete Taktsignal, zur Synchronisation an die Slaves übermittelt. Die Busleitungen sind im Grundzustand HIGH. Der Zustand LOW kann durch jeden einzelnen Busteilnehmer erzeugt werden, indem dieser die Leitung auf Masse zieht. Dies wird erreicht, indem die Busleitungen über Pull-Up-Widerstände an die positive Spannungsversorgung angeschlossen werden.

Seit der 6. Version der I²C Spezifikation von 2014 werden die maximalen und minimalen Werte der Widerstände R_p über Funktionen definiert. [2, p. 55]

$$R_{p(max)} = \frac{t_r}{0{,}8473 \cdot C_b} \tag{1}$$

$$R_{p(\min)} = \frac{V_{DD} - V_{OL(max)}}{I_{OL}} \tag{2}$$

Die Formel (1) beinhaltet einmal die Anstiegszeit der positiven Flanke beider Busleitungen t_r sowie die Kapazität der jeweiligen Busleitung C_b als Variablen.

Die Kapazität C_b ist von der Länge der Leitungen und der Anzahl der angeschlossenen Teilnehmer abhängig. Der maximal erlaubte Wert beträgt 400 pF. Dieser Wert kann je

nach eingesetzter Frequenz und Versorgungsspannung variieren. Die Anstiegszeit t_r ist durch die endliche Grenzfrequenz der gesamten Anordnung der Teilnehmer inklusive der Leitung bedingt. Mit der Anstiegszeit ist in der Regel der Zeitraum gemeint, welches ein Signal benötigt, um gemessen von 10%, 90% seines Sollwertes zu erreichen. In der I²C-Spezifikation ist jedoch der Zeitraum zwischen 30% und 70% herangezogen worden, da nach Spezifikation eine Eins bereits bei 0,7 V_{DD} und eine Null von -0,5 V bis einschließlich 0,3 V_{DD} am Bus anliegt, sodass der Anstieg von 30% auf 70% V_{DD} dem Übergang von LOW zu HIGH entspricht.

Die Größen Formel (2) entsprechen der Versorgungsspannung V_{DD}, der maximal anliegenden Ausgangsspannung $V_{OL(max)}$ bei einem LOW Buszustand und der Strom I_{OL}, der durch den Teilnehmer, der die Leitung auf Masse zieht fließt. Die Versorgungsspannung V_{DD} ist hierbei so zu wählen, dass der Wert von 5,5 V nicht überschritten wird. Für die meisten Anwendungen im Standartmodus reichen 10 kΩ Pull-Up-Widerstände aus.

Des Weiteren müssen alle am Bus angeschlossenen Teilnehmer ihre Ausgänge hochohmig bzw. auf Z schalten, wenn sie nicht aktiv eine Null setzen wollen. Dies wird auf der Transistorebene mit einer Wired-And-Verknüpfung in Form einer Open-Collector- bzw. Open-Drain-Schaltung erreicht. Dabei wird der Kollektor des Bipolartransistors, bzw. der Drain des Feldeffekttransistors an die Busleitung gelegt, Emitter- bzw. Source wird an Masse gelegt. Je nachdem was für ein Signal an der Basis bzw. dem Gate anliegt, sperrt oder leitet der Transistor. Im ersten Fall hat der SDA Ausgang des Teilnehmers den Wert Z, im zweiten Fall eine Null. Somit wird eine Eins nicht aktiv von den Teilnehmern erzeugt, sondern durch die Pull-Up Widerstände, wenn alle Teilnehmer ihre Ausgänge auf Z geschaltet haben. Zieht ein einziger Teilnehmer den Bus auf Masse, so liegt überall eine Null an. Im Standardmodus darf dazu der Bus nur über eine maximale Frequenz von 100 kHz getaktet werden.

In Abbildung 3 ist ein schematisches Beispiel für ein I²C-System mit einem Master und zwei Slaves dargestellt.

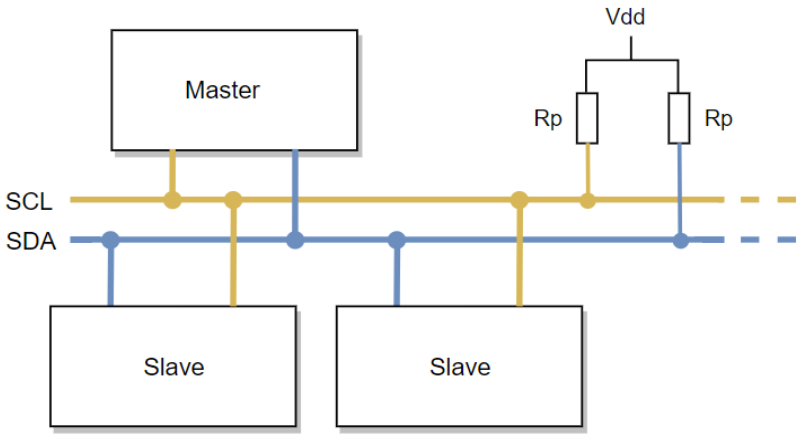

Abbildung 3: Beispiel eines I²C-Systems

2.2 Datenübertragung

Daten werden in Byte-Paketen zwischen den einzelnen Teilnehmern versendet. Der Initiator ist hierbei der Master. Dieser erzeugt eine Startbedingung, sodass alle über den Bus erreichbaren Slaves sich auf den kommenden Datenverkehr einstellen können.

Daraufhin wird vom Master die Slave-Adresse über SDA übertragen sowie ein Bit, welches den Slaves mitteilt ob, es sich um einen lesenden oder schreibenden Zugriff handelt. Die Slaves überprüfen nun die vom Master gesendete Adresse. Erkennt ein Slave nun seine intern abgespeicherte Adresse, so setzt es vor der 9. steigenden Taktflanke das ACK-Bit (engl. acknowledge), in dem SDA auf Null gezogen wird. Jenes wird vom Master erkannt und dieser stellt sich nun auf das Empfangen oder Senden von Daten ein. Daraufhin wird vom Master oder vom Slave das erste Byte versendet.

Wenn der Master als Empfänger fungiert, so muss er selbst das ACK-Bit setzen, um dem Slave mitzuteilen, dass die Datenübertragung fortgeführt werden soll. Daten werden solange übertragen bis von keinem Teilnehmer ein ACK gesetzt wird, dies wird auch als NACK (engl. not acknowledge) bezeichnet. Als Nächstes wird ein Stoppsignal erzeugt, welches beide Leitungen wieder auf 1 zieht.

Übermittelte Bits sind nur beim Taktzustand HIGH der SCL Leitung gültig, währenddessen darf sich SDA nicht ändern. In der Regel verwendet ein Slave auch weitere Adressen für die Unterscheidung interner Register. Daher wird bei der Adressierung durch den Master neben der Slave-Adresse, auch eine Registeradresse versendet. Diese muss jedoch in vielen Anwendungsfällen nur am Anfang der Kommunikation mitgeteilt werden. Slaves inkrementieren die intern ausgewählte Registeradresse, nach einem lesenden oder schreibenden Registerzugriff so nach einem ACK das nächste Register für die jeweilige Operation ausgewählt wird.

2.2.1 Startbedingung

Die im vorherigen Abschnitt erwähnte Startbedingung ist wie folgt definiert. Eine Startbedingung tritt dann auf, wenn auf SDA ein Übergang von HIGH auf LOW stattfindet, während SCL auf HIGH ist. Die Startbedingung ist in der Abbildung 4 dargestellt. Sie kann auch in Form eines Repeated-Starts vorkommen (Abbildung 5), welches auf das ACK folgt, um z.B. einen anderen Slave anzusprechen. Die Kommunikation wird durch einen Repeated-Start nicht beendet, die Slaves erwarten unmittelbar danach ein Adressbyte. Eine Startbedingung kann nur erzeugt werden, wenn der Bus freigegeben ist.

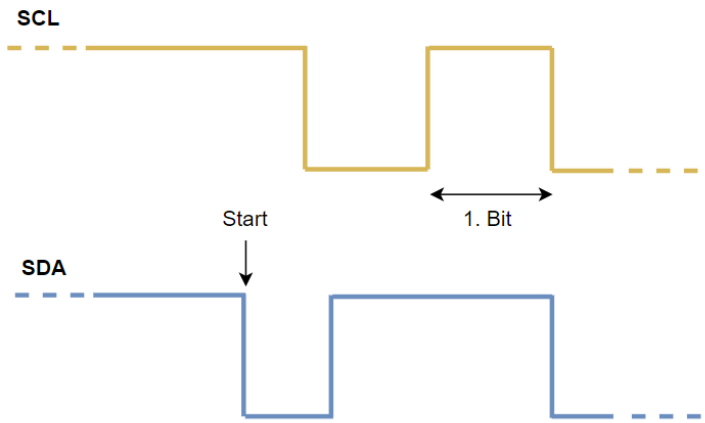

Abbildung 4 : Startbedingung, die Kommunikation beginnt, das 1. Bit ist eine Eins.

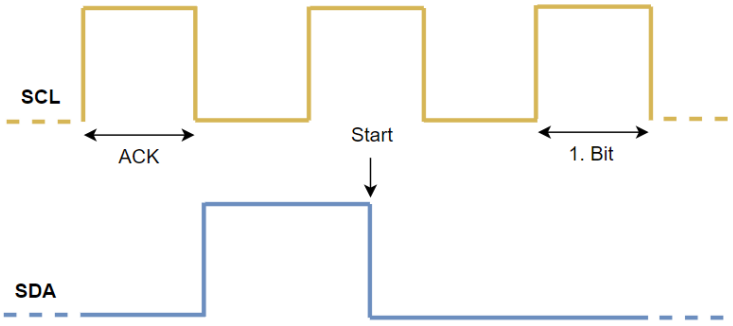

Abbildung 5: Repeated-Start, das 1. Bit ist eine Null.

2.2.2 Stoppbedingung

Die Stoppbedingung ist das Pendant zur Start-Bedingung, sie liegt dann vor, wenn auf SDA ein Übergang von LOW auf HIGH stattfindet, während SCL auf HIGH ist. Ein Stoppsignal beendet die Kommunikation aller Teilnehmer und setzt diese zurück. Beide

Leitungen werden danach freigegeben und verbleiben auf HIGH, sofern alle Teilnehmer das Stoppsignal korrekt erkannt haben. Erst durch eine neue von einem Master initiierte Startbedingung kann wieder eine Kommunikation stattfinden.

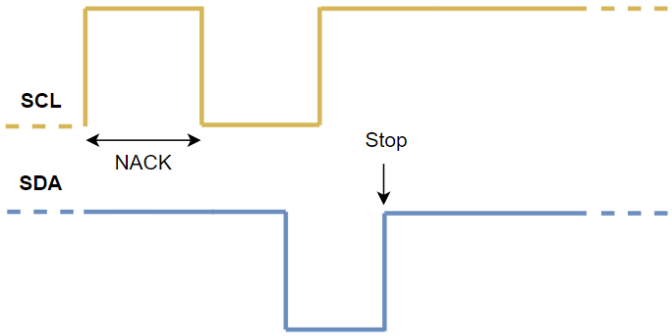

Abbildung 6: Auf ein NACK folgende Stoppbedingung.

2.2.3 Acknowledge-Bit/ Not Acknowledge

Das ACK bzw. NACK wird immer nach dem 8. Bit versendet und dient dazu, dass die Kommunikationseinheit den erfolgreichen Datenaustausch bestätigt. Ein ACK hat den Wert Null, das NACK Eins. Wird ein NACK gesetzt, so führt dies zu einem Abbruch der Kommunikation.

Es gibt laut Spezifikation fünf Gründe, die zu einem NACK führen:

1. Es ist kein Empfänger mit der versendeten Adresse am Bus angeschlossen.
2. Dem Empfänger ist es nicht möglich, die Adresse oder das ACK zu verarbeiten, weil dieser zurzeit andere Funktionen ausführt und noch nicht bereit ist mit dem Master zu kommunizieren.
3. Während des Transfers erhält der Empfänger Daten oder Befehle, welche ihm unbekannt sind.
4. Während des Transfers ist es dem Empfänger nicht möglich weitere Daten zu empfangen.
5. Ein Master im Empfangsmodus signalisiert dem sich im Sendemodus befindlichen Slave das Ende der Kommunikation. [2, p. 10]

2.2.4 Synchronisation des Taktsignals

Da nach Spezifikation der Master das Taktsignal erzeugt und grundsätzlich mehrere Master-Instanzen an einem Bus betrieben werden dürfen, muss geregelt sein, mit welchem Taktsignal nun die Datenübertragung stattfindet. Für den Fall das sich mehrere Master-Taktsignale überlappen, wird SCL aufgrund der Wired-And-Verknüpfung

solange auf LOW gehalten, bis auch der letzte Master sein Ausgang von LOW auf HIGH (technisch Z) geschaltet hat. Während dieser Zeit zählen die Master ihre LOW-Periode runter. Master, welche den Zählvorgang beendet haben schalten ihren Ausgang auf HIGH und verweilen in einem Wartezustand. Sobald der letzte Master seine LOW-Periode runtergezählt hat, wird der Bus freigegeben und wechselt auf HIGH. Ab diesem Zeitpunkt zählen die Master nun ihre HIGH-Periode herunter. Sobald der erste Master wieder seinen Ausgang von HIGH auf LOW wechselt, zieht er für alle Teilnehmer den Bus auf Masse. Mit dieser Funktion wird eine synchrones Clock-Signal erzeugt, dessen LOW-Periode von dem jeweiligen Master mit der längsten LOW-Periode und dessen HIGH-Periode von dem Master mit der kürzesten HIGH-Periode erzeugt wird.

2.2.5 Clock stretching

„Clock stretching“ erlaubt es auch Busteilnehmern die Datenübertragung auszusetzen, wenn diese mit der Datenverarbeitung nicht mithalten können. Hierzu wird die SCL-Leitung nach dem ACK solange LOW gehalten, bis die vorher empfangenen Daten verarbeitet und der Teilnehmer wieder bereit ist, an der Kommunikation teilzunehmen. Master-Instanzen wechseln währenddessen in einen Wartezustand und SDA wird von allen Teilnehmern freigeben.

2.2.6 Arbitrierung der Zugriffe auf SDA

Wenn mehrere Master-Instanzen eine Startbedingung innerhalb der minimalen Haltedauer erzeugen, so muss für diesen Fall bestimmt werden, wessen Daten auf SDA nun gültig sind. Hierzu wird wieder die Wired-And-Verknüpfung genutzt. Während die Master-Instanzen Ihre Daten auf den Bus legen, kontrollieren sie gleichzeitig bitweise, ob der auf den Bus vorhandene Wert auch jener ist, den der jeweilige Master selbst auf den Bus legt. So verliert ein Master den Buszugriff, für den Fall, dass ein anderer Master den Bus auf LOW zieht, während er selbst ein HIGH geschrieben hat. Welcher Master den Zugriff nun gewinnt, hängt einzig und allein von den Adressdaten ab. Sendet Master 1 die Daten „0001000“ und Master 2 die Adresse „00000111“, so gewinnt der Master 2 während des 4. Taktes die Kontrolle über den Bus, weil das 4. Bit des zweiten Masters das 4. Bit des ersten Masters überschreibt. Aus dem obigen Prozess lässt sich erschließen, dass es keine priorisierten Master-Instanzen geben kann.

2.3 Adressierung

Bevor Daten übermittelt werden können, muss vorher ein Master den Slave adressieren, mit welchem der Datenaustausch stattfinden soll. Mastereinheiten haben keine eigene Adresse und können dementsprechend auch untereinander nicht unmittelbar kommunizieren. Dies ist nur durch hintereinander folgende Zugriffe auf Slaves möglich. Das I²C-Protokoll sieht für die Adressierung von Slaves zwei Varianten vor. Einmal steht die 7- Bit-Adressierung zur Verfügung, welche es erlaubt, 2^7 bzw. 128 Adressen zu vergeben. Nach der Spezifikation sind jedoch zusätzlich 16 Adressen für andere Zwecke reserviert, womit sich die Anzahl der Slaves, die über 7-Bit adressiert werden können, auf 112 reduziert. Die reservierten Adresen sind in der Tabelle 1 aufgelistet. Für die Adressierung einer größeren Anzahl an Slaves steht die 10-Bit-Adressierung zur Verfügung. Im 10-Bit Adressformat wird die Adresse über zwei Bytes verteilt verschickt. Damit es zu keinen Diskrepanzen kommt, beinhaltet das erste Byte einen Code, mit denen die Slaves erkennen können, dass es sich um das erste Paket einer 10-Bit-Adresse und nicht um eine 7-Bit-Adresse handelt. Dieser Code nimmt bereits vier der oben genannten 16 Adressen ein. Mit der 10-Bit-Adressierung sind dementsprechend 2^{10}, daher 1024 Adressen verfügbar. Werden beiden Adressierungsverfahren kombiniert, können bis zu 1136 Slave-Instanzen an einem Bus betrieben werden.

Tabelle 1: Für 7-Bit Slave-Instanzen nicht verwendbare Adressen. [2, p. 17]

Adresse	R/W	Verwendungszweck
0000000	0	general-call Adresse
0000000	1	START byte
0000 001	X	CBUS Adresse
0000 010	X	Reserviert für andere Busformate
0000 011	X	Reserviert für zukünftige Änderungen
0000 1XX	X	Reserviert für den high-speed mode
1111 1XX	1	Reserviert für die Geräte Identifikation
1111 0XX	X	Code für die 10-Bit-Adressierung

2.3.1 7-Bit-Adressierung

Die Adressierung von 7-Bit Slaves muss nach dem in Abbildung 7 dargestellten Schema erfolgen. Zunächst wird die Kommunikation von einem Master mit der Erzeugung einer Startbedingung eingeleitet. Daraufhin wird die 7-Bit lange Slave-Adresse versendet.

Dabei wird zeitlich das MSB (engl. most significant bit) als erstes auf den Bus gelegt und das LSB (engl. least significant bit) als letztes. Die Bits müssen wie bereits erwähnt vor der steigenden Flanke auf den Bus gelegt und bis nach der fallenden Flanke von SCL gehalten werden, ansonsten würde eine Start- bzw. Stoppbedingung vorliegen.

Nach der Adresse folgt das Read-Write-Bit „R/W“, welches vom Master gesetzt wird und bestimmt, ob die nachfolgenden Daten vom Master oder vom vorher adressierten Slave gesendet werden müssen. Ein auf Eins gesetztes R/W deutet dabei einen Lesezugriff bzw. ein auf Null gesetztes R/W dementsprechend einen Schreibzugriff an. Nachdem die Adresse und das Read-Write-Bit gesendet wurde, zieht das Slave, welches seine Adresse erkannt hat, den Bus auf Null, um damit wird das ACK Bit zu setzen und dem Master eine erfolgreiche Adressierung zu signalisieren. Antwortet kein Slave auf die vom Master versendete Adresse mit einem ACK, wird die Kommunikation als Nächstes mit einer Stoppbedingung ohne die Versendung weiterer Daten beendet.

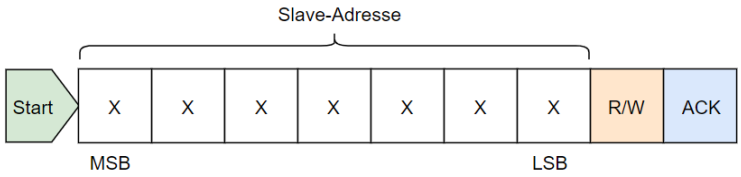

Abbildung 7: Ablauf der 7-Bit-Adressierung.

Slaves haben in der Regel zusätzlich Register, die einzeln angesprochen werden können, dementsprechend können nach der Slave-Adresse weitere Adress-Bytes von Registern folgen.Ein Beispiel wie die Kommunikation mit einer zusätzlichen Registeradresse aussieht ist in Abbildung 8 gezeigt. Die von einem Master ausgehenden Daten sind in einem rötlichen, die des Slaves in einem bläulichen Farbton hervorgehoben. Der Master erzeugt als Erstes eine Startbedingung, daraufhin wird die Slave-Adresse und ein Write-Bit versendet. Das Slave wiederum erkennt seine Adresse und setzt ein ACK. Daraufhin verschickt der Master nun die Registeradresse. Nach erfolgtem Empfang der Registeradresse setzt das Slave abermals ein ACK. Nach einem Repeated-Start erfolgt die erneute Übermittlung der Slave-Adresse in Kombination mit einem Read-Bit. Da diesmal das R/W Bit bei der Slave-Adressierung gesetzt ist, legt das Slave als Nächstes die im Register gespeicherten Daten auf den Bus. Nun muss der Master ein ACK senden, falls das Slave weitere Daten versenden soll. Nach dem ACK des Masters, wird in der Regel die im Slave ausgewählte Registeradresse um eins inkrementiert und dann der Speicherinhalt des nächsten Registers ausgegeben. Alternativ ist nach dem ACK ein

Repeated-Start möglich, um ein Register mit einer ganz anderen Adresse oder Registerdaten aus einem anderen Slave auszulesen oder zu schreiben. In Abbildung 8 wurde vom Master jedoch ein NACK gesetzt, daraufhin folgt eine Stoppbedingung und die Kommunikation wird bis zum nächsten Start-Signal beendet.

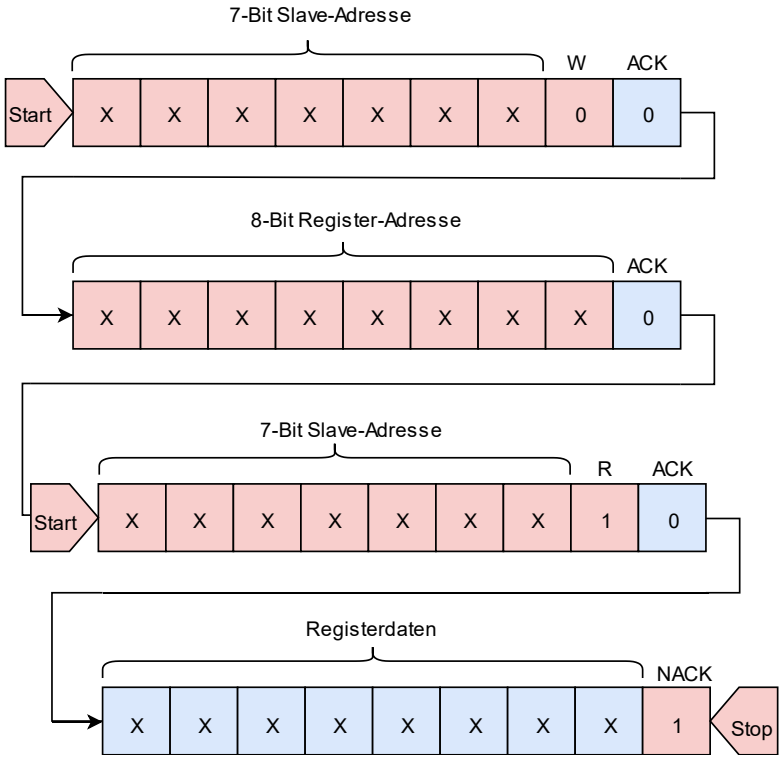

Abbildung 8: Das Auslesen von Daten eines Slave-Registers durch einen Master. Rot: Master-Aktionen, Blau: Slave-Aktionen.

In der Abbildung 9 ist der schreibenden Zugriff des Masters auf ein Slave-Register zu erkennen. Wie oben bereits beschrieben, läuft die Adressierung nach dem gleichen Schema ab. Diesmal wird kein „Repeated-Start“ ausgelöst, wodurch die Daten direkt im 3. Byte nun vom Master gesendet und vom Slave empfangen werden.

Die erhaltenen Daten müssen vom Slave mit einem ACK bestätigt werden. Ein NACK interpretiert der Master als fehlgeschlagene Übertragung. Die Stoppbedingung wird vom Master erzeugt, wenn dieser keine weiteren Register beschreiben möchte.

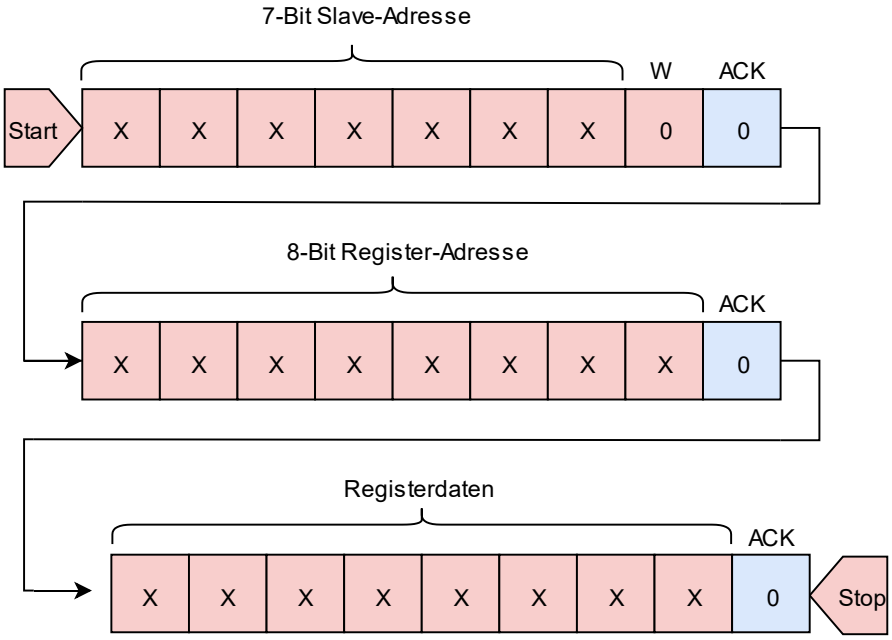

Abbildung 9: Das beschreiben eines 8-Bit Slave-Registers. Rot: Master-Aktionen, Blau: Slave-Aktionen.

2.3.2 10-Bit-Adressierung

Wie im Abschnitt 2.3 bereits erläutert, wird bei der 10-Bit-Adressierung zunächst der reservierte 10-Bit Code versendet. Danach folgen die ersten zwei Bits der 10-Bit-Slave-Adresse sowie das Read-Write-Bit. Erkennt mindestens ein Slave den Code und stimmen die ersten zwei gesendeten Adressbits mit der eigenen Adresse überein, so wird das ACK vom Slave gesetzt. Als Nächstes wird von allen 10-Bit Instanzen, die ihre ersten zwei Adressbits erkannt haben, ein weiteres Adressbyte erwartet.

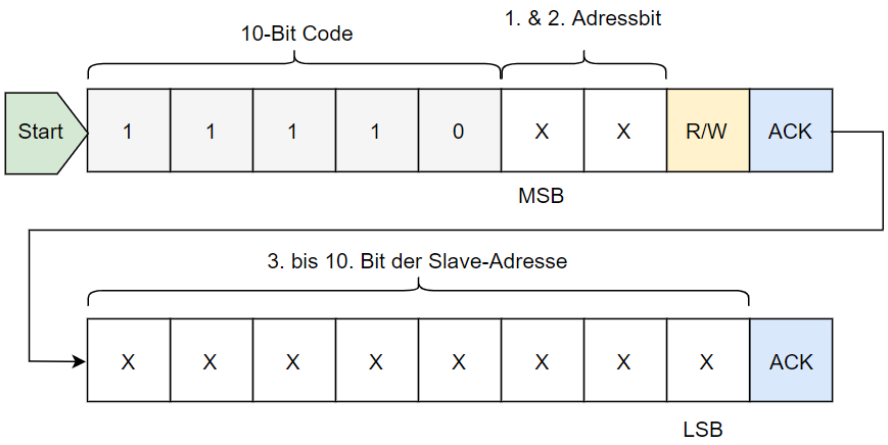

Abbildung 10: Ablauf der 10-Bit Adressierung.

Entspricht der Inhalt des zweiten Adressbytes zusammen mit den ersten zwei Adressbit einer vorhandenen Slave-Adresse, so setzt der adressierte Slave das 2. ACK wie in Abbildung 10 dargestellt. In Abbildung 9 ist ein Schreibvorgang und in Abbildung 10 ein Lesevorgang dargestellt. Beim Schreibvorgang ist es ähnlich wie bei der 7-Bit Adressierung. Nach der Übermittlung der Adresse und des R/W-Bits folgt die Registeradresse und daraufhin wird das Byte, welches in das Slave-Register geschrieben werden soll, übermittelt. Der Auslesevorgang von Daten mit 10-Bit Adressierung ist jedoch einiges komplexer. Zunächst wird die 10-Bit Adresse mit einem R/W-Bit von Null sowie der Adresse des internen Slave-Registers übermittelt. Damit unterscheidet sich die Adressierung zunächst nicht von der eines Schreibvorgangs. Daraufhin folgt jedoch ein Repeated-Start (Abbildung 5) und es wird wieder das erste Byte der 10-Bit Adressierung verschickt und zwar diesmal mit gesetztem R/W-Bit. Danach gibt der Master den Bus frei und der Slave setzt die Inhalte des Registers auf den Bus. Soweit der Master keine weiteren Daten braucht, setzt dieser ein NACK und beendet die Kommunikation mit einem Stopp-Signal.

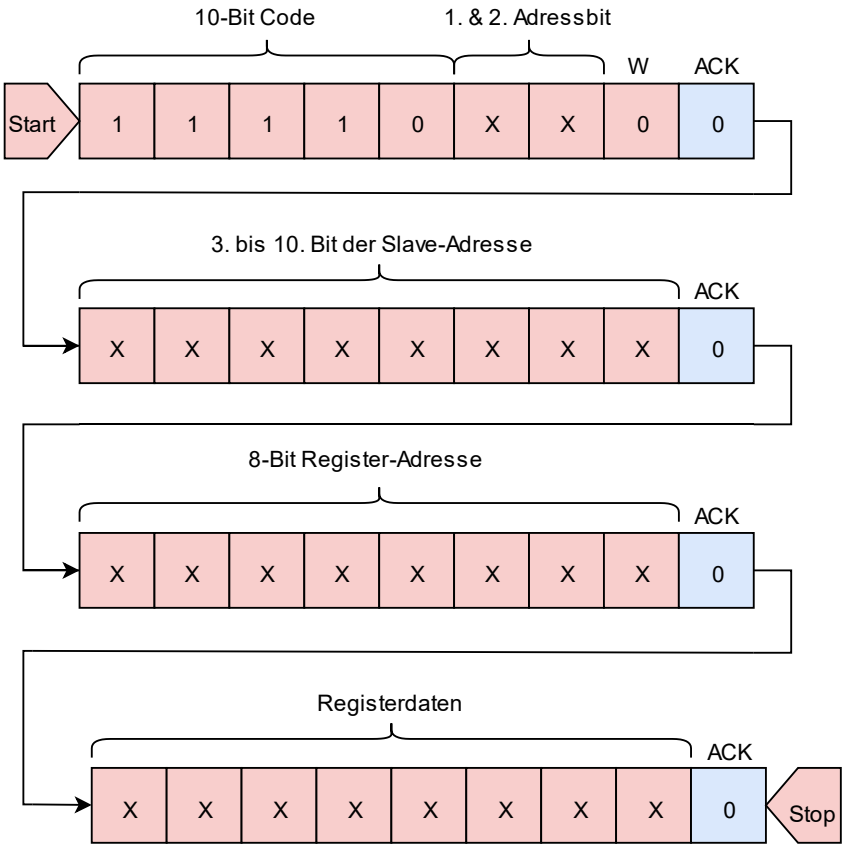

Abbildung 11: Das Beschreiben eines 8-Bit Slave-Registers. Rot Master-Aktionen, Blau Slave-Aktionen.

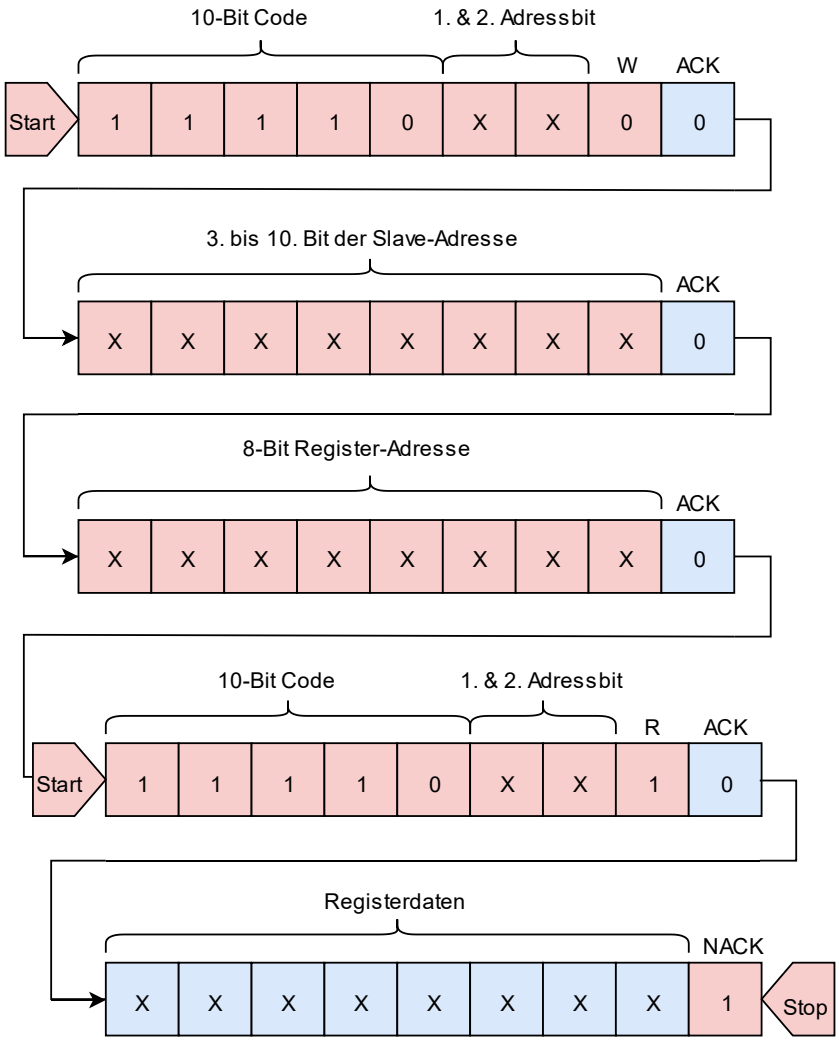

Abbildung 12: Das Auslesen von Daten eines Slave-Registers durch einen Master. Rot: Master-Aktionen, Blau: Slave-Aktionen.

3 CAN Protokoll

Das CAN-Protokoll ist ein Busprotokoll, welches 1983 von Bosch entwickelt wurde. Daten werden nach dem „Multi-Master-Prinzip“ zwischen den gleichwertigen CAN-Instanzen seriell übertragen. Das CAN-Protokoll ist in der internationalen Norm ISO 11898 beschrieben. Im Gegensatz zu anderen seriellen Busprotokollen wie I2C, existiert keine Clock-Leitung und die Busteilnehmer synchronisieren sich allein über die Datenleitungen. Über zwei Datenleitungen werden dieselben Informationen gleichzeitig differentiell übertragen. Datenpakete werden in sogenannten „Frames“, z. D. „Rahmen“ versendet. Hierbei wird kein bestimmter Teilnehmer adressiert, stattdessen werden die identifizierbaren Nachrichten von allen Teilnehmern empfangen und können je nach Relevanz verarbeitet oder ignoriert werden. In diesem Projekt wird der High-Speed CAN-Bus genutzt. Die CAN-Spezifikation ist sehr umfangreich und kann in dieser Arbeit nur oberflächlich behandelt werden.

3.1 Elektrische Spezifikation

Der CAN-Bus enthält die zwei Datenleitungen „CAN-High“ und „CAN-Low“. Auf beiden Leitungen werden dieselben Daten gleichzeitig übertragen. Hierbei werden einzelne Bits anhand der Spanungsdifferenz V_{diff} der beiden Leitungen bestimmt.

Es gilt:

$$V_{diff} = V_{CANH} - V_{CANL} \tag{3}$$

Es wird zwischen einem dominanten und einem rezessiven Zustand unterschieden. Im dominanten Zustand beträgt die Spannung V_{diff} nominal 2 V und entspricht einer logischen 0. Im rezessiven Zustand beträgt V_{diff} nominal 0 V und entspricht einer logischen 1. V_{CANH} und V_{CANL} haben im dominanten Zustand einen nominalen Wert von 3,5 bzw. 1,5 V und im rezessiven jeweils 2,5 V. Die Teilnehmer am CAN-Bus sind quasi über eine Wired-And-Verknüpfung gekoppelt. Erzeugt ein Teilnehmer den dominanten Zustand auf dem Bus und damit eine 0, so überschreibt er immer den rezessiven Zustand. Beide Leitungen werden am Ende jeweils mit einem Abschlusswiderstand von 120 Ω terminiert. Diese wirken einmal für CAN-High als Pull-Down- und für CAN-Low als Pull-Up-Widerstände und erzeugen damit den rezessiven Zustand. Darüber hinaus unterdrückt der Abschlusswiderstand Reflexionen an den Leitungsenden des Busses.

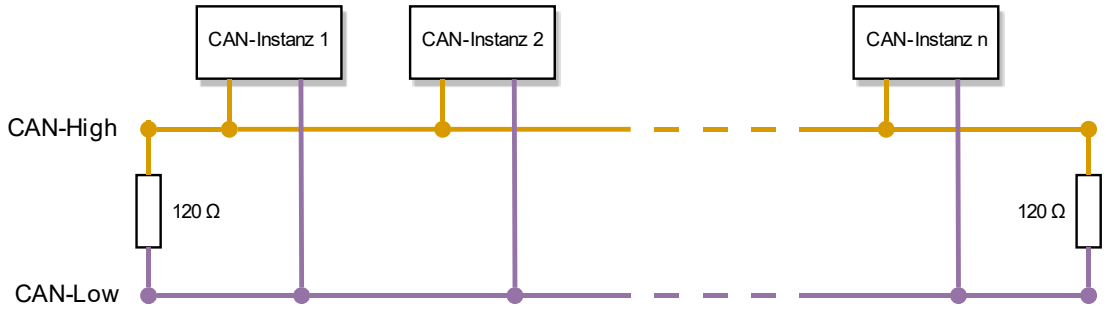

Abbildung 13: Beispiel eines CAN-Bus Systems.

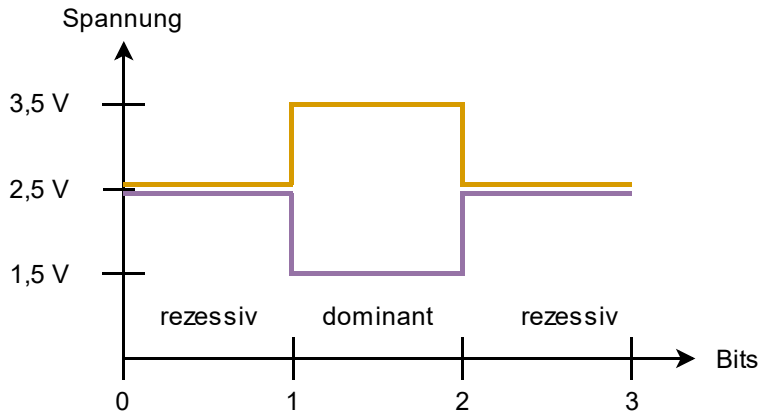

Abbildung 14: Signalpegel bei der CAN-Kommunikation. CAN-H in orange und CAN-L in violett.

3.2 Datenübertragung

Daten werden gemäß des CAN-Protokoll in bestimmten Rahmen bzw. „Frames“ versendet.

Es gibt vier Frames, die in unterschiedlichen Situationen genutzt werden. Die Frames sind in folgende Funktionalitäten aufgeteilt:

1. Das Data Frame übermittelt Daten eines Senders an alle Teilnehmer.
2. Ein von einem Teilnehmer versendetes Remote Frame, fordert ein Data Frame an, welches den selben Identifier besitzt. Ein Teilnehmer, der diese Information besitzt, kann daraufhin auf das Remote Frame mit einem Data Frame antworten.
3. Das Error Frame kann von jedem Teilnehmer versendet werden, sobald ein Fehlerfall auftritt. Durch ein Error Frame wird jeder über den Fehler in Kenntnis gesetzt.
4. Das Overload Frame dient dazu, die Kommunikation zwischen den Frames bei Bedarf zu pausieren. Dies kann dann erforderlich sein, wenn die Datenverarbeitung von Teilnehmern andauert. Aufgrund der geringen Relevanz im behandelten Anwendungsfall, wird dieses Frame nicht weiter behandelt.

3.2.1 Data Frame

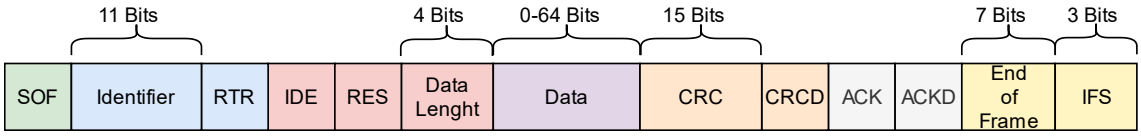

Abbildung 15: Standardformat des Data Frames.

In der Abbildung 15 ist das Standardformat des Data Frames zu erkennen. Es besteht aus dem Start of Frame (grün), Arbitration Field (blau), Control Field (rot), Data Field (violet), CRC Field (orange), ACK Field (grau) und aus dem End of Frame und Inter Frame Space (gelb).

- Das Start of Frame (SOF) signalisiert allen Teilnehmern den Start eines Frames. Ein SOF wird mit einem dominanten Bit eingeleitet.
- Über den Identifier können Teilnehmer die Datenpakete eindeutig identifizieren. Ebenfalls wird über den Identifier die Arbitration durchgeführt.
- Das RTR-Bit (Remote Transmission Request) kennzeichnet den Rahmentyp. Ist das Bit gesetzt, so handelt es sich um ein Remot Frame, andernfalls um ein Data Frame.
- Das IDE-Bit (Identifier-Extension) gibt den Identifier-Typ an. Bei gesetztem IDE-Bit (rezessiver Zustand) handelt es sich um ein Data Frame oder ein Remote Frame im Extended Frame Format mit einem Identifier von 29 Bit. Bei nicht gesetztem Bit (dominanter Zustand) handelt es sich um ein Frame im Base bzw. Standard Frame Format mit einem Identifier von 11 Bit.
- Als nächstes folgt ein reserviertes Bit (RES), welches über keine Funktion verfügt und vom Sender 0 gesetzt werden sollte.
- Data Length bzw. Data Length Code (DLC) beinhaltet die Menge der 8-Bit Datenpakete. Data Length kann einen Wert von 0x00 bis 0x08 haben.
- Data beinhaltet die eigentlichen Daten. Es können pro Frame maximal 8 Byte versendet werden. In der Erweiterung des CAN-Protokolls „CAN FD“ (CAN with Flexible Data-Rate), welches ebenfalls in der ISO 11898-1 spezifiziert ist, ist es möglich, bis zu 64 Byte an Daten pro Frame zu versenden. Diese Erweiterung wird in dieser Arbeit nicht weiter behandelt.

- CRC steht für „cyclic redundancy check“. Es handelt sich um ein Verfahren, mit dem die Validität von Daten überprüft werden kann. Die Verarbeitung von fehlerhaften Daten kann so verhindert werden. Die CRC-Sequenz im Frame ist eine Prüfsumme, die vom Sender berechnet wurde. Die Berechnung wird mittels eine binäre Polynomdivision durchgeführt. Der Dividend besteht aus allen Bits des Frames beginnend mit dem SOF bis zum letzten Datenbit aber ohne Stuffing. Zusätzlich werden 15 Bits mit dem Wert 0, in das CRC-Register eingeschoben. Nun wird die Division mit einem Divisor in Form eines Generatorpolynomes durchgeführt. Dieses Polynom ist in der ISO Norm 11898-1 vordefiniert als $x^{15}+x^{14}+x^{10}+x^{8}+x^{7}+x^{4}+x^{3}+1$. Die Division ergibt in den meisten Fällen einen unteilbarer Rest „x“, dessen niederwertigsten fünfzehn Bit die CRC-Prüfsumme bilden. Der Empfänger des Frames muss die CRC-Prüfsumme daraufhin selbst berechnen. Stimmen die empfangene und die selbstberechnete Prüfsumme nicht überein, so muss ein Error Frame versendet werden, um dem Sender den Fehlerfall mitzuteilen und so die Daten erneut zu erhalten.
- Die Delimiter Bits CRCD und ACKD dienen vorwiegend der Überprüfung des Frames im Form-Check. Beide Bits müssen rezessiv sein.
- Das ACK dient der Überprüfung des korrekten Empfanges. Alle Empfänger der Nachricht müssen mit einem ACK bestätigen, dass sie die Nachricht korrekt empfangen haben. Hierzu reicht es aus, wenn mindestens ein Empfänger ein dominantes Bit erzeugt. Ein ausbleibendes ACK-Bit ergibt einen ACK-Fehler.
- Das End of Frame (EOF) ist eine Bitfolge, welche das Ende des Frames signalisiert. Hierzu werden sieben rezessive Bits übertragen. Wird das EOF nicht korrekt übertragen, so handelt es sich um einen Formfehler.
- Das Inter Frame Space (ISF) besteht aus mindestens 3 rezessiven Bits, die Frames von einander trennen. Während dieses Bereiches kann ein Overload Frame versendet werden.

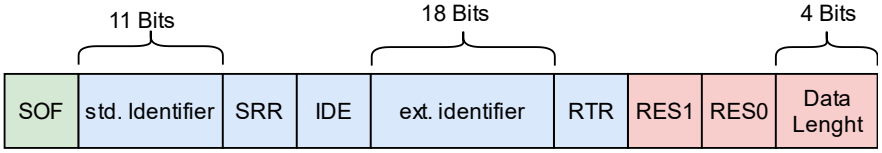

Abbildung 16: Extended Frame von SOF bis zu Data Length.

In der Abbildung 16 ist der Aufbau des Extended Frames zu erkennen. Im Gegensatz zum Standardformat aus Abbildung 15 wird anstatt des RTR-Bits das Substitute Remote

Request Bit (SRR) verwendet. Dieses muss rezessiv sein und sorgt in Kombination mit dem darauffolgenden ebenfalls rezessiven IDE Bit dafür, dass Nachrichten im Standard Frame eine höhere Priorität haben als Nachrichten im Extended Frame.

Danach folgen die restlichen 18 Identifier Bits, das RTR Bit, zwei reservierte Bits und die Datenlänge.

3.2.2 Remote Frame

Das Remote Frame dient dazu, Daten von anderen Teilnehmern anzufordern. Hierzu wird im Prinzip das gleiche Frame wie in Abbildung 15 verwendet, jedoch ohne Nutzung des „Data“ Feldes. Das Remoteframe enthält den Identifier der gewünschten Nachricht. Empfängt der Teilnehmer, welcher dem angefragten Identifier zugeordnet worden ist, die Nachricht, so sendet dieser im Idealfall direkt danach das gewünschte Data Frame. Das Extended Remote Frame hat in Bezug auf das Arbitrations- und Kontrollfeld, denselben Aufbau wie in Abbildung 16.

3.2.3 Error Frame

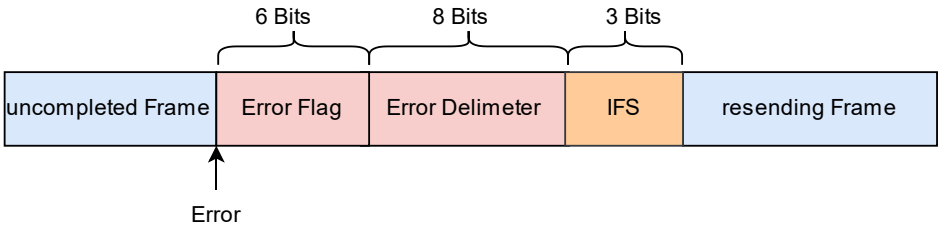

Abbildung 17: Auftritt eines Errors mit darauffolgendem Error Frame.

In Abbildung 17 ist ein während der Kommunikation auftretendes Error Frame (rot) dargestellt. Es besteht aus dem Error Flag und dem Error Delimeter. Error Flags können ein „Passive Error Flag“ oder „Active Error Flag“ sein. Ersteres beinhaltet 6 rezessive, zweiteres 6 dominante Bits. Darauf folgt ein Error Delimeter Feld mit 8 rezessiven Bits. Nach dem Inter Frame Space, wird vom Sender ein erneuter Übertragungsversuch gestartet. Setzt ein Teilnehmer ein aktives Error Frame, wird dies von den anderen erkannt.

3.2.4 Fehlererkennung

Das CAN-Protokoll hebt sich von anderen Protokollen mit seiner weitreichenden Fehlerbehandlung ab. Die folgenden Mechanismen zur Fehlererkennung werden bereitgestellt:

1. Bitmonitoring
2. Form Check bzw. Frame Check
3. Stuff Check
4. ACK Check
5. Cyclic Redundancy Check

1. Beim Bitmonitoring vergleicht der Sender einer Nachricht, die auf den Bus gelegten Bits, mit denen, die tatsächlich auf dem Bus anliegen. Stimmen diese nicht überein, so handelt es sich um einen Bit Error. Während des Arbitrationsprozesses werden keine Fehler ausgelöst.
2. Durch den Form Check wird die Korrektheit der Frame-Formats überprüft. Ein Form Error liegt dann vor, wenn z.B. die Delimeter Bits oder das EOF mindestens einen dominanten Wert aufweisen.
3. Im Stuff Check werden die Stuff Bits überprüft. Das CAN-Protokoll schreibt vor, dass nach fünf gleichwertigen Bits, ein reziprokes Bit gesetzt werden muss. Werden in einer Nachricht sechs gleichwertige Bits sequenziell übertragen, so löst dies einen Stuff Error aus.
4. Mit dem ACK Check wird vom Sender überprüft, ob die Nachricht empfangen wurde. Hierzu prüft der Sender, ob ein Empfänger, ein dominantes Bit an die Stelle des ACK Feldes gesetzt hat.
5. Beim CRC Check wird die empfangene Prüfsumme mit der berechneten verglichen. Dies wurde im Kapitel Data Frame bereits erläutert. Die angeschlossenen CAN-Instanzen verfügen über spezielle Zähler, die jeweils Fehler beim Senden und Empfangen von Nachrichten mitzählen. Die Fehlerzähler besitzen die Bezeichnung Transmit Error Counter (TEC) und Receive Error Counter (REC). Je nach Fehlerart und Betriebszustand, werden durch Fehler die Zähler mit unterschiedlicher Gewichtung hochgezählt und durch fehlerfreie Abläufe heruntergezählt. Bis zu einem Zählwert von 127, befindet sich eine CAN Instanz im Betriebszustand "Bus Active" und kann Active Error Frames versenden. Ab einem Zählerstand von 128, wechselt die betreffende CAN Instanz in den Betriebszustand Bus Passive und kann nur noch Passive Error Frames

versenden, die keinen Einfluss mehr auf die Kommunikation von anderen Teilnehmern haben. Ab einem Zählerstand von 256, geht die CAN Instanz in den Betriebszustand Bus Off über und entkoppelt sich selbst vom Bus.

3.2.5 Arbitrierung der Buszugriffe

Durch die im CAN-Bus eingesetzte bitweise Arbitrierung, setzt sich bei parallelen Buszugriffen, nur die Nachricht durch, welche die höchste Priorität hat. Der Arbitrationsbereich ist in der Abbildung 15 und in der Abbildung 16 in blau gekennzeichnet. Diese beginnt mit der Übertragung des Identifier. Die CAN-Instanzen legen den Identifier Bit für Bit auf den Bus. Gleichzeit wird der aktuell anliegende Wert am Bus überwacht. Eine sendende CAN-Instanz wechselt vom Sender- in den Empfängerzustand, sobald ein von ihr gesetztes rezessives Bit, durch ein dominantes Bit überschrieben wurde. Demzufolge haben niederwertige Identifier einen Vorrang gegenüber höherwertigen. Das gleiche Prinzip gilt für Data Frames im Vergleich zu Remote Frames und Standard Indentifier gegenüber Extended Identfier. In der Abbildung 18 ist eine bitweise Arbitration dargestellt. Die Nachricht, die von Teilnehmer 2 versendet wird, hat aufgrund der führenden dominanten Bits im Identifier eine höhere Priorität und setzt sich so gegenüber den anderen Nachrichten durch. Nach dem Verlust des Buszugriffs, gehen die anderen Sender in den Empfangsmodus über.

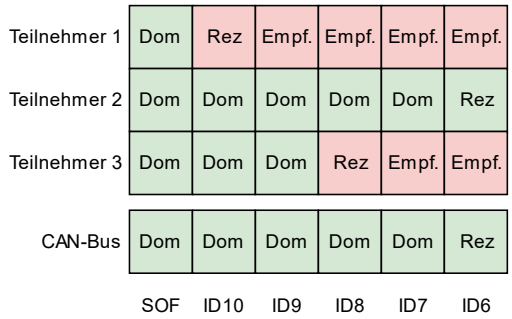

Abbildung 18: CAN-Arbitration. Rote Felder kennzeichnen den Verlust des Buszugriffes.

3.2.6 Synchronisation

1. Entsprechend des CAN-Protokolls findet keine übergeordnete Synchronisation aller am selben Bus angeschlossenen CAN-Teilnehmer statt. Es wird vorausgesetzt, dass alle Teilnehmer Daten mit der gleichen Bitrate senden und empfangen. Hierbei ist die „Bitzeit“ in mehrere Zeitsegmente aufgeteilt, über deren Konfiguration die Bitrate eingestellt werden kann. Die Bitzeit besteht aus dem „Synchronisationssegment“, „Propagationssegment“ Phasesegment 1 und 2. Dieses Thema wurde bereits ausreichend dokumentiert. [3]

4 I2C Slave

Das I2C Slave wurde von einem Studenten der FH Köln für einen ASIC entwickeltund hatte die Aufgabe Digital-Analog-Umsetzer anzusteuern. Der Slave wurde in Verilog geschrieben und teilt sich in viele Untermodule auf. Einige von denen, werden hier ausführlicher behandelt, da das Verständnis der Funktionalität des I2C-Slaves für die Konzipierung der Bridge von hoher Bedeutung ist. [4]

4.1 Topmodule

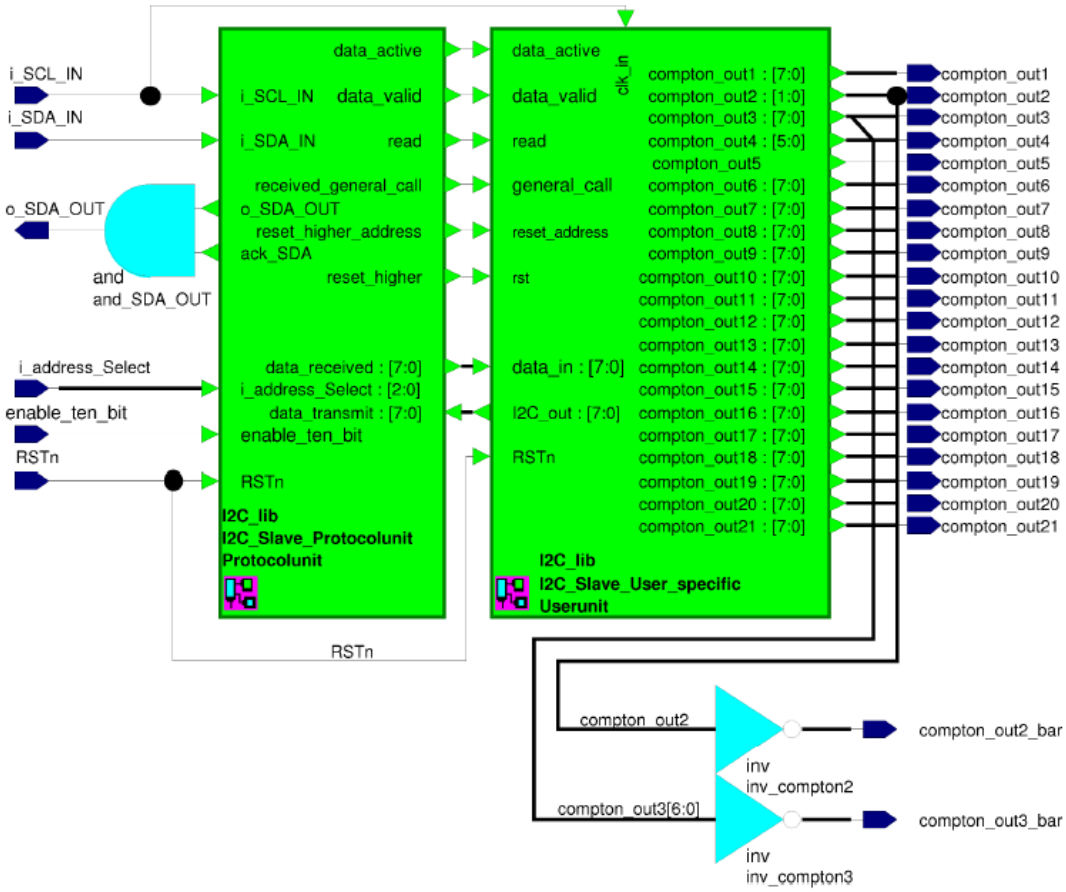

Abbildung 19: I²C-Slave mit den ersten zwei Untermodulen. Protocolunit (links). [4]

Das Topmodule beinhaltet zwei Untermodule und zwar die „Protocolunit“, welche die Kommunikation nach dem I2C-Protokoll realisiert und eine „Userunit“, in der die Register verwaltet werden. Es gibt fünf Eingänge:

1. „i_SCL_IN“, der Eingang für die Taktleitung SCL.
2. „I_SDA_IN“, der Eingang für die Datenleitung SDA.
3. „RSTn“, low-aktiver Rücksetzeingang des Slaves.

4. „enable_ten_bit“, bei dem Zustand HIGH muss das Slave über seine 10-Bit Adresse angesprochen werden, bei LOW über die 7-Bit Adresse.
5. „i_adress_Select“, ein Bus über den die letzten 3 Bits der 7- bzw. 10 Bit Adresse extern eingestellt werden können.

Die Ausgänge sind der Ausgangsport für SDA „o_SDA_OUT“ sowie die Slaveregister mit den Bezeichnungen „compton_out“.

4.2 Protocolunit

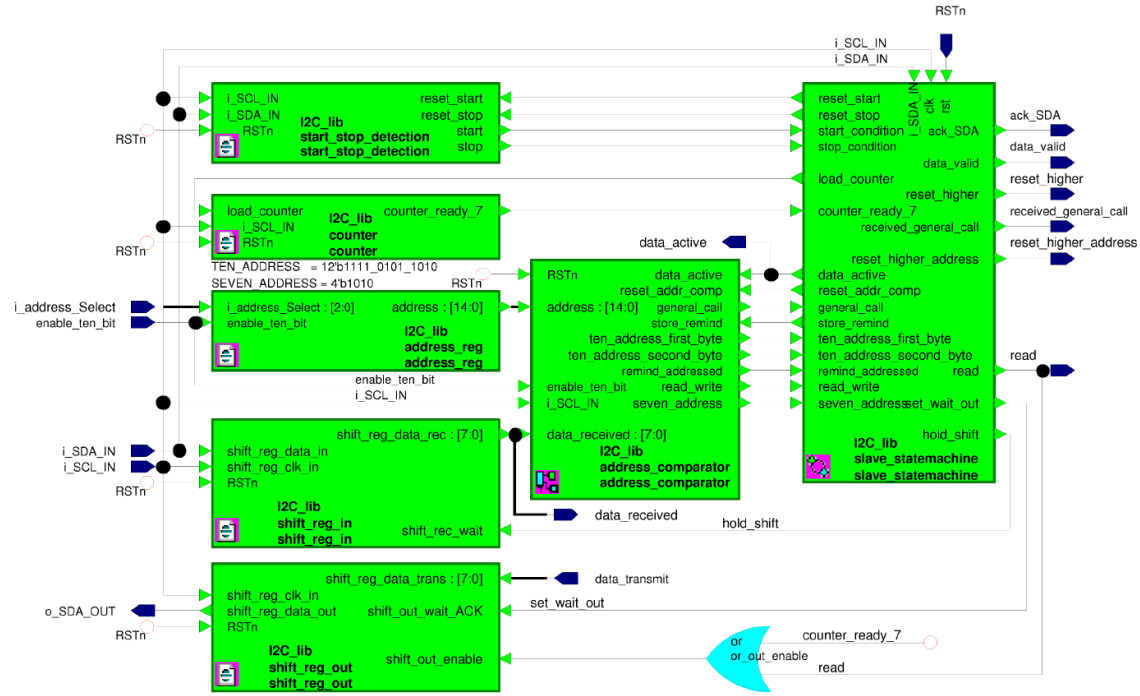

Abbildung 20: Protocolunit des Slaves mit den Untermodulen. [4]

Die Protocolunit teilt sich in folgende Module auf:

1. „start_stop_detection“ tastet SCL mit SDA ab, um eine Start- bzw. Stoppbedingungen zu erkennen. Dabei setzt sie für den jeweiligen Fall das Signal start oder stop. Hat die slave_statemachine das Signal verarbeitet wird das Modul über die Signale reset_start bzw. reset_stop zurückgesetzt.

2. „counter“ dient dazu, zu erfassen, wann ein ganzes Byte an Daten übermittelt wurde, um z.B. genau zu bestimmen, wann ein ACK gesetzt werden soll. Über „counter_ready_7“ wird dies der Zustandsmaschine mitgeteilt. Durch „load_counter“ wird dieser Zähler von der Zustandsmaschine zurückgesetzt.

3. „address_reg“ speichert die interne Adresse ab. Die Eingänge sind bereits beschrieben. Die Adresse wird danach dem Modul „address_comparator“ übergeben.

4. „address_comparator“ enthält mehrere Komparatoren, die die in „shift_reg_in“ gespeicherten Daten über „data_received“, mit den vorgegebenen Werten oder Signalen (z.B. das Signal „address“) vergleichen.
    - Über „data_active“, wird der Adresskomparator deaktiviert, wenn Daten und keine Adressen erwartet werden.
    - Das Signal „seven_address“ wird gesetzt, wenn der Komparator sein 7-Bit Adresse (die letzten 7. Bits des Signals „address“) erkannt hat.
    - Read_write leitet das erkannte R/W-Bit an die Zustandsmaschine weiter.
    - „ten_address_first_byte“ und „ten_address_second_byte“ werden immer dann gesetzt, wenn jeweils das erste oder das zweite Adressbyte der 10-Bit-Slave-Adresse erkannt wird.
    - „general_call“ wird gesetzt, falls dieser eine I2C-Slave-Adresse empfangen wird, die nur aus Nullen besteht.
    - „store_remind“ wird von der Zustandsmaschine gesetzt, damit eine Speicherung der Adresse erfolgt, sodass auch ein 10-Bit Lesezugriff erfolgen kann.
    - „remind_addressed“ liegt an ein Register an, welches von „store_remind gesetzt oder von „reset_addr_comp“ zurückgesetzt wird.
    - „reset_addr_comp“ wird von der Zustandsmaschine gesetzt, um eine vorher gespeicherte 10-Bit Adresse zurückzusetzen.

5. „shift_reg_in“ speichert die Bits, die an SDA bei jeder steigenden Flanke anliegen. Wenn das Slave sendet und nicht empfängt, wird das Modul über „shift_rec_wait“ pausiert. Der Inhalt wird über „data_received“ ausgegeben.

6. „shift_reg_out“ dient dazu, die von der Userunit kommenden Daten des Signals „data_transmit“ nach jeder fallenden Flanke von SCL auf SDA zu legen.

7. „slave_statemachine“ ist die Zustandsmaschine, die den kompletten Ablauf innerhalb der Protocolunit steuert und weitere Steuersignale für die Userunit

erzeugt. Hierzu sind weitere Signale zu erwähnen, die noch nicht beschrieben wurden.

- „ack_SDA“ ist das ACK-Signal, welches in der Zustandsmaschine erzeugt und mit o_SDA_Out mit einer logischen UND-Operation verknüpft wird. Zusammen bilden beide den Ausgangsport des Slaves.
- Mit „data_valid“ signalisiert die Zustandsmaschine der Protocolunit der Userunit, dass das auf „data_received“ bzw. „data_in“ anliegende 8-Bit Datenpaket valide ist und damit verarbeitet werden kann.
- Über „reset_higher“ wird die Zustandsmaschine der userunit zurückgesetzt. Dies ist immer dann der Fall, wenn eine Start-, Stopp-, Start-Stoppbedingung (Start und Stopp gleichzeitig) erkannt wird oder aktuell keine Datenübertragung stattfindet und auf eine Startbedingung gewartet wird.
- „received_general_call“ wird gesetzt, sobald ein „general call“ erfasst wurde. Darauf folgt das Rücksetzen aller Register.
- „reset_higher_adress“ setzt die in der Usereinheit gespeicherten Registeradresse auf 0x00 zurück. Dies ist bei einer Stopp- oder Start-Stoppbedingung oder wenn keine Datenübertragung stattfindet der Fall.
- „read“ wird gesetzt, sobald ein Leseanfrage erkannt wird und die Zustandsmaschine der Protocolunit in die entsprechenden „Lese-Zustände“ wechselt.

4.3 Userunit

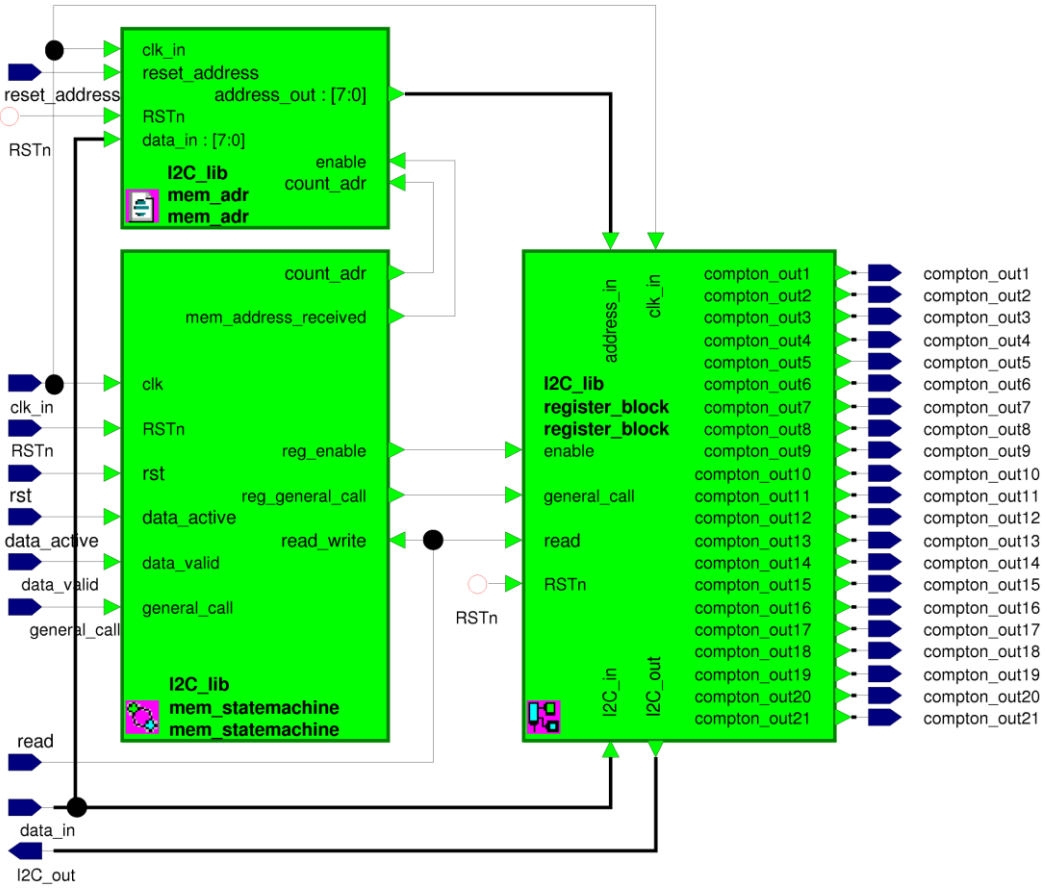

Abbildung 21: Userunit des I2C-Slaves mit den Untermodulen. [4]

Die Userunit besteht aus folgenden Modulen:

1. „mem_statemachine“ steuert den Ablauf der Speicherung sowie die Ausgabe von Daten. Hierzu sind folgende Signale zu nennen:
    - „rst“ entspicht dem im vorherigen Kapitel erwähnten Signal „reset_higher“. Die Zustandsmaschine geht beim gesetzten Signal in den Ruhezustand über.
    - "data_active“ sorgt an dieser Stelle dafür, dass die Zustandsmaschine ihren Ruhezustand erst verlassen kann, wenn nach korrekter Adressübertragung Daten als Nächstes zu erwarten sind.
    - „mem_address_received“ wird immer dann aktiviert, wenn das erste 8-Bit Datenpaket empfangen wurde. Dieses beinhaltet die Adresse des Registers, welches angesteuert werden soll. Sobald „mem_address_received“ gesetzt wird, speichert das Modul „mem_adr“ die an „data_in“ anliegende Adresse. Das Signal „data_in“ entspricht dem Ausgang „data_received“ des Eingangsregisters „shift_reg_out“.

- „count_adr“ wird durch die Zustandsmaschine gesetzt, um die Registeradresse zu inkrementieren, damit ein fließendes Auslesen bzw. Beschreiben des Registers möglich ist.
- „reg_enable“ aktiviert jeweils das Register, welches durch das Modul „mem_adr“ adressiert wird. Je nachdem ob das Signal „read“ gesetzt ist, wird das Register mit den Daten die an „data_in“ anliegen beschrieben bzw. der Inhalt des Registers an „I2C_out“ gelegt.

2. „mem_adr“ dient zur Speicherung der Empfangen Registeradressen. Folgende Signale sind zu nennen:
    - „reset_adress“ entspricht dem im vorherigen Kapitel erwähntem Signal „reset_higher_adress“. Es setzt die Registeradresse zurück.
    - Die an „data_in“ anliegenden Daten werden dann, wenn die mem_statemachine das empfangene Datenpaket als Registeradresse identifiziert hat, im Modul abgespeichert.
    - „address_out“ gibt die aktuell gespeicherte Registeradresse aus.
    - „enable“ entspricht dem vorher erläutertem Signal „mem_address_received“.

3. „register_block“ beinhaltet alle Register sowie einen Multiplexer der den Inhalt des jeweils adressierten Registers ausgibt.
    - „address_in“ entspricht dem vorher erwähnten „address_out”. Dieses Signal liegt an jedem Register an. Bei jedem Zugriff wird die Adresse, die dem Register zugewiesen ist, mit der Adresse die an „address_in“ anliegt verglichen.
    - Über „enable“ wird dem Modul „register_block“ ein valider Zugriff signalisiert. Sobald das Signal gesetzt wird, reagiert das Register, welches seine Adresse an „address_in“ erkennt. Wenn das Signal „read“ gesetzt ist, wird der Inhalt des ausgewählten Registers an „I2C_out“ gelegt. Falls „read“ nicht gesetzt ist, werden die an „I2C_in“ anliegenden Daten in das Register geschrieben. An „I2C_in“ liegen die aktuellen Daten des Eingangsregisters aus der Protocolunit an („data_received“).

5 CAN-Controller

Der CAN-Controller „Canakari“, wurde im Jahre 2000 im Rahmen einer Diplomarbeit an der Fachhochschule Köln in VHDL entwickelt. [5] Im Laufe der Jahre wurden Erweiterungen, durch Studierende der FH Köln und der FH Dortmund, dem Design hinzugefügt. [6] [7] [8] [9] In dieser Arbeit wird die Verilog Übersetzung des Controllers genutzt. Aufgrund der Komplexität des CAN-Controllers werden hier nur relevante Bereiche behandelt.

5.1 Nutzer-Schnittstelle

Über die Nutzer-Schnittstelle lässt sich der CAN-Controller vollständig steuern. Ursprünglich für die Kommunikation mit Microkontrollern konzipiert, soll die Schnittstelle nun über einen I2C-Master angesteuert werden können. Über die Schnittstelle werden Register beschrieben, mit denen sich Einstellungen konfigurieren und Sendevorgänge auslösen lassen. Ebenfalls sind Empfangsinformationen in diesen Registern gespeichert und können über die Schnittstelle abgerufen werden.

Die Schnittstelle beinhaltet die Ports aus der unteren Tabelle.

Tabelle 2: Die Canakari Schnittstelle [9]

Name	**Richtung**	**Breite**	**Funktion**
address	IN	5	Übermittelt die Adresse des Registers, auf dem ein Lese- oder Schreibzugriff erfolgen soll.
readdata	OUT	16	Bei einem Lesezugriff werden die Registerdaten auf readdata gelegt und können dann ausgelesen werden.
writedata	IN	16	Die an writedata anliegenden Daten, werden bei einem Schreibzugriff, in das Register geschrieben.
chipselect	IN	1	chipselect startet den Zugriff.
read_n	IN	1	Aktiviert den Lesezugriff, sodass die Registerdaten auf „readdata“ gelegt werden.
write_n	IN	1	Aktiviert den Schreibzugriff, sodass die an writedata anliegenden Daten in das Register geschrieben werden.

Die Abläufe des Schreib- und Lesezugriffes sind in den unteren Abbildungen dargestellt. Bei einem Schreibzugriff werden direkt nach der fallenden Flanke, die Signale address, writedata, chipselect und write_n (low-aktiv) gesetzt. Der CAN-Controller wird dann die Schreibdaten in das Register übernehmen, welches die gleiche Adresse aufweist, wie die anliegende. Ursprünglich wurde chipselect erst gesetzt, nachdem die anderen Daten bereits gesetzt waren. Dies wurde in der Bridge jedoch nicht implementiert. Das gleichzeitige Setzen hat jedoch keine negative Auswirkung, da sowohl der CAN-Controller als auch die Bridge, die Daten erst mit der nächsten steigenden Flanke endgültig übernehmen. Beim Lesezugriff muss, statt dem write_n, das ebenfalls low-aktive read_n Signal gesetzt werden. Die vom CAN-Controller auf readdata gelegten Daten, werden von der Bridge bei der nächsten steigenden Taktflanke übernommen.

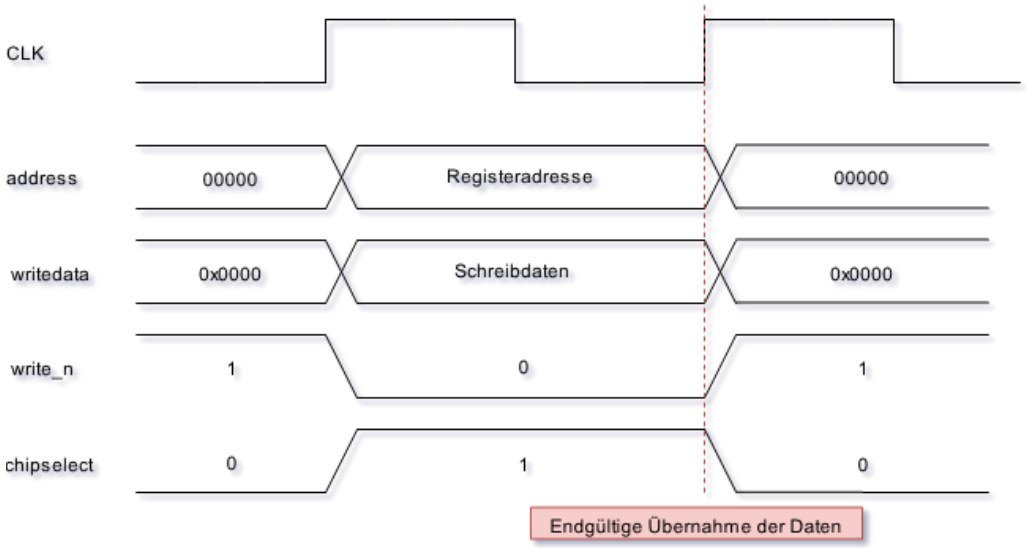

Abbildung 22: Schreibzugriff der Canakari-Schnittstelle.

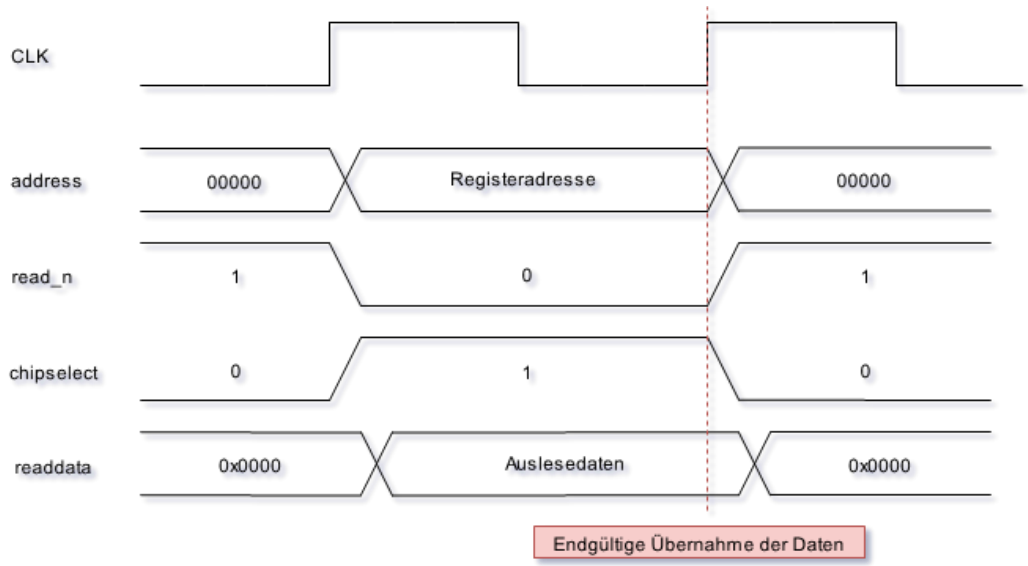

Abbildung 23: Lesezugriff der Canakari-Schnittstelle.

5.2 Canakari Register

Tabelle 3: Canakari Register [10]

Addr (Hex)	Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Comment
0	Receive Data 7-8	data 7								data 8								
1	Receive Data 5-6	data 5								data 6								
2	Receive Data 3-4	data 3								data 4								
3	Receive Data 1-2	data 1								data 2								
4	Receive Identifier 2	ID[12:0]											reserved					Extended identifier
5	Receive Identifier 1	ID[28:13]											reserved					Extended identifier
6	Receive Identifier 1	ID[10:0]											reserved					Standard identifier
7	Receive Control	ovlindc	recindc	reserved				ienable	reserved			remote	extended	DLC[3:0]				
8	Transmission Data 6-7	data 7								data 8								
9	Transmission Data 5-6	data 5								data 6								
A	Transmission Data 3-4	data 3								data 4								
B	Transmission Data 1-2	data 1								data 2								
C	Transmission Identifier 2	ID[12:0]											reserved					Extended identifier
C	Transmission Identifier 1	ID[28:13]											reserved					Extended identifier
D	Transmission Identifier 1	ID[10:0]											reserved					Standard identifier
D	Transmission Control	treg	tindic	reserved				ienable	reserved			remote	extended	DLC[3:0]				
E	General	Busoff	ErrActiv	ErrPasv	Warning	SucSend	SucRec	Reset	sjw[2:0]				tseg1[2:0]		tseg2[2:0]			
F	Prescaler	reserved							High[3:0]						Low[3:0]			
10	Acceptionmask2	ID[12:0]														reserved		Extended identifier
11	Acceptionmask1	ID[28:13]																Extended identifier
11	Acceptionmask1	ID[10:0]											reserved					Standard identifier
12	Interrupt	TX enable	reserved						Status En	Suctra En			Sucrec En	reserved	IRQ Status	IRQ Suctra	IRQ Sucrec	

Addr (Hex)	Name	Bits	Description
6	Receive Control	ovflindc	informiert die CPU darüber, daß eine Botschaft empfangen und die Register abgespeichert worden ist, bevor die alte Nachricht aus den Registern gelesen worden ist.
		recindc	deutet darauf hin, das seit dem letzten Rücksetzen dieses Bits durch die CPU mindestens eine Botschaft erfolgreich empfangen worden ist.
		ienable	entscheidet darüber, ob ein Interrupt abgesetzt werden soll, wenn eine Nachricht erfolgreich empfangen worden ist. Wird noch nicht unterstützt.
		remote	spezifizieren den Rahmentyp
		extended	
		DLC[3:0]	beinhalten die Empfangene data length code Information
D	Transmission Control	treg	wird durch die CPU gesetzt, um die Übermittlung der Daten im Sendearbitrierungs- und Sendedatenregister einzuleiten
		tindic	deutet darauf hin, das seit dem letzten Rücksetzen dieses Bits durch die CPU mindestens eine Botschaft erfolgreich gesendet worden ist.
		ienable	entscheidet darüber, ob ein Interrupt abgesetzt werden soll, wenn eine Nachricht erfolgreich gesendet worden ist. Wird noch nicht unterstützt.
		remote	spezifizieren den Rahmentyp
		extended	
		DLC[3:0]	beinhalten die Empfangene data length code Information
E	General	Busoff ErrActive ErrPasv	Diese Bits geben Auskunft über den aktuellen Fehlerzustand des Controllers. Diese Signale werden durch die Fault Confinement Unit in jedem Taktzyklus aktualisiert.
		Warning	informiert die von der CPU ausgeführte Applikation das einer der Fehlerzähler den Stand von 96 erreicht hat.
		SucSend	deutet darauf hin, das seit dem letzten Rücksetzen dieses Bits durch die CPU mindestens eine Botschaft erfolgreich gesendet worden ist.
		SucRec	deutet darauf hin, das seit dem letzten Rücksetzen dieses Bits durch die CPU mindestens eine Botschaft erfolgreich empfangen worden ist.
		Reset	zwingt den Controller einen Reset bzw. Initialisierungsvorgang auszulösen.
		sjw	bestimmt die maximale Sprungweite, die während eines Resynchronisationsvorgangs ausgeführt werden kann
		tseg1	ist das Zeitsegment, das vor dem Samplezeitpunkt liegt
		tseg1	ist das Zeitsegment, das nach dem Samplezeitpunkt liegt
F	Prescaler	High	Anzahl clock für High-time. sampling frequency = f_sys/(high+low). Baudrate = f_sample/10
		Low	Anzahl cloks für low-time
12	Interrupt	TX enable	aktiviert den physical layer für das senden
		Status En	Enable des Interrupts bei einem Statuswechsel
		Suctra En	Enable des Interrupts bei erfolgreichem senden
		Sucrec En	Enable des Interrupts bei erfolgreichem empfangen
		IRQ Status	Interrupt flag für Statuswechsel
		IRQ Suctra	Interrupt Flag für erfolgreiches Senden
		IRQ Sucrec	Interrupt Flag frü erfolgreiches Empfangen

6 Bridge

Um die Register des Canakari über I²C beschreiben zu können, ist ein Design gefordert, welches I2C-Daten aufnehmen und versendet konnte, wobei die Kommunikation nicht wie üblich mit slave-internen Registern stattfindet, sondern mit denen des CAN-Controllers, auf die wiederum nur über ein weiteres Datenprotokoll zugegriffen werden kann. Dieses Problem wird über eine Bridge gelöst. Die Bridge koppelt das I2C Slave und den CAN-Controller, sodass die Register des CAN-Controllers über einen I2C-Master beschrieben und ausgelesen werden können. Über die Register lässt sich wiederum der CAN-Controller steuern, sodass sich CAN-Nachrichten über I2C versenden und empfangen lassen. Hierzu mussten zum einen das Slave und seine Signale genau studiert werden. Zum anderen musste mit Letzterem das Protokoll der Schnittstelle realisiert und ein logisches Konzept für die Ansteuerung der Schnittstelle über I2C gefunden werden.

6.1 Steuersignale der Bridge

Im Zuge eines vorherigen Debuggings wurde der unter Vivado verfügbare „Intergrated Logic Analyzer“ (ILA) ebenfalls für die Analyse der Slave-Signale genutzt. Es wurden einige Signale untersucht und ein Konzept für eine mögliche Bridge erstellt. In den folgenden Kapiteln werden die Signale zur Steuerung der Bridge erläutert.

6.1.1 reg_enable & read

„reg_enable“ diente im Registerblock der Userunit zur Aktivierung der Registeradressen. In der Abbildung 24 ist reg_enable gezeigt, während I2C-seitig geschrieben wird. Es wird immer dann aktiviert, wenn nach der Übermittlung der Registeradresse weitere valide Daten versendet werden. Mit der erfolgreichen Abfrage des ACKs wird reg_enable gesetzt und mit der ersten steigenden Flanke des nächsten Bytepaketes zurückgesetzt. Das Signal reg_enable kann damit als Steuersignal zur Beschreibung der Register verwendet werden.

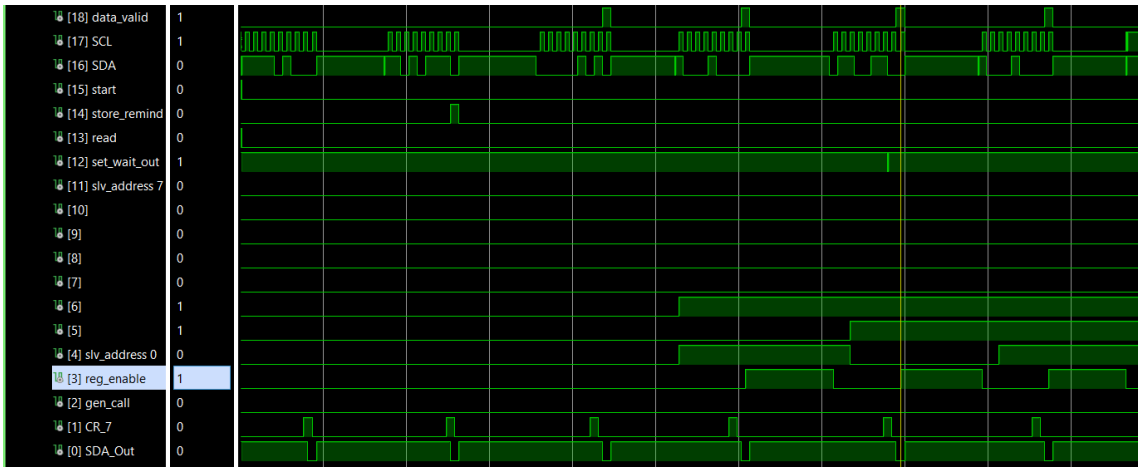

Abbildung 24: reg_enable während I2C-seitig geschrieben wird.

Während I2C-setig gelesen wird, weist reg_enable ein anderes Verhalten auf. Dies ist in Abbildung 25 zu erkennen. Beim ersten Lesezugriff wird reg_enable noch mit der steigenden Flanke des ACKs gesetzt (blaue Linie). Hierbei ist zu beachten, dass das „shift_reg_out“ Modul (Abbildung 20) die Daten der Register, die im nächsten Byte versendet werden sollen, während des vorherigen ACKs (High-Pegel SCL) ausliest. Da die I2C-Kommunikation mit maximal 100 kHz erfolgt, die Bridge und der CAN-Controller jedoch mit 10 MHz arbeiten, können Daten aus dem CAN-Controller ausgelesen und nach „shift_reg_out“ weitergeleitet werden, bevor das ACK mit der fallenden Flanke von SCL beendet wird. Mithilfe des gesetzten „read“ Signals lässt sich auch dieser Fall von dem des Schreibzugriffes unterscheiden. Read wird immer dann gesetzt, wenn ein Read-Bit gesetzt wurde. Beim zweiten Lesezugriff wird reg_enable jedoch erst beim ersten Bit des nächsten Bytepaketes gesetzt (gelbe Linie). Ebenfalls ist das Signal „data_valid“ (ganz oben) nicht verwendbar, da es erst nach dem ACK gesetzt wird. Deshalb muss für den Lesezugriff auf ein anderes Signal zurückgegriffen werden.

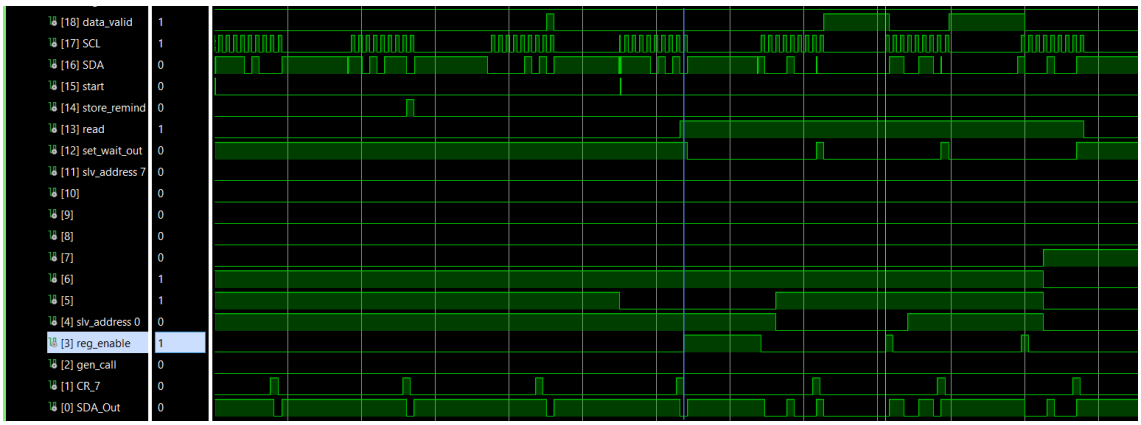

Abbildung 25: reg_enable und read während I2C-seitig gelesen wird.

6.1.2 address

„address“ entspricht dem Signal „address_out” aus dem Modul „mem_adr“ in Abbildung 21. In diesem Modul ist die über I2C empfangene Registeradresse gespeichert. Dieses Signal wird dazu genutzt, die Canakari-Register zu adressieren und als Steuersignal für den Schreib- und Lesezugriff. Die Adresse wird von der „mem_statemachine“ inkrementiert. Dies erfolgt immer zur 3. bzw. 4. steigenden Flanke, während der Datenübertragung. Der Signalverlauf ist in Abbildung 26 zu sehen. Ebenfalls wurde festgelegt, wie die Adressierung der Canakari-Register über I2C zu erfolgen hat. Da jeder gültigen 5-Bit Registeradresse jeweils ein 16-Bit Register zugeordnet ist, das I2C-Slave jedoch eine Registeradresse pro 8-Bit Datenpaket verwendet, wurden den Canakari-Registern I2C-seitig für das höher- und niederwertige Byte, jeweils eine 6-Bit Adresse zugewiesen. Die Canakari-Registeradresse wird dabei jeweils ein weiteres nierderwertiges Bit hinzugefügt. Beispielsweise wird aus der Registeradresse „00000“ für das 16-Bit Canakari-Register „Receive Data 7-8“ auf der I2C-Seite die Adresse „00000|0“ für das höherwertige Byte (Bits 15-8) und „00000|1“ für das niederwertige Byte (Bits 7-0).

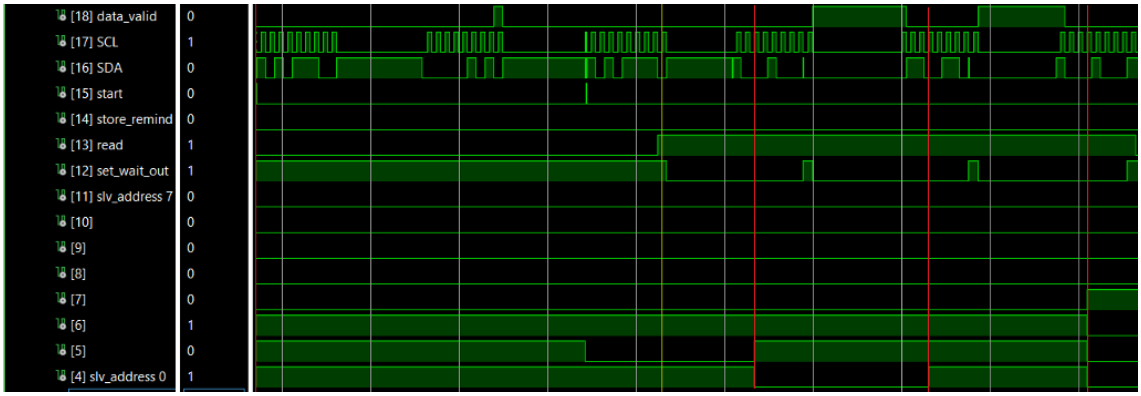

Abbildung 26: Änderung des Signals "address" durch inkrementieren.

Tabelle 4: I2C-Adressen der Canakari-Register

I2C-Adresse	**Name**	**Bits**	**Funktion**
000000	Receive Data 7-8	15-8	Empfangsdatenregister Byte 7-8 (rom)
000001		7-0	
000010	Receive Data 5-6	15-8	Empfangsdatenregister Byte 5-6 (rom)
000011		7-0	
000100	Receive Data 3-4	15-8	Empfangsdatenregister Byte 3-4 (rom)
000101		7-0	
000110	Receive Data 1-2	15-8	Empfangsdatenregister Byte 1-2 (rom)
000111		7-0	
001000	Receive Identifier 2	15-8	Empfangsadresse Bit 12-0
001001		7-0	
001010	Receive Identifier 1	15-8	Empfangsadresse Bit 28-13
001011		7-0	
001100	Receive Control	15-8	Empfangskontrollregister
001101		7-0	
001110	Transmission Data 7-8	15-8	Sendedatenregister Byte 7-8
001111		7-0	
010000	Transmission Data 5-6	15-8	Sendedatenregister Byte 5-6
010001		7-0	
010010	Transmission Data 3-4	15-8	Sendedatenregister Byte 3-4
010011		7-0	
010100	Transmission Data 1-2	15-8	Sendedatenregister Byte 1-2
010101		7-0	
010110	Transmission Identifier 2	15-8	Sendeadresse Bit 12-0
010111		7-0	
011000	Transmission Identifier 1	15-8	Sendeadresse Bit 28-13
011001		7-0	
011010	Transmission Control	15-8	Sendekontrollregister
011011		7-0	
011100	General	15-8	Mehrzweckregister
011101		7-0	
011110	Prescaler	15-8	Vorteilerkonfiguration
011111		7-0	
100000	Acceptionmask2	15-8	Empfangsadressmaske Bit 12-0
100001		7-0	
100010	Acceptionmask1	15-8	Empfangsadressmaske Bit 28-13
100011		7-0	
100100	Interrupt	15-8	Interruptstatus u. -konfiguration
100101		7-0	
100110	Fault Data	7-0	Fehlerregister der Bridge

6.1.3 reset_higher & received_general_call

„reset_higher“ wird dazu genutzt, die Zustandsmaschine der Bridge zurückzusetzen. Der General Call soll von der Bridge nicht verarbeitet werden. Daher werden bei dem Empfang eines General Calls Lese- und Schreibzugriffe der Bridge auf den CAN-Controller unterbunden. Die beiden Signale wurden im Kapitel Userunit bereits ausführlich besprochen.

6.2 Aufbau

Die Bridge besteht aus vier Modulen.

1. data_reg
2. bridge_statemachine
3. fault_data_reg
4. slave_data_output

Signale des Slaves bzw. des Canakari enthalten die Präfixe „slv“ bzw. „cnk“.

Die Funktionen werden in den folgenden Unterkapiteln erläutert.

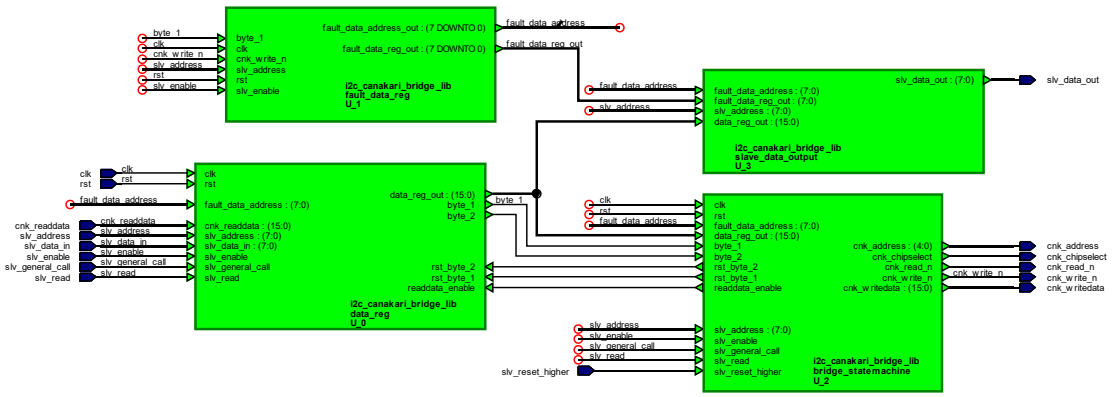

Abbildung 27: Der Aufbau der Bridge.

6.2.1 data_reg

Im 16-bit breiten Datenregister der Bridge werden Daten zwischengespeichert. Zum einen werden die Daten gespeichert, die über I2C übertragen werden und zum anderen die, die aus dem CAN-Controller ausgelesen werden. Nach der Tabelle 4 sind die I2C-Registeradressen mit dem LSB von 0 jeweils das erste Byte (Bit 15-8) der Canakari-Register. Empfangenen I2C-Datenpakete werden daher je nach Wert des LSBs den Canakari-Registerbits 15-8 (LSB = 0) bzw. 7-0 Bits (LSB = 1) zugeordnet. Sie werden in das Datenregister übernommen, sofern die nötigen Steuersignale gesetzt werden. Wenn

dies passiert, werden die an „slave_data_in“ anliegenden Daten in den oberen bzw. unteren 8 Bit des Datenregisters abgespeichert. Die nicht beschriebenen 8-Bit behalten ihre gespeicherten Daten bei. Eine Anforderung für die Bridge war, dass die Canakari-Register nur als Ganzes beschrieben werden. Erst nach der erfolgten Abspeicherung eines höherwertigen Bytes (Regeradresse mit LSB 0) wird der Zustandsmaschine durch das Signal "byte_1" mitgeteilt, dass nun auch die Abspeicherung eines niederwertigen Bytes (Registeradresse LSB 1) erfolgen kann. Datenpakete, welche mit Registeradressen verknüpft sind, die nicht aus dem gültigen Adressraum stammen, werden durch die Bridge nicht verarbeitet. Auf der CAN-Seite ist das Datenregister mit dem Ausgangsport des CAN-Controllers „cnk_readdata“ verknüpft. Die dort anliegenden Daten werden übernommen, wenn das Steuersignal der Statemachine „readdata_enable“ gesetzt wird. Die Daten des 16-Bit Registers werden über das Signal „data_reg_out“ aus dem Modul geführt.

6.2.2 bridge_statemachine

Das Modul „bridge_statemachine“ koordiniert den gesamten Ablauf der Kommunikation. Die Zustandsmaschine besteht aus zwei Zweigen, die dem Lese- und dem Schreibzyklus entsprechen. Zustände werden nur dann gewechselt, wenn bei steigender Taktflanke die entsprechenden Bedingungen aus Abbildung 28 erfüllt sind. Im Idle-Zustand findet keine Kommunikation statt. Es wird nach Idle gewechselt, wenn...

- über dem asynchronen Reset „rst“ alle Module und damit auch die Zustandsmaschine zurückgesetzt werden.
- die Zustandsmaschine über dem synchronen Reset „slv_reset_higher“ zurückgesetzt wird.
- am Ende eines Schreibzykluses.

In der Tabelle 5 sind die Defaultwerte der Ausgangssignale im Idle-Zustand aufgelistet.

Tabelle 5: Defaultwerte des Moduls "bridge_statemachine"

Signalname	Defaultwert
cnk_address	00000
cnk_chipselect	0
cnk_read_n	1
cnk_write_n	1
cnk_writedata	0x0000
readdata_enable	0
rst_byte_1	0
rst_byte_2	1

Wie bereits erwähnt, existiert die Anforderung, dass ein Canakari-Register nur mit einem vollständigen 16 Bit Word überschrieben werden soll, um ein Fehlverhalten der CAN Protokolleinheit durch Dateninkonsistenzen zu vermeiden. Der Schreibzyklus läuft so ab, dass das Modul „data_reg“ ein Datenbyte empfangen muss, welches für ein Register bestimmt ist, dessen Adresse valide ist und ein LSB von 0 hat. Ist dies der Fall, wechselt die Zustandsmaschine in den Zustand „write_wait“. Durch die Setzung des Signals „rst_byte_2“ kann im Modul „data_reg“ das Signal „byte_2“ gesetzt werden. Dieses Signal wird gesetzt, wenn als nächstes ein Byte empfangen wird, dessen zugewiesene Registeradresse ein LSB von 1 hat. Hierbei müssen beide Bits in einem Buszugriff empfangen werden. Auch über einen Repeated-Start ist es nicht möglich, eine andere Registeradresse mit LSB 1 auszuwählen, da bei jedem Start und Stopp die Zustandsmaschine zurückgesetzt wird. Ist die Übertragung der zwei Bytes korrekt verlaufen, wechselt die Zustandsmaschine in den Zustand „write“. Hier wird die ursprüngliche Registeradresse „slv_address(5 downto 1)“ an cnk_address und die in data_reg abgespeicherten Daten auf cnk_writedata gelegt sowie die Signale cnk_write_n (low-aktiv) und cnk_chipselect gesetzt. Nach einem Taktzyklus wechselt die Zustandsmaschine automatisch nach idle.

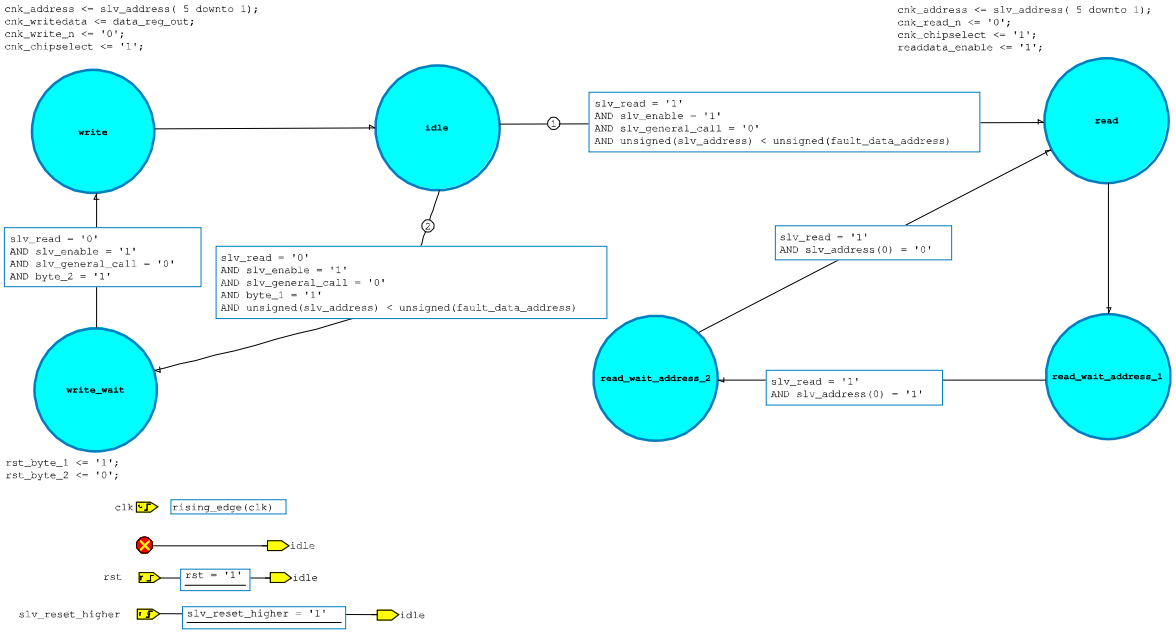

Abbildung 28: Zustandsablauf der Statemachine.

Die Ausgangsituation für den Lesezugriff wurde in den Kapiteln „reg_enable & read“ und „address“ beschrieben. Die Zustandsmaschine wechselt in den Zustand „read“, wenn ein I2C-Lesezugriff mit einer gültigen Registeradresse erfolgt ist. Hierbei spielt das Signal slv_reg_enable eine Rolle, welches zur steigenden Flanke des ACKs anliegt. Die Daten werden daraufhin aus dem Ausgangsbus „cnk_readdata“ des CAN-Controller in das Datenregister geschrieben. Hierbei ist zu beachten, dass das gesamte Register ausgelesen und gespeichert wird, I2C-seitig jedoch zu dem Zeitpunkt nur ein Byte des Registers ausgelesen werden soll. Das Untermodul der Bridge „slave_data_output“ legt dann je nachdem was für ein LSB die Registeradresse hat, die oberen oder unteren 8 Bits des Datenregisters an das Ausgangs-Schieberegister der Protocolunit „shift_reg_out“. Letzteres übernimmt die Daten bis zur nächsten fallenden Flanke. Die Zustandsmaschine wechselt dann in die Wartezustände „read_wait_address_1“ und „read_wait_address_2“. Das Auslesen der Daten aus den Canakari-Registern läuft über die Registeradresse, die sich automatisch inkrementiert. Die Adresse wird hier als Steuersignal genutzt, um die Daten aus dem nächsten Register vorzuladen. Dies erfolgt auf dieser Weise, da keine anderen Steuersignale zur Verfügung stehen (Siehe Kapitel 6.1.1 und 6.1.2). Sobald die Registeradresse mit einem LSB von 1 zu einer mit einem LSB von 0 inkrementiert wird, werden die Daten des entsprechenden Canakari-Registers geladen. Dies ist in der Abbildung 29 zu sehen. Der blaue Marker zeigt den Übergang der Statemachine von Idle nach read. Der gelbe Marker zeigt, wie die neuen Daten in das Datenregister geschrieben

werden, nach dem slv_address von 0x0F (Transmission Data 7-8, Bit 7-0) zu 0x10 (Transmission Data 5-6, Bit 15-8) inkrementiert wird.

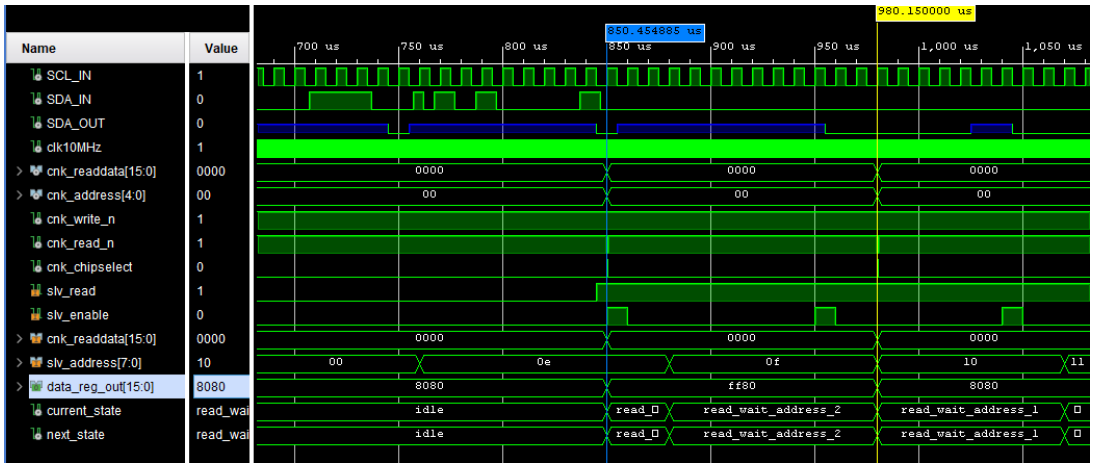

Abbildung 29: Darstellung des Lesezyklus in der Testbench.

Erst wenn der Lesezugriff beendet wurde, wird die Schleife verlassen und in den Idle-Zustand gewechselt.

6.2.3 slave_data_output

Dieses Modul ist ein schlichter Multiplexer. Er leitet Daten von data_reg und fault_data_reg an das Ausgangs-Schieberegister der Protocolunit „shift_reg_out“ weiter. Die Ausgabe ist abhängig von slv_address. Liegt beispielsweise die Adresse des Fehlerregisters an, so liegen die Daten des Fehlerregisters am Ausgang „slv_data_out“ an. Ist slv_address kleiner als die Adresse des Fehlerregisters, so werden in Abhängigkeit des LSB entweder die Bits 15-8 (LSB=0) oder 7-0 (LSB=1) des Datenregisters weitergeleitet.

6.2.4 fault_data_reg

Im Modul „fault_data_reg“ ist ein Register enthalten, welches Fehler des letzten I2C-Zugriffes anzeigt. Es besteht die Möglichkeit eine Auskunft des Fehlerzustands über die Auslesung zu erhalten. Das Fehlerregister ist 8 Bit breit, jedoch werden nur die Bits an der Position 1 und 0 (LSB) genutzt, Bit 7-2 sind reserviert. Die Adresse des Fehlerregisters ist die höchste Adresse. Sie wird in die anderen Module geleitet und dient in den dortigen IF-Abfragen dazu, auszuschließen, dass die Bridge auf Anfragen mit invaliden Registeradressen reagiert. Die Fehler werden durch das asynchrone Reset oder durch darauffolgende, fehlerfreie Zugriffe gelöscht.

Es werden folgende Fehlerfälle behandelt:

Tabelle 6: Fehlerfälle der Bridge

Fehlerstatus der Bridge	**Code**
Kein Fehler	0x00
Schreibender I2C Zugriff, bei dem auf das 1. Byte kein 2. folgte, sodass die Daten nicht in ein Register geschrieben wurden.	0x01
I2C Zugriff mit einer ungültigen Registeradresse. (> fault_data_address)	0x02
Beides möglichen Fehler sind erfolgt.	0x03

6.3 Testbench

In der Testbench werden I2C-Zugriffe simuliert, indem auf SDA_IN bestimmte Pegel zu bestimmten Zeiten gelegt werden. Die Funktionalität der Bridge kann dann anhand der Signalverläufe verifiziert werden.

6.3.1 Die Canakari-Register beschreiben

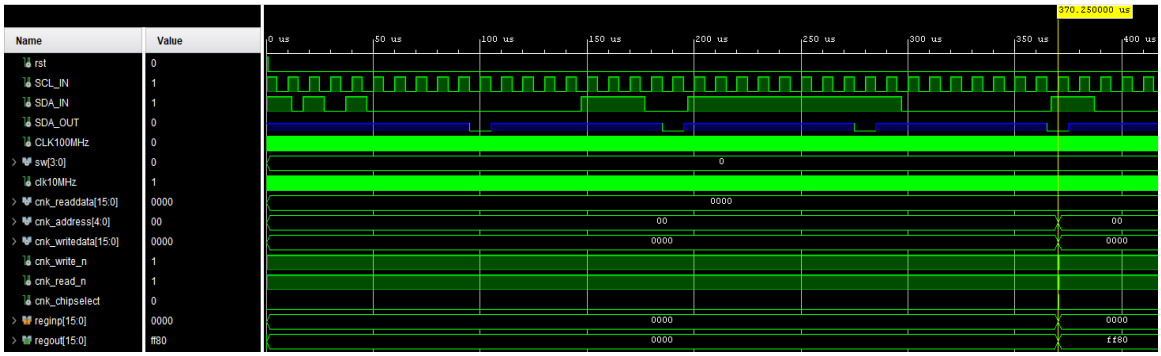

Abbildung 30: Testbench 0-370 µs, 1. Schreibzugriff.

Es wird zunächst die Slave-Adresse versendet 0xA0 (left alligned) (0-90 µs). Danach folgen die Registeradresse 0x07 (100-180 µs) und die Datenpakete 0xFF und 0x80 (200-370 µs). Beim ACK des letzten Datenpaketes wird das Canakari-Register tdata78 („regout[15:0]“) mit den übertragenen Daten beschrieben (370,25 µs). Dies ist in der nächsten Abbildung im Detail zu sehen.

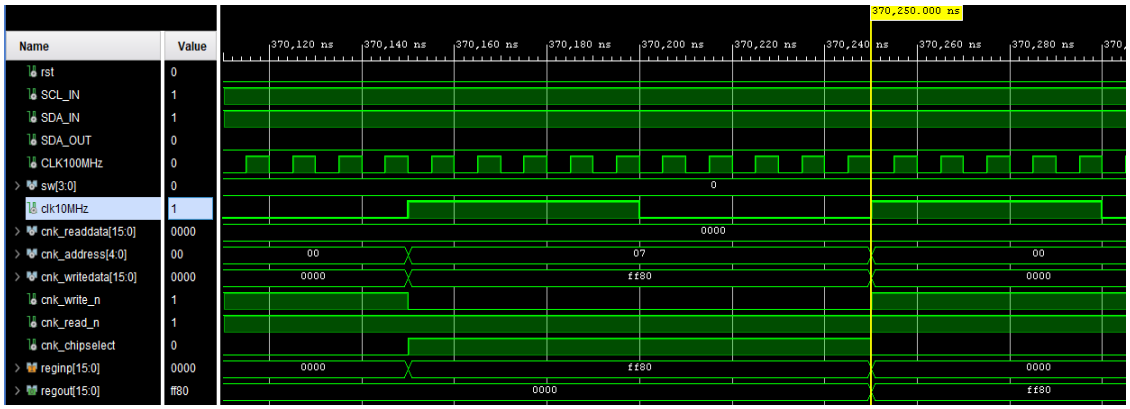

Abbildung 31: Testbench 370 µs, 1. Schreibzugriff.

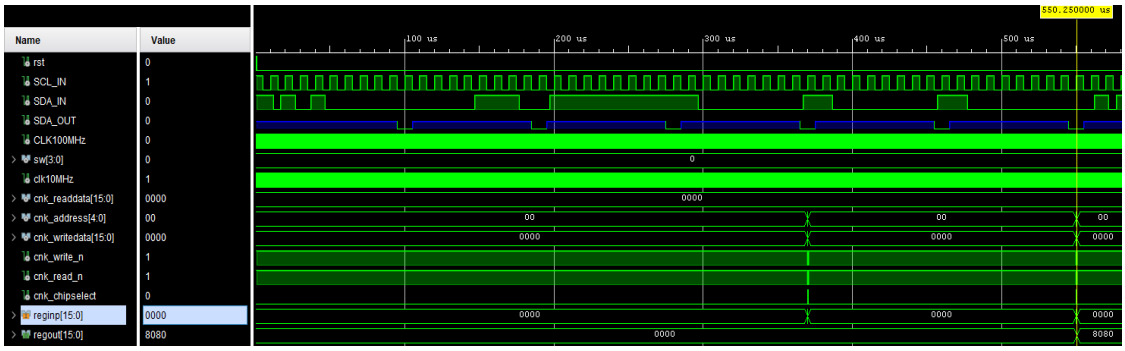

Abbildung 32: Testbench 0- 550 µs, 2. Schreibzugriff.

Nach der Übetragung von zwei Bytes an Daten wird „cnk_address“ jeweils um eins inkrementiert. Deshalb wird das nächste Register beschrieben. In der unteren Abbildung ist im Detail zu sehen, wie das Register tdata56 („regout[15:0]“) mit der Registeradresse 0x08, mit dem Wert 0x8080 beschrieben wird.

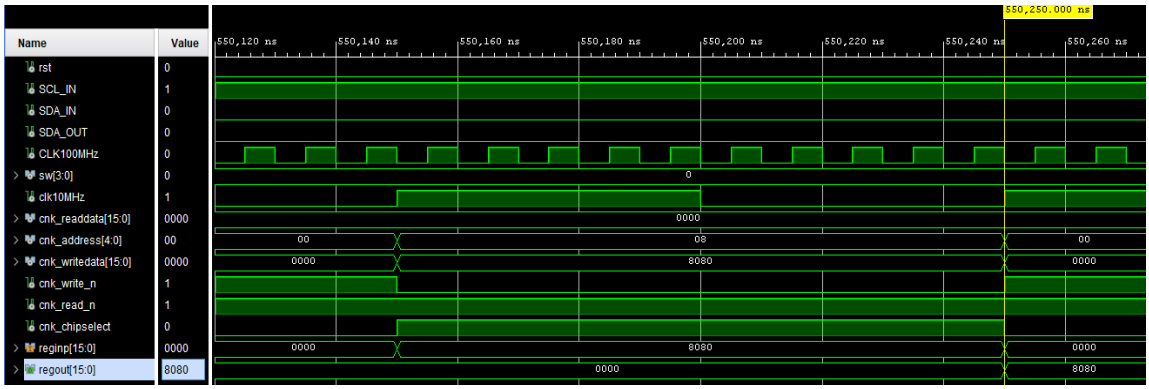

Abbildung 33: Testbench 550 µs, 2. Schreibzugriff.

6.3.2 Daten aus den Canakari-Registern auslesen

In diesem Abschnitt werden die vorher beschriebenen Canakari-Register ausgelesen. Die entsprechenden Daten, müssen vom Slave auf SDA_OUT gelegt werden.

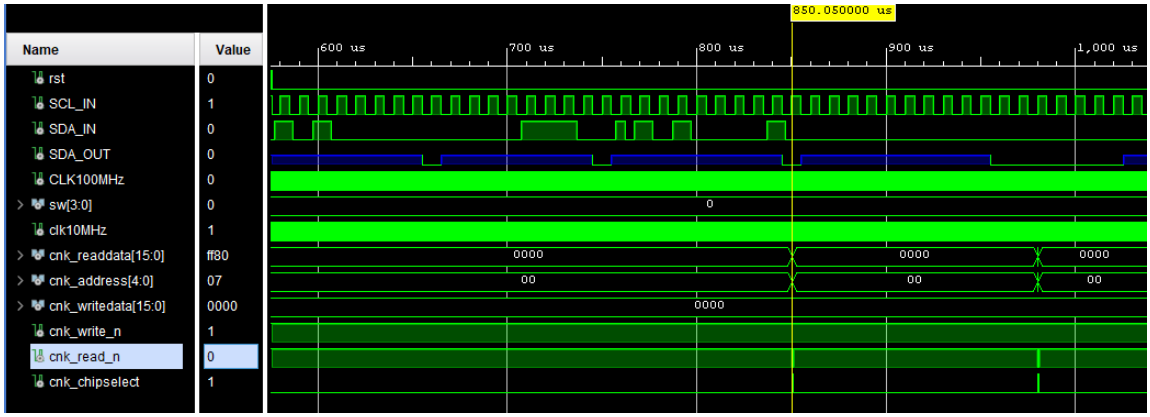

Abbildung 34: Testbench 580-1030 µs, 1. Lesezugriff.

In der oberen Abbildung ist zu sehen, wie das vorher beschriebene Register ausgelesen wird. Bei 570 µs wird eine Startbedingung erzeugt, woraufhin die Adresse 0xA0 und ein R/W-Bit von 0 versendet werden. Daraufhin folgt bei 660 µs das ACK des Slaves auf SDA_OUT. Nun wird die Registeradresse 0x0E versendet (670-740 µs), um auf das vorher beschriebene Register „tdata78“ zugreifen zu können. Bei 762 µs wird ein „repeatet start“ ausgelöst. Es wird erneut die Slave-Adresse versendet, diesmal mit einem gesetzten R/W-Bit (770-840 µs). Nach 850 µs werden dann die Daten aus dem CAN-Controller ausgelesen und direkt danach auf „SDA_OUT“ gelegt.

Zwischen 860 und 1020 µs sind die Daten 0xFF80 auf „SDA_OUT“ zu erkennen. Vom Slave wird ein ACK für das 2. Byte-Paket und für das 1. Byte-Paket des nächsten Registers jeweils bei 940 µs bzw. 1030 µs erkannt. Bei 980 µs wird dann das nächste Register ausgelesen.

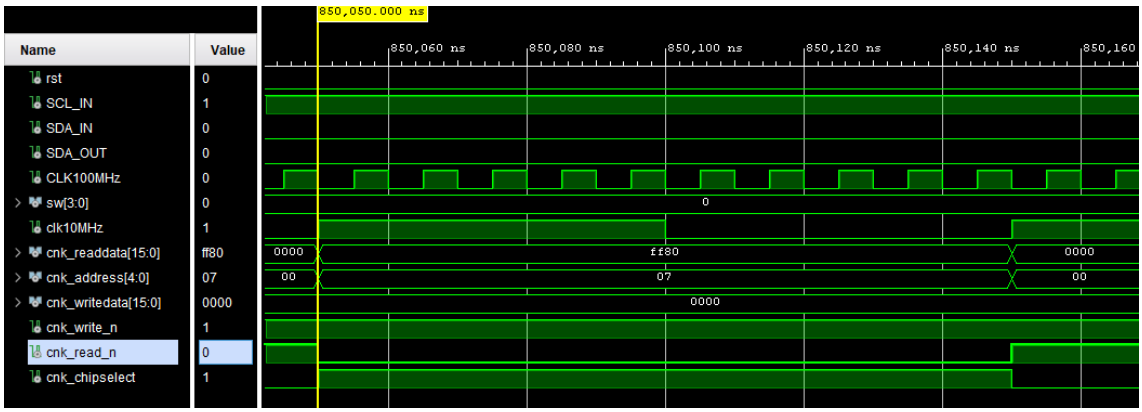

Abbildung 35: Testbench 850 µs, 1. Lesezugriff.

Das Auslesung der Daten ist in der obigen Abbildung im Detail dargestellt. Der Leseprozess wird über die Signale „cnk_read_n“ und „cnk_chipselect“ eingeleitet. Dabei wird die Adresse des auszulesenden Registers auf „cnk_address“ gelegt. Der CAN-Controller legt dann im selben Taktzyklus den Inhalt des Registers 0x07 (tdata78) auf „cnk_readdate[15:0]“, welcher dann von der Bridge abgespeichert und vom Slave in zwei Bytepaketen ausgegeben wird (860 µs – 1020).

6.3.3 Das Fehlerregister der Bridge

In der unteren Abbildung ist zu sehen, wie das Fehlerregister reagiert. Da nur ein Byte an Daten I2C-seitig versendet wurden, ändert das Register seinen Wert ändert bei 1490 µs.

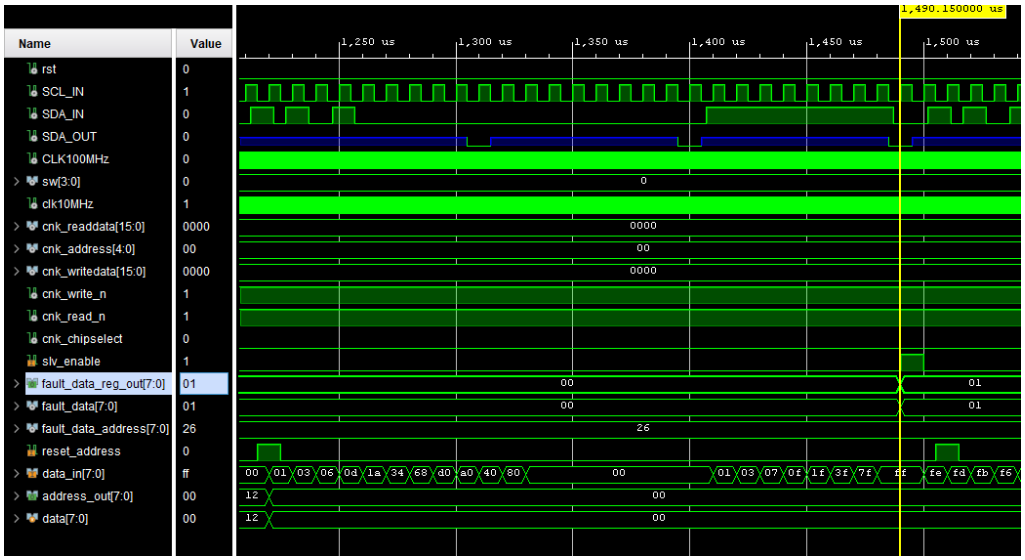

Abbildung 36: Testbench 1210 - 1540 µs, Fehlerregistercode 0x01.

Es wurde die Slave-Addresse 0xA0 mit einem R/W von 0 und das Datenpaket 0xFF versendet. Daraufhin ändert sich der Wert auf 0x01, welches bedeutet, dass nur ein Bytepaket gesendet worden ist, anstatt der erforderlichen zwei Bytes um ein Register zu beschreiben. Dieses Fehlerbit wird auch durch ein „repeated start“ oder Stopp-Signal nicht zurückgesetzt. Erst wenn das asynchrone Reset ausgelöst wurde oder ein doppeltes Bytepaket versendet worden ist, wird es zurückgesetzt.

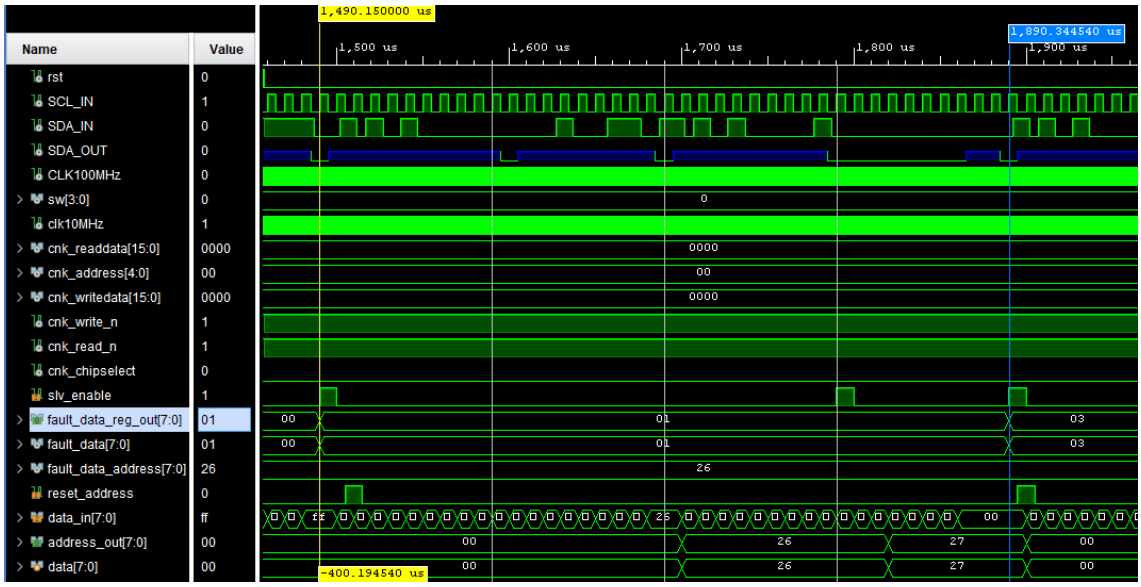

Abbildung 37: Testbench 1460 - 1960 µs, Fehlerregistercode 0x01/0x03.

Als nächstes wird das Fehlerregister I2C-seitig ausgelesen (oberes Bild). Dazu werden nach einem Stopp- und Start-Signal wieder die Slave-Adresse sowie die Adresse des Fehlerregisters 0x26 gesendet. Daraufhin folgt ein „Repeated Start“ und erneut die Slave-Adresse mit einem gesetztem R/W-Bit. Der Slave legt dann ab 1790 µs die Daten des Fehlerregisters auf den Bus. In der obigen Abbildung ist zu erkennen, dass der Wert 1 ausgegeben (1870 µs) wird und vom Slave kein ACK folgt (1880 µs), da es ein ACK vom I2C Master erwartet. Da bei 1880 µs SDA_IN auf 0 gesetzt ist, erkennt der Slave ein ACK. Dies bedeutet, dass weiterhin Daten ausgelesen werden sollen. Da aber die Adresse des Fehlerregisters die größte Adresse ist, wird beim Zugriff der Daten der invaliden Registeradresse 0x27 das zweite Fehlerbit gesetzt. Der Fehlercode beträgt nun 0x03. Da der Slave möglichst nicht verändert werden sollte, sendet dieser auf einen Lesezugriff mit einer invaliden Registeradresse kein NACK sondern den Bitcode 0x00 als Datenpaket und setzt das entsprechende Fehlercode im Fehlerregister. Auf einen Schreibzugriff reagiert die Bridge in diesem Fall gar nicht.

In der nächsten Abbildung wird das Fehlerregister erneut ausgelesen. Zu beachten ist, dass die Auslesung des Fehlerregisters das Fehlerbit für invalide Registeradressen nicht zurücksetzt. Nur durch den Zugriff auf reguläre Registeradressen oder durch ein asynchrones Reset wird jenes zurückgesetzt.

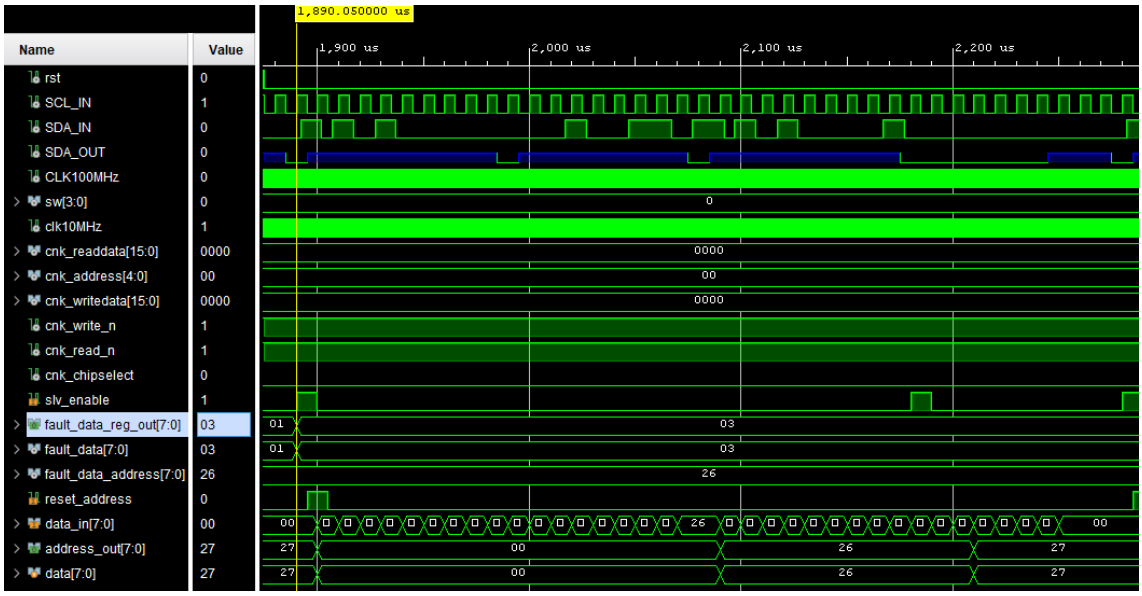

Abbildung 38:Testbench 1870 - 2280 µs, Fehlerregistercode 0x03.

Im unteren Bild sollen zwei Byte versendet werden, um die Fehlercodes zu eliminieren. Zunächst wird wieder die Slave-Adresse versendet, die Registeradresse 0x00 und die Daten 0xFF80. Hierbei ist zu sehen, dass sich das Signal „fault_data[7:0]“ von 0x03 auf 0x01 ändert, da die Daten 0xFF mit einer validen Registeradresse versendet worden sind.

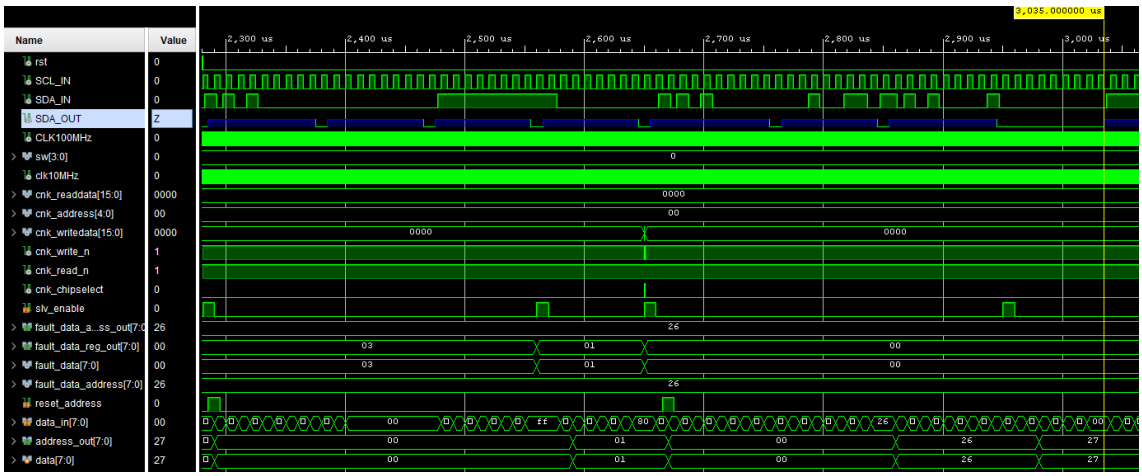

Abbildung 39: Testbench 2280 - 3000 µs, Fehlerregistercodes löschen.

Weiterhin wird auch das andere Fehlerbit zurückgesetzt, sobald das zweite Byte versendet worden ist. Danach wird das Fehlerregister erneut ausgelesen. Die Slave-Adresse und die Adresse des Fehlerregisters wird versendet. Danach erfolgt eine erneute Versendung der Slave-Adresse mit einem gesetztem R/W-Bit. Die Ausgabe ist wie erwartet 0x00 (2950-3030 µs).

7 Hardware

7.1 NEXYS 4 DDR Board

Das NEXYS 4 DDR Board ist ein Entwicklungsboard vom Hersteller Digilent, welches mit einem XILINX FPGA der Artix-7 Familie und einer großen Auswahl bereits integrierter und ansteuerbarer Komponenten wie Knöpfe, Schalter, LED's, 7-Segment-Anzeigen, usw. ausgestattet ist.

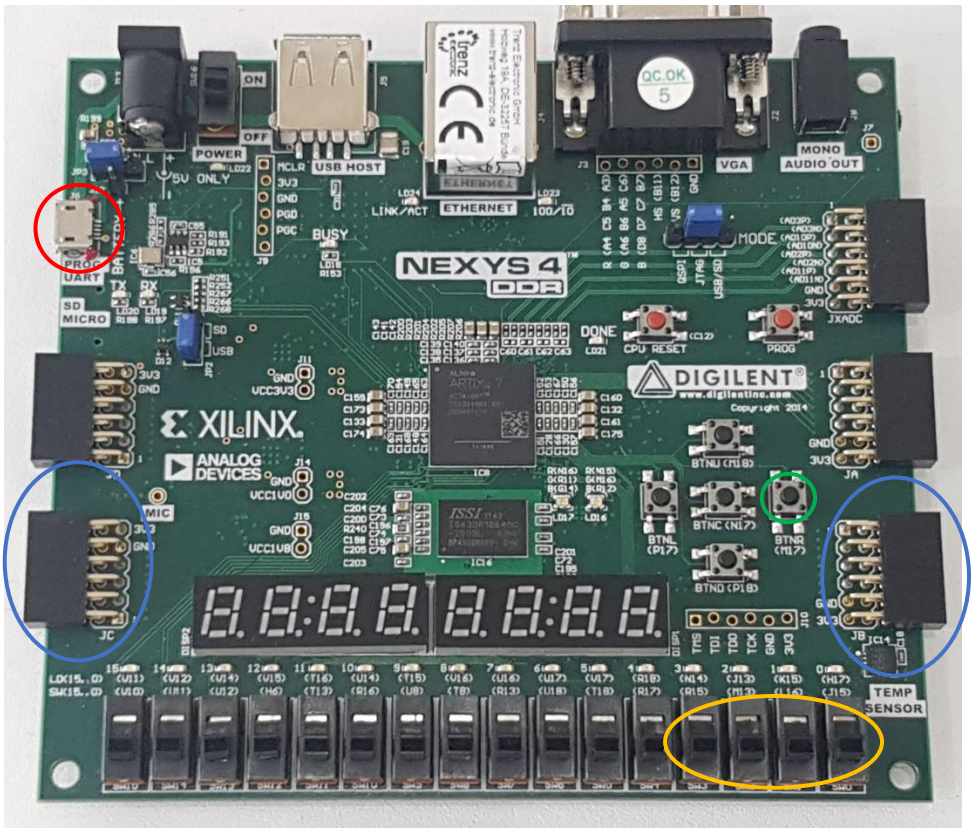

Abbildung 40: NEXYS 4 DDR Board.

In dem Projekt werden vorwiegend die Schalter SW0 bis SW3 (gelb), der Taster BTNR (grün), die Pmod Anschlüsse JB und JC (blau) genutzt. Die Spannungsversorgung sowie die Konfiguration des FPGA wird über den UART/ JTAG USB-Port (rot) mit einer Verbindung zu einem PC realisiert. Auf JB wird das I2C-Modul angeschlossen, während auf JC der CAN-Physical-Layer angeschlossen wird. Über BTNR können alle integrierten Module über das asynchrone Reset zurückgesetzt werden. Weiterführende Informationen finden sich im Benutzerhandbuch. [11]

7.2 FTDI 2232H Mini Modul

Das FTDI 2232H Mini Modul ist ein Entwicklungsmodul, welches mit einem IC ausgestattet ist, das als Schnittstelle für eine serielle synchrone bzw. asynchrone oder parallele FIFO Datenübertragung über USB 2.0 genutzt werden kann. Es können Daten mit verschiedenen Protokollen wie SPI, I²C, etc. versendet und empfangen werden. Über vorher installierte USB Treiber kann dieses Modul angesteuert werden. Es stehen je zwei 26-Pin Anschlussleisten CN2 und CN3 zur Verfügung, mit denen sich das Modul einfach auf Platinen anbringen lässt.

Abbildung 41: FT2232H Mini Module [11, p. 1]

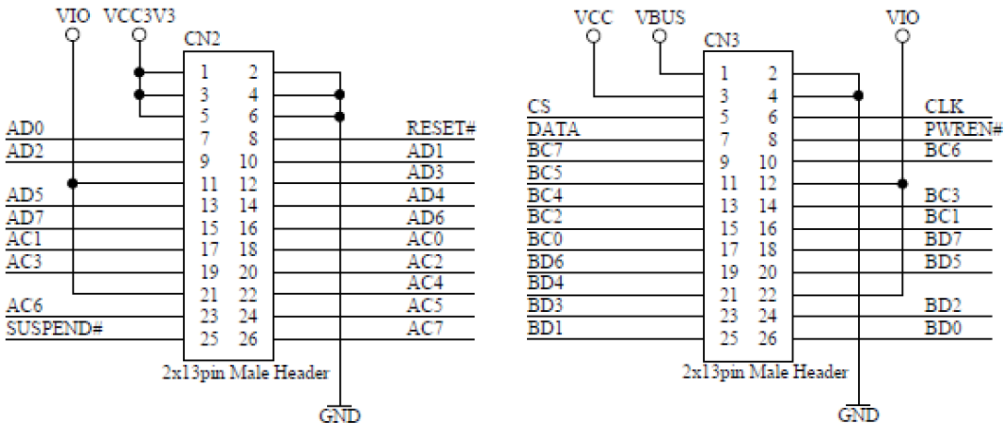

Abbildung 42: Die zwei Anschlussleisten CN2 und CN3 des FT2232H Mini Moduls. [11, p. 9]

Das FTDI Modul wird in diesem Projekt mit dem D2XX Treiber über ein Qt Programm als I²C-Master verwendet. Die nötigen Methoden um das FTDI Modul als I²C Schnittstelle zu nutzen wurden bereits von FTDI entwickelt und sind in der Bibliothek D2XX.dll enthalten, die in einem Programm eingebunden und genutzt werden kann. Die Funktionalität des I²C Master‘s wird über eine im IC integrierte Zustandsmaschine „Multiprotokoll-Synchronous-Serial-Engine“ (MPSSE) realisiert. In diesem Projekt werden folgende Pins zur Kommunikation mit der Zustandsmaschine genutzt.

Tabelle 7: A

Pin	Beschreibung
CN2-7	Takt-Signal
CN2-9	Serieller Dateneingang
CN2-10	Serieller Datenausgang

Es liegt ebenfalls eine API mit entsprechenden Funktionen zur Nutzung des FTDI-Bausteines vor. [12]

7.3 CAN-Interface

Es wurde im Aufbau des Systems das Kvaser USBcan II genutzt. Mit der erhaltenen High Level API, ist auch die Nutzung eines anderen CAN-Interfaces möglich. Das CAN-Interface von Kvaser verfügt über zwei Kanäle, die über D-SUB 9 Stecker an CAN-Netzwerken angeschlossen werden können. Die nötigen Treiber, Installationsanleitung und weitere Infos finden in der Dokumentation. [3]

Abbildung 43: Kvaser CAN-Interface. [13]

7.4 Topmodule Hinweise

- Das Topmodule des Vivado Projektes beinhaltet 4 Module. Den CAN-Controller, die Bridge, der Slave sowie einen Taktteiler. Der Taktteiler teilt für die Taktung des CAN-Controllers und der Bridge die 100 MHz Clock des FPGA-Boards auf 10 MHz runter.

- Im Slave sind die letzten drei Bits der Slave_Adresse an den Schaltern SW3 bis SW0 und das 10-Bit Aktivierungssignal an SW4 angeschlossen. So können verschiedene Slave-Adressen getestet werden.

- Der FTDI-Baustein ist mit seinen Ports CN2-9 und CN2-10 an SDA und mit CN2-7 an SCL angeschlossen. Das SDA Signal des FTDI-Bausteines wird im Topmodule verzögert an den Slave weitergeleitet. Hierbei wird SDA mit 100 MHz abgetastet. Der FTDI Baustein kann in der Virtuellen Maschine nur in USB 1.x oder USB 3.x Modus laufen, da der Baustein ansonsten nicht reagiert. Die Hintergründe hierzu wurden noch nicht geklärt.

- Wenn die Testbench genutzt werden soll, müssen Signale aus dem Topmodul auskommentiert/aktiviert werden. Die entsprechenden Signale sind mit Kommentaren markiert. Da für Hardware Tests im Nachhinein zusätzliche Signale ins Top Modul geführt wurden sowie der Taktteiler eingesetzt wurde. Diese sind in der Testbench nicht berücksichtigt.

- Auf der CAN-Seite werden ausgangsseitig nur die Signale rx und tx benötigt, um die Kommunikation zu führen. Für die I2C-Seite werden nur die Signale SDA_OUT, SDA_IN, SCL_IN verwendet.

- Der CAN-Controller und das Slave verfügen über low-aktive asynchrone Resets, die Bridge jedoch über ein high-aktives.

8 Software

Die Softwareoberfläche wurde in der Qt Umgebung unter Debian in einer virtuellen Maschine entwickelt. Dieser Schritt ist erfolgt, da zum einen die FTDI-Treiber bis ausschließlich Centos 7 lauffähig sind zum anderen die CAN-Treiber erst ab Centos 7 lauffähig sind. Die entwickelte Software sollte das Versenden und Empfangen von CAN und I2C Nachrichten ermöglichen. Die Software zur Ansteuerung des Systems besteht aus zwanzig Dateien und weiteren Treiberbibliotheken. Es sind Bibliotheken von Hardwarezulieferern und APIs enthalten, die von anderen Studierenden in Projektarbeiten erstellt worden sind. Um die Software zu nutzen sind Treiber-Installationen für den FTDI-Baustein sowie für die CAN-Interfaces erforderlich. Diese sind bereits in anderen Arbeiten dokumentiert [3], [14], [15] . Im Rahmen dieser Arbeit wurde die Oberfläche selbst (mainwindow.ui), die Ansteuerung (mainwindow.cpp/ threadworker.cpp) und eine Klasse mit Funktionen erstellt, die speziell für die Ansteuerung des CAN-Controllers über I2C geeignet ist (canakari_control.cpp).

Das Qt-Projekt enthält folgende Dateien:

Dateien	Zweck	Literatur
I2c_can_control.pro	Projektdatei	
api_i2c.h /.ccp	Header Datei der API mit Funktionen zur Kommunikation über I2C mittels FTDI Baustein.	[12]
ftd2xx.h	Bibliothek zur Ansteuerung des FTDI Bausteines.	[11] [15]
WinTypes.h	Bibliothek zur Ansteuerung des FTDI Bausteines.	[11] [15]
can_interface_base.h /.ccp	Klasse die Basisfunktion zur CAN-Komm. mit Interfaces enthält.	[3]
can_interface_kvaser.h /.ccp	Klasse zur Nutzung des Kvaser CAN-Interfaces.	[3]
Can_interface_socketCAN.h /.ccp	Klasse zur Nutztung von SocketCAN-Interfacees	[3]

can_handler.h/.ccp	Wrapper-Klasse zur Nutzung der anderen CAN Klassen. Enthält Funktionen zur Komm. über CAN.	[3]
canakari_control.h /.cpp	Bietet Funktionen die, das beschreiben und auslesen der Canakari register ausführen unter verwendung von api_i2c.	Hier
threadworker.h /.cpp	Enthält Code der Ansteuerung Softwareoberfläche, der in einem anderen Thread ausgeführt werden soll	Hier
Mainwindow.h/ .cpp	Enthält den Code für die Ansteuerung der Softwareoberfläche.	Hier
Main.cpp	Führt mainwindow aus	Hier
Mainwindow.ui	Enthält das Design der Oberfläche.	Hier

8.1 Canakari Control

Die Canakari_Control Klasse wurde auf Basis der api_i2c Klasse entwickelt. Da die Adressen der Register des CAN-Controllers für eine gängige I2C-Kommunikation unpassend gewählt sind, wurde diese Klasse mit Funktionen entwickelt, mit der etwaige Problem umgangen werden konnten.

Hier ist ein Beispiel für eine Funktionen aus der Klasse „api_i2c“:

```
void i2c_write(int slave_adres, int registercontent[],int registeradreslenght,int outputdata[],int numberofbytes);
```

Mit dieser Funktion kann der FTDI-Baustein angesteuert und so I2C Daten verschickt werden. Hierbei muss lediglich die Slave-Adresse, die Registeradresse und deren Länge, das Array mit den Daten und die Anzahl der Bytes angegeben werden. Die Struktur der Funktion soll an Hand des Beispiels der Transmission Data Register des CAN-Controllers erläutert werden. Im Folgenden sind die Registeradressen des I2C Slaves für die 4 Byte des Transmission Data CAN Registers aufgelistet.

010010	Transmission Data 3
010011	Transmission Data 4
010100	Transmission Data 1
010101	Transmission Data 2

Ist es erforderlich, die ersten zwei Senderegister des CAN-Controllers (TD 1-2 und TD 3-4) zu beschreiben, so muss die Funktion einmal für TD 1-2 mit der Registeradresse 1010100 von TD 1 aufgerufen werden und kann dann anschließend automatisch auch das Register TD 2 beschreiben, da die Adresse durch den Zugriff auf das Register TD1 inkrementiert wird. Die Registeradresse von TD 3 ist jedoch niedriger, als die von TD 2, sodass um TD 3 zu beschreiben, der Buszugriff beendet und die Funktion erneut aufgerufen werden muss.

8.1.1 Schreibfunktionen

Gewünscht wären Funktionen, die wie die folgende realisiert wurden:

```
int transmission_data_write(int bytes_to_write, int send_data[]);
```

Bei dieser Funktion wird lediglich die Anzahl der Bytes und das Array mit den Daten übergeben. Ein mehrfacher Funktionsaufruf wird unnötig.

Neben einigen Fehlerabfragen sind die Funktionen im Kern wie folgt aufgebaut.

Es gibt eine Grundfunktion für den schreibenden Zugriff auf die Register des CAN Controllers:

```
int canakari_control::write_cnk_reg(int reg_address, int send_data[])
```

Diese ruft die Funktion der „api_i2c“ immer mit einer Länge von zwei Bytes auf, sodass die Daten eines ganzen 16-Bit Registers geschrieben werden.

```
i2c.i2c_write(slave_address, &reg_address, 1, send_data, 2);
```

Darauf aufbauend setzt die Funktion „transmission_data_write“ folgendes um:

```
for(int i=0, x=0, y, reg_address=0x14, data[2]; i<(bytes_to_write/2); i++)
  {
     data[0] = send_data[x];
     data[1] = send_data[x+1];
     y = write_cnk_reg(reg_address, data);

     if (y != 0)
        {return y;}

     reg_address=reg_address-0x2;
     x=x+2;
  }
```

Aus dem an die Funktion übergebenen Array „send_data“, werden bei jedem Schleifendurchlauf zwei Byte an Daten dem Array „data“ übergeben und mit der Funktion „write_cnk_reg“ verschickt. Dies geht solange, bis die kompletten 8 Byte der vier Transmission_Data Register beschrieben wurden. Die Registeradresse wird nach der Übertragung um den Wert zwei dekrementiert. Gemäß des vorherigen Beispiels, führt dies dazu, dass die Adresse von Transmission Data 1 „010101“, durch die Dekrementierung auf die Adresse „010010“ des nächsten Bytes Transmission Data 3 verändert wird.

8.1.2 Lesefunktionen

Für den Lesezugriff stellt es sich ähnliche da. An die Funktion „transmission_data_read“ wird die Anzahl der zu lesenden Bytes und ein Array zum Abspeichern der Daten übergeben.

```
int transmission_data_read(int bytes_to_read, int read_data[]);
```

In der darauffolgenden Verarbeitung wird die Funktion „read_cnk_reg“ aufgerufen, die zwei Bytes ausliest, was jeweils einem 16-Bit Register entspricht. In der Schleife werden die zwei Bytes im Array „read_data“ abgespeichert. Die Adresse wird wieder um zwei verringert. Im nächsten Durchgang wird dann das nächste Register ausgelesen und in die nächsten zwei Stellen des Arrays abgelegt.

```
for(int i=0, x=0, y, reg_address=0x14, data[2]; i<(bytes_to_read/2); i++)

  {

  y= read_cnk_reg(reg_address, data);

  if (y != 0)
     {return y;}

  read_data[x] = data[0];
  read_data[x+1] = data[1];
  reg_address=reg_address-0x2;
  x=x+2;
  }
```

So ähnlich sind im Prinzip alle Funktionen dieser Klasse aufgebaut. Funktionen die kein „bytes_to_read“ als Parameter beinhalten, sind für die Behandlung einzelner Register vorgesehen. Diesen wird immer ein zwei Byte Array bzw. Pointer übergeben.

8.1.3 Rückgabewerte

Die Rückgabewerte der Funktion sind wie folgt definiert:

Tabelle 8: Rückgabewerte canakari_control

Rückgabewert (int)	Bedeutung
0	Keine Fehler durch die Funktionen der Klasse.
1	Ungültige Registeradresse (max).
2	Adressformat wurde nicht gesetzt.
3	Bytes_to_read ist eine unzulässige Zahl (max. 8).
4	Bytes_to_read ist eine ungerade Zahl.

8.1.4 Adress- und Frequenzfunktion

Die Slave-Adresse und die Frequenz der I2C Kommunikation werden ebenfalls über Funktionen in privaten Variablen gespeichert, auf die die anderen Funktionen zugreifen können.

```
void set_slave_address(int slv_addr, bool slv_addr_type);
void set_i2cfrequency(int frequency);
```

8.2 mainwindow

Über die Klasse mainwindow wird die grafische Nutzeroberfläche (GUI) mit allen anderen Klassen verknüpft. Sie besteht hauptsächlich aus 22 Slots, die Quellcode beinhalten. Diese Slots werden, bis auf wenige Ausnahmen, durch Buttons ausgelöst. Die Quellcodes entnehmen im Prinzip die Eingaben aus der Benutzeroberfläche, fangen fehlerhafte Eingaben ab, wandeln die erhaltenen Daten so um, dass sie von den unteren Schichten verarbeitet werden können und versenden sie letztendlich. Ebenfalls gibt es Slots die Daten empfangen, umwandeln, analysieren und dann auf der Benutzeroberfläche anzeigen. Aufgrund des großen Umfanges können nicht alle Funktionalitäten des Mainwindows erläutert werden. Anhand eines Beispiels wird jedoch aufgezeigt, wie das prinzipielle Schema aussieht.

8.2.1 on_pushButton_canreceivemessage_clicked()

Ein interessanter Slot ist „on_pushButton_canreceivemessage_clicked()", da hiermit eine spezielle Problemstellung gelöst wurde. In diesem Slot, welcher über einen Button aufgerufen wird, soll das CAN-Interface mittels einer Funktion auf den Empfang von Daten warten. Das Problem hierbei ist, dass die Funktion das Programm blockiert, sodass keine Nachricht an das Interface gesendet werden kann. Damit die GUI noch ansprechbar bleibt, muss die Funktion in einem anderen Thread ausgeführt werden. Dieser Thread wiederum, muss die Daten, welche die Funktion empfangen hat an das Hauptprogramm übergeben. Dies ist mit Qt-spezifischen Signalen realisiert worden. Im ersten Code-Abschnitt, der im Folgendem dargestellt ist, wurden Variablen definiert und Daten aus der GUI entnommen:

```
void MainWindow::on_pushButton_canreceivemessage_clicked()
{
    QString channelnumber_qstr, timeout_qstr;
    unsigned int channelnumber_uint, timeout_uint;
    ui->textEdit_can->insertPlainText( " \n read data: ");
    channelnumber_qstr=ui->lineEdit_cancommunicationchannel->text();
    timeout_qstr=ui->lineEdit_cantimeoutreceive->text();
```

Daraufhin werden invalide Eingaben abgefangen ggfs. in der GUI angezeigt und die Werte umgewandelt.

```
    channelnumber_uint = channelnumber_qstr.toUInt(&ok, 10);
    if (!ok)
        {
            ui->textEdit_can->insertPlainText( "channelnumber cant be
converted to uint \n");
            return;
        }

    timeout_uint = timeout_qstr.toUInt(&ok,10);
    if (!ok)
        {
            ui->textEdit_can->insertPlainText( "timeout cant be
converted to uint \n");
            return;
        }
```

Danach wird ein Objekt „receiving_message“ der Klasse ThreadWorker erstellt. ThreadWorker ist vom Typ QThread.

```
ThreadWorker *receiving_message = new ThreadWorker;
```

In der Headerdatei des ThreadWorker sind die unteren Signale definiert, mit denen kommuniziert werden soll.

```
void receiving_finished(unsigned int channel_number, unsigned int
identifier,    unsigned int flags, QString message, unsigned int
length);
void finished();
void error();
```

Die Signale werden nun, mit anderen Slots und Funktionen verknüpft. Im mainwindow Slot wird der Befehl connect angewendet, um das Signal „receiving_finished“ des Objektes „receiving_message” mit dem Slot “onreceiving_finished” zu verknüpfen. Hierbei werden zusätzlich Daten ausgetauscht. Diese Verknüpfung dient dazu, die empfangenen Daten in den Haupthread zu übergeben, wo sie in der GUI dargestellt werden können.

```
connect(receiving_message , SIGNAL(receiving_finished(unsigned int , unsigned int ,
unsigned int , QString , unsigned int )),
        this, SLOT(onreceiving_finished(unsigned int , unsigned int , unsigned int , QString ,
unsigned int )));
```

Das Signal “finished()” des Objektes „receiving_message”, wird mit dem Slot „deleteLate()“ des selben Objektes verknüpft. Dies dient dazu, den Thread nach dem ausführen des Quellcodes wieder freizugeben.

```
connect(receiving_message, SIGNAL(finished()),
receiving_message, SLOT(deleteLater()));
```

Mit der unten aufgeführten Verbindung wird im Fall, dass der Empfang einer Nachricht fehlgeschlagen ist, über das Signal „error()) der Slot „onerror()“ im mainwindow aktiviert, der dann die Fehlermeldung in der GUI darstellt.

```
connect(receiving_message, SIGNAL(error()),
this, SLOT(onerror()));
```

Daraufhin werden dem ThreadWorker mittels einer definierten Variablen und ein Pointer auf das Interface übergeben und der Thread gestartet mittels der Funktion„start()“.

```
receiving_message->set_information(interface, channelnumber_uint, timeout_uint);
receiving_message->start();
```

8.2.2 ThreadWorker

Im ThreadWorker wird dann mittels „try“ versucht die Funktion auszuführen. Sollte dies nicht gelingen, wird der catch Zweich ausgeführt und die Signale error() und finished() aktivieren ihre jeweiligen Slots. Während die unten dargestellte Funktion ausgeführt wird, kann die GUI weiterhin verwendet werden.

```
try
        {
           length = caninterface->get_message(channel_number, &identifier, message, &flags,
timeout);
        }

 catch(...)
        {
           emit error();
           emit finished();
        }
```

Wird die Funktion erfolgreich ausgeführt, werden die empfangenen Daten zur besseren Übertragung von unsigned char arrays zu QStrings gewandelt. Die erhaltenen Daten werden per emit, an das mit dem Signal verknüpfte Slot verschickt.

```
        std::stringstream s;
        s << "0x:";
        for(unsigned int i=0; i < length; i++)
            {
               if(message[i] < 0x10)
                {    s << "0";}

               s << std::hex << (unsigned int) message[i] << " | ";
            }
        QString message_qstr = QString::fromStdString(s.str());

        emit receiving_finished(channel_number, identifier, flags,
message_qstr , length);
        emit finished();
```

8.2.3 onreceiving_finished

Im Slot onreceiving werden dann die Daten verarbeitet und in der GUI angezeigt.

```
void MainWindow::onreceiving_finished(unsigned int channel_number, unsigned int
identifier, unsigned int flags, QString message, unsigned int length)
{
   if(length == 0)
      {
         ui->textEdit_can->insertPlainText("no message");
         return;
      }

   stringstream receive_msg_strstm;
   receive_msg_strstm << "Received message:: ch-number: " << channel_number << "
identifier: " << identifier << " flag: " << flags <<" message: " << message.toStdString() << "
messagelength: " << length;
   QString receive_msg_qstr = QString::fromStdString(receive_msg_strstm.str());
   ui->textEdit_can->insertPlainText(receive_msg_qstr);
```

8.3 Softwareoberfläche

Die Softwareoberfläche ist in zwei Bereiche aufgeteilt. Der Bereich I2C-Canakari-Control dient dazu, die Canakari-Register über I2C zu beschreiben und auszulesen. Der Bereich CAN-Interface-Control wird verwendet, um CAN-Interfaces anzusteuern.

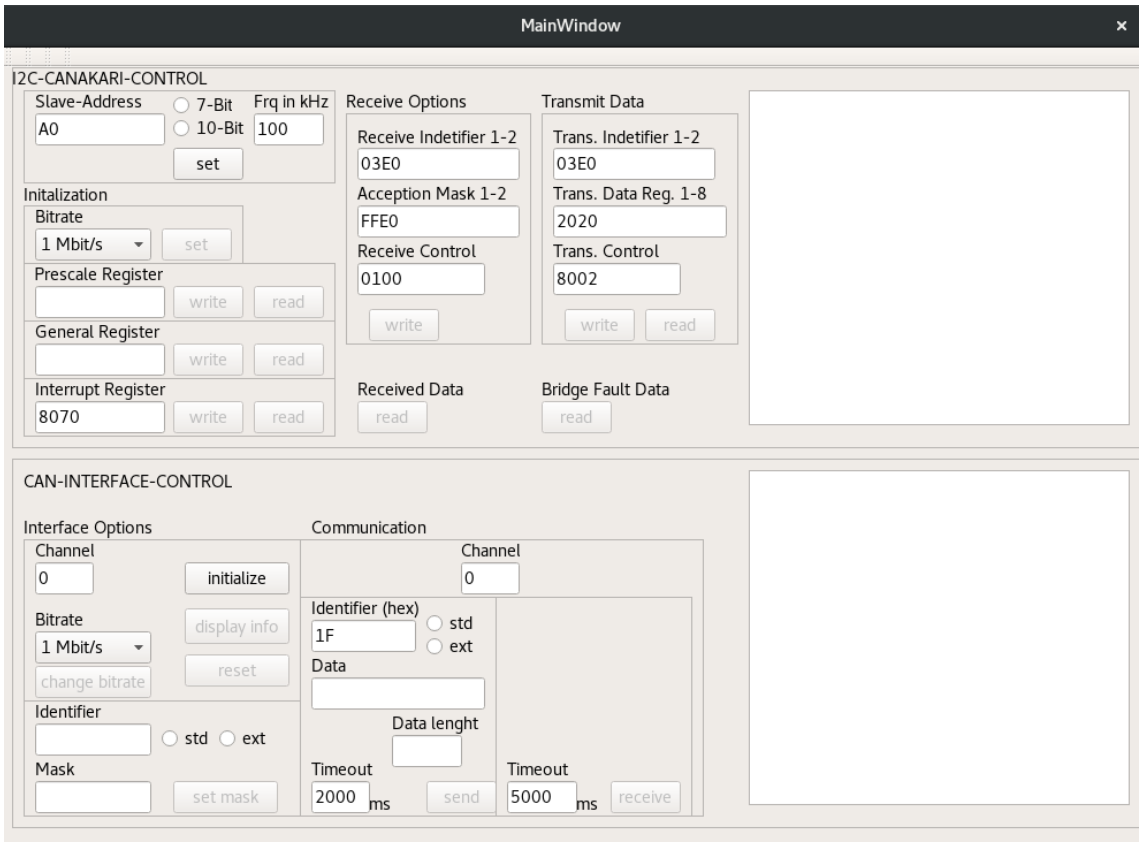

Abbildung 44: Die Softwareoberfläche.

8.3.1 I2C-Canakari-Control

I2C-Canakari-Control teilt sich in mehrere Bereiche auf. Es bietet Optionen für Einstellungen im Slave, zur Initialisierung der CAN Controllers, zum Setzen der Empfangs- und Sendeoptionen sowie zur Auslesung von Daten. Informationen werden in der TextBox rechts angezeigt. In den Eingabefeldern werden immer Hexadezimalwerte eingegeben, außer im Falle der Frequenz. Zunächst kann eine Slave-Adresse für die Kommunikation festgelegt werden. Hierbei muss diese mit jener des tatsächlich verwendeten Slaves übereinstimmen, andernfalls kann keine erfolgreiche Kommunikation stattfinden. Hierbei ist zu beachten, dass eine Registeradresse eingegeben wird, die left-alligned ist. Dies bedeutet, dass der eingegebene 8-Bit Hex-Wert im Falle einer 7-Bit Slave-Adresse Registeradresse rechts abgeschnitten wird. Die Übertragungsfrequenz darf ebenfalls maximal 100 kHz betragen. Nach dem setzten der Slave-Adresse werden weitere Buttons aktiviert. Die Bitrate des CAN-Controllers wird über das Prescale- und Generalregister eingestellt. Es ist jedoch auch möglich über einen Dropdown-Menu verschiedene, voreingestellte Bitraten auszuwählen und zu setzen. Über Receive Data „read“, werden alle für den Empfang relevanten Register ausgelesen. Hierbei sollte auf die Eingaben geachtete werden, die in Tabelle 3 dargestellten sind. Es sind reservierte Bits in den Registern der Acceptionmasks enthalten. In der Software werden Eingaben zurückgewiesen, die reservierte Bits setzten.

8.3.2 CAN-Interface-Control

CAN-Interface-Control ist grob in den Bereichen Interface Options und Communication aufgeteilt. Über Initialize wird das CAN-Interface initialisiert. Über Display Info, lassen sich z.B. Typ- oder Statusinformationen anzeigen. Durch ein Dropdown-Menu lassen sich verschiedenen Bitraten auswählen und setzten. Hierbei ist zu beachten, dass die Bitrate des CAN-Interfaces, der Bitrate des CAN-Controllers entsprechen muss. Eine Identifier Maske, um bestimmte CAN-Nachrichten zu empfangen, lässt sich ebenfalls einstellen. Im Communication Bereich lassen sich CAN-Nachrichten verschicken und empfangen. Hier ist ebenfalls zu beachten, dass nur Hex-Werte eingegeben werden sollten, außer bei Data length, Channel und den Timeouts. Im Gegensatz zu I2C-Canakari-Control, werden in diesem Abschnitt keine Registerwerte beschrieben. So wird der Identifier vor dem Senden einer Nachricht mit 0x1F eingestellt, während im obigen Fall anhand der Dokumentation der Register aus Tabelle 3 festgestellt werden muss, wie ein Identifier mit dem Wert 0x1F einzutragen ist. Im Fall des Trans. Identfier 1-2, entspricht der eingetragene Registerwert 0x03E0, einem Identifier 0x1F.

8.3.3 Empfangsfall des CAN-Controllers

Im Empfangsfalls werden Daten über das CAN-Interface an den CAN-Controller geschickt. Hierzu müssen zunächst die Slave-Optionen Adresse und Frequenz gesetzt werden. Dann muss die Bitrate festgelegt werden. Nun wird der Physical Layer im Interruptregister aktiviert, damit ACKs gesetzt werden können. Es kann bei Bedarf eine Acceptionmask gesetzt werden.

Als nächstes muss der CAN-Controller initalisiert werden. Hierzu wird der Channel auf dem Wert 0 belassen, die Bitrate gleich der des CAN-Controllers gesetzt und mit initalize die Konfiguration bestätigt. Über display info kann überprüft werden, ob das CAN-Interface aktiv ist. Nun können Identifier, Data, Data lenth und das timeout eingetragen werden. Mit send wird die CAN-Nachricht verschickt.

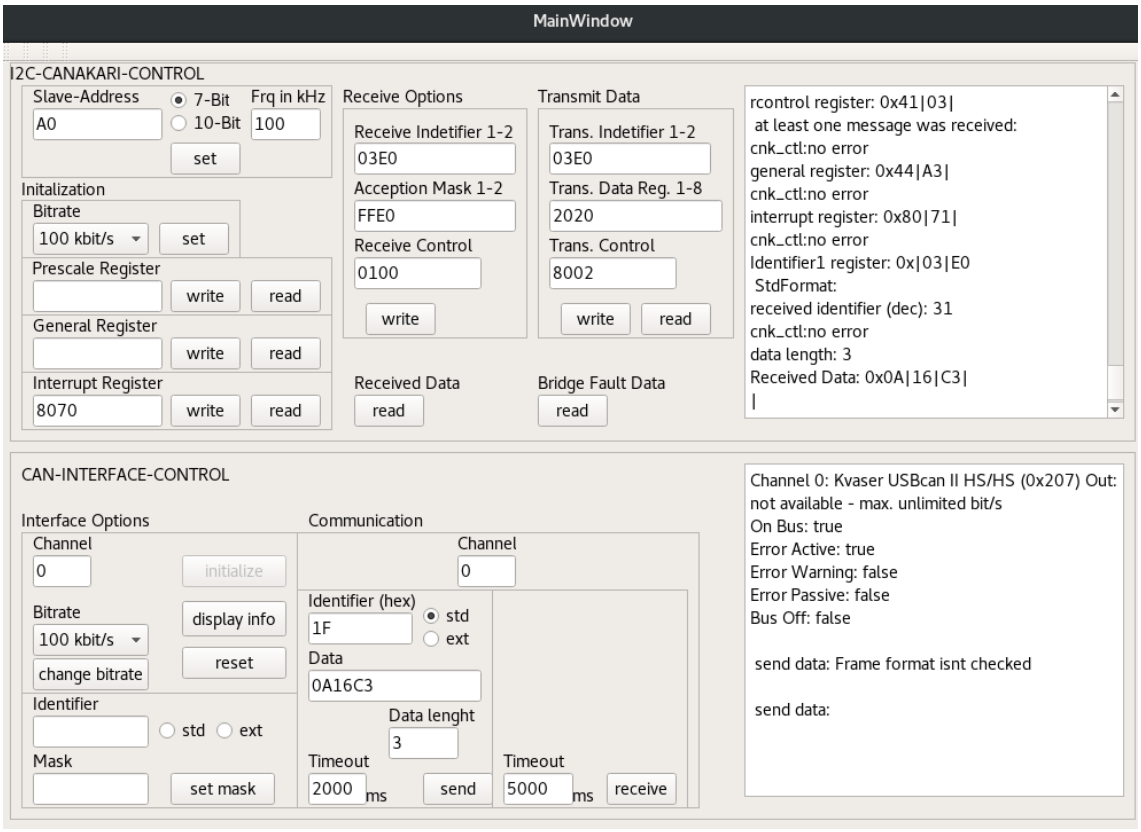

Abbildung 45: Empfangsfall des CAN-Controllers

Als nächstes lässt sich über Receive Data read, die empfangenen Daten auslesen. Diese werden dann in der oberen TextBox angezeigt. In den unteren Abbildungen ist die Messung der CAN-Nachricht sowie ein Ausschnitt aus der I2C Kommunikation zu sehen.

Abbildung 46: Die an den CAN_Controller gesendete Nachricht mit den Daten 0x0A16C3.

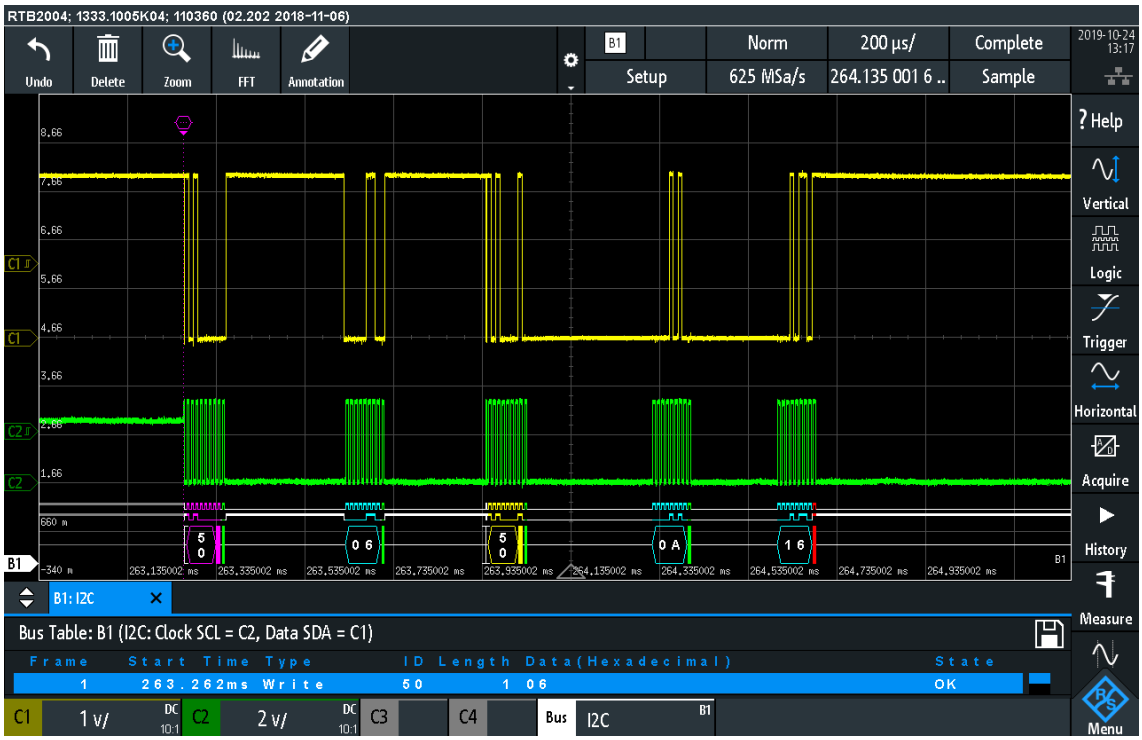

Abbildung 47: Das Auslesen des Registers Receive Data 12, mit den empfangenen Daten 0x0A16.

8.3.4 Sendefall des CAN-Controllers

Im Sendefall des CAN-Controllers, wird ähnlich vorangegangen. Im Unterschied zum Empfangsfall muss die Kommunikation durch die Beschreibung des treq-bits (MSB), im Transmission Control Register gestartet werden. Bevor dies getan wird, muss im CAN-Interface die Funktion zum Empfang von Daten über Communication receive aktiviert werden. Bis zum Ablauf des Timouts lässt sich nun die Übertragung der CAN-Nachricht über Transmit Data write starten. Wenn die Kommunikation erfolgreich war, werden die Daten wie in der unteren Abbildung in der TextBox angezeigt.

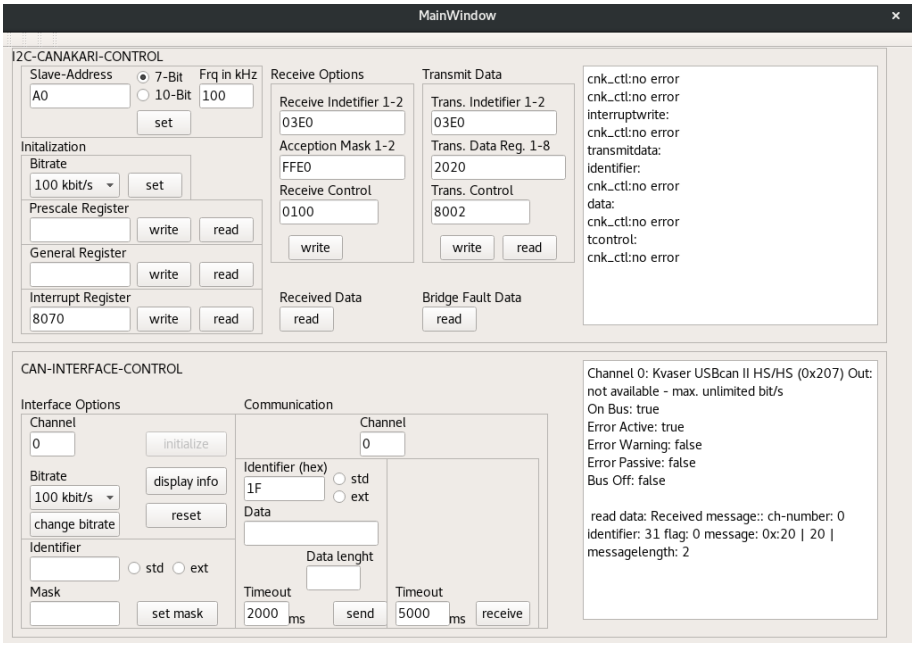

Abbildung 48: Sendefall des CAN-Controllers.

Abbildung 49: Über I2C versendete CAN-Nachricht, mit den Daten 0x2020.

9 Ausblick

In Bezug auf zukünftige Entwicklungen stellt sich die Frage, welche Erweiterung des Systems sinnvoll wären. Zunächst bietet sich der Entwurf einer Benutzeroberfläche mit höherem Bedienkomfort an. Diese Oberfläche würde die manuelle Berechnung von Registerwerten ablösen durch die direkte Umsetzung der gewünschten Einstellungen über eine automatische Ermittlung der benötigen Bitkombination und die Übermittlung dieser Daten in das entsprechende Register. Dies wurde im Fall des Dropdown-Menu für die Konfiguration der Bitrate des CAN-Controllers bereits demonstriert. Für die Durchführung von Systemtests wäre eine Erhöhung des Automatisierungsgrades ebenfalls wünschenswert und hilfreich. Hierbei kann Beispielsweise der I2C-Master immer wechselseitig eine CAN-Nachricht versenden und die Empfangsregister danach auslesen. Auf der CAN-Seite wird, wenn eine Nachricht empfangen wurde, mit einer weiteren geantwortet. Beispielsweise lässt sich hier ein Hexadezimalwert verschicken, der bei jedem hin und zurück inkrementiert wird. Da die I2C-Kommunikation langsamer ist, wird die Nachricht vom CAN-Controller Empfangen, bevor der I2C Master die Empfangsregister ausliest. Dieser kann dann wiederum den empfangenen Hexadezimalwert inkrementieren und erneut versenden. Ein weiterer automatisierter Test wäre beispielsweise ein Algorithmus, der bei jeder neuen Nachricht auf beiden Seiten einen zufällige Identifier einstellt. Außerdem wird daran gearbeitet, in Zukunft das teure CAN USB Interface durch einen Microcontroller mit CAN Protokolleinheit und Physical Layer Baustein zu ersetzen.

Literaturverzeichnis

[1] K. B. J. B. S. K. P. K. P. M. C. Z. L. Püllen, „Studies for the detector control system of the ATLAS pixel at the HL-LHC,“ 2012.

[2] NXP Semiconductors N.V., „https://www.nxp.com/,“ 04 04 2014. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[3] A. Walsemann, „Entwicklung einer universellen High-level CAN-API für USB-CAN-Adapter unter Linux,“ Dortmund, 2017.

[4] T. Büth, „Logikentwurf, Layout und Verifikation einer I2C Einheit für einen ASIC, diploma thesis,“ Faculty of Information-, Media- and Electrical Technology, University of Applied Sciences Cologne, Cologne, 2009.

[5] M. Karagounis, „Design eines CAN Controllers mit VHDL und SpecCharts,“ Fachhochschule Köln, Köln, 2000.

[6] T. Krawutschke, „Test und Synthese eines CAN–Controllers, “ Fachhochschule Köln, Köln, 2001.

[7] A. Kirsten, „Entwicklung eines Linux Device Treibers für einen CAN–Controller, “ Fachhochschule Köln, Köln, 2004.

[8] T. Hetzler, „Implementation eines VHDL–CAN–Controllers in ein Embedded System,“ Fachhochschule Köln, Köln, 2004.

[9] A. Walsemann, „Entwicklung einer Testebench in VHDL zur Verifikation von CAN-Controllern,“ Dortmund, 2018.

[10] A. Beer, „Dokumentation, Verifikation und Modifikation des CANakari CAN-Controllers, Projektarbeit 1,“ FH-Dortmund, Dortmund, 2019.

[11] Future Technology Devices International Limited, „https://www.ftdichip.com/,“ 25 06 2012. [Online]. Available: https://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_FT2232H_Mini_Module.pdf.

[12] N. Özkan, „Entwicklung einer API für die I2C Schnittstelle des FTDI 2232H Mini Moduls,“ Dortmund, 2019.

[13] Kvaser, „kvaser.com,“ 2019. [Online]. Available: https://www.kvaser.com/product/kvaser-usbcan-professional-2/.

[14] R. Paulus, „Qt-Projekt von Herrn Oezcan (FTDI_I2C) auf einen anderen Linuxrechner portieren,“ Dortmund.

[15] N. Özkan, „Konfiguration und Inbetriebnahme des FTDI 2232H Mini Moduls als I2C Schnittstelle,“ Dortmund, 2019.

[16] Digilent Inc., „https://www.xilinx.com/,“ 11 09 2014. [Online].Available:https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPNexys4DDR/documentation/Nexys4-DDR_rm.pdf.

Anhang

CD mit folgendem Inhalt:

- Bachelorthesis (pdf)
- HDL-Designer Projektordner
- VHDL-Quellcodes
- Der Vivado Projektordner
- Die Softwareumgebung.