Fachhochschule Dortmund
University of Applied Sciences and Arts

elmos

Entwicklung eines Testsystems für den Physical- und Data-Link-Layer des PSI5-Busses mit automatisierter Auswertung auf Basis der Xilinx ZYNQ SoCs

Masterthesis

Fachhochschule Dortmund

Fachbereich 3 - Elektrotechnik

Labor für integrierten Schaltungsentwurf

Elmos Semiconductor AG

Erstprüfer:	Prof. Dr. Michael Karagounis
Zweitprüfer:	Dipl.-Ing. Uwe Friemann
Autor:	Alexander Walsemann
Abgabetermin:	22.03.2019

Thema der Masterthesis

Entwicklung eines Testsystems für den Physical- und Data-Link-Layer des PSI5-Busses mit automatisierter Auswertung auf Basis der Xilinx ZYNQ SoCs.

Kurzzusammenfassung

Die vorliegende Masterthesis beschreibt die Entwicklung eines Testsystems für die PSI5-Schnittstelle von ASICs und ASSPs. Zunächst werden anhand des PSI5-Standards die Eigenschaften des Physical- und Data-Link-Layers aufgezeigt, welche neben etwaigen Störgrößen relevant für die Entwicklung des Testsystems sind. Das anschließend entwickelte Testsystem besteht aus einem physical Layer und einem ZYNQ SoC, welcher programmierbare Logik (FPGA) und CPU-Kerne vereint. Die Kernfunktionen der Sensorsimulation zum Testen eines PSI5-Master-Interfaces sind in programmierbarer Logik umgesetzt, während Softwareapplikationen für den Testablauf und die automatisierte Auswertung der Ergebnisse verantwortlich sind. Die beiden CPU-Kerne des ZYNQ SoCs werden als ein asymmetrisches Multiprozessorsystem aus dem Echtzeitbetriebssystem FreeRTOS für zeitkritische Aufgaben und einem modifizierten Linux-Kernel genutzt. Die Bedienung des Testsystems erfolgt über ein Webinterface.

Title of the Thesis

Development of a Xilinx ZYNQ SoC based automated testsystem for the PSI5 bus physical and data link layer.

Abstract

This thesis documents the development of a testsystem for the PSI5 interface of ASICs and ASSPs. Initially the properties and features relevant to the development of the testsystem are demonstrated by reference to the PSI5 standard. The subsequently developed testsystem consists of a configurable physical layer and a Xilinx ZYNQ SoC, which combines programmable logic and a dual-core ARM CPU into a single SoC. The core functionality of the sensor-simulation used for testing PSI5 master interfaces is build using the programmable logic. In contrast the flow control of the test procedure and the evaluation of the results is performed by software applications. Both CPU cores are used in a asymmetric multiprocessing configuration. One cpu core is handling timing-critical tasks using the real-time operating system FreeRTOS, while the other is executing a modified Linux kernel. The testsystem is operated via a custom web interface.

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

Abkürzungsverzeichnis

ADC	Analog-To-Digital Converter
AMBA	Arm Advanced Microcontroller Bus Architecture
APU	Application Processor Unit
ASIC	Application Specific Integrated Circuit
ASIL	Automotive Safety Integrity Level
ASSP	Application Specific Standard Product
ATE	Automatic Test Equipment
AXI	Advanced eXtensible Interface
BGA	Ball Grid Array
CAN	Controller Area Network
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRC	zyklische Redundanzprüfung (Cyclic Redundancy Check)
CS	Chip-Select
DAC	Digital-To-Analog Converter
DSP	Digitaler Signalprozessor
DUT	Device Under Test
ELF	Executable and Linkable Format
eMMC	embedded Multimedia Card
FPGA	Field Programmable Gate Array
FSBL	First Stage Boot Loader
GPIO	General Purpose Input/Output
GPO	General Purpose Output
HIL	Hardware Interface Layer
I2C	Inter-Integrated Circuit
IDE	integrierte Entwicklungsumgebung (Integrated Development Environment)
ISR	Interrupt Service Routine
JSON	JavaScript Object Notation
JTAG	Joint Test Action Group
KFZ	Kraftfahrzeug
LED	Light-Emitting Diode
LIN	Local Interconnect Network
MISO	Master Input, Slave Output
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MOSI	Master Output, Slave Input
PAS	Peripheral Acceleration Sensor
PLL	Phase Locked Loop
PSI	Peripheral Sensor Interface
RAM	Random-Access Memory
remoteproc	Remote Processor Framework
rpmsg	Remote Processor Messaging Framework
RTC	Real-Time Clock
SCK	Serial Clock
SDK	Software Development Kit

SoC	System-On-Chip
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
TDMA	synchrones Zeitmultiplexverfahren (Time Division Multiple Access)
TFTP	Trivial File Transfer Protocol
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus

Inhaltsverzeichnis

1 Einleitung

Die Elmos Semiconductor AG ist ein Halbleiterhersteller, welcher seit über 30 Jahren Halbleiter und Sensoren entwickelt, produziert und testet. Die Entwicklungen der Elmos Semiconductor AG werden heute vornehmlich von Kunden aus der Automobilbranche eingesetzt. Ein Teil dieser Entwicklungen sind PSI5-Controller, welche als ASSP im Produktkatalog der Elmos Semiconductor AG verfügbar sind und für welche im Rahmen dieser Arbeit ein Testsystem entwickelt werden soll.
Der PSI5-Bus ist ein 2-Draht Bussystem, welches in der Automobilindustrie eingesetzt wird. Er wird im Kraftfahrzeug als Schnittstelle zwischen dezentralen Sensorsatelliten und einem zentralen Steuergerät eingesetzt. Das Alleinstellungsmerkmal des PSI5-Busses liegt in der Fähigkeit des Bussystems mit zwei Leitungen zu einem Sensor bzw. Slave auszukommen. Andere ebenfalls als 2-Draht Bussysteme bezeichnete Datenbusse beziehen diese Angabe nur auf die Anzahl der benötigten Datenleitungen und können ohne zusätzliche Leitungen für die Spannungsversorgung oder Referenzmasse nicht betrieben werden. Der PSI5-Bus hingegen nutzt eine 2-Draht-Leitung als kombinierte Leitung, sowohl für den Datentransfer, als auch für die Versorgung der Sensoren bzw. Slaves mit Spannung.
Dazu werden bei einem PSI5-Bus die Daten von dem Sensor durch eine geschaltete Last innerhalb seiner PSI5-Schnittstelle als Stromimpulse moduliert (siehe Abbildung 2.3). Die generierten Stromimpulse auf den Versorgungsleitungen werden von dem Steuergerät, welches die Funktionen des Bus-Masters übernimmt, ausgewertet, sodass die Daten zurückgewonnen werden können. Dieses Verfahren ist dabei durch den PSI5-Standard mit einer Datenrate von bis zu $189\,kbit/s$ spezifiziert.
Dieser Standard wurde gemeinsam von dem Firmen Autoliv, Bosch und Continental als Zusammenschluss in der PSI5-Organisation entwickelt. Er ist als offener Standard verfügbar und kann ohne Lizenzabgaben kommerziell genutzt werden. Der PSI5-Organisation gehören heute neben den Gründungsmitgliedern viele namhafte Automobilzulieferer und Halbleiterhersteller wie die Elmos Semiconductor AG an.
Der PSI5-Standard spezifiziert den selbigen Bus als bidirektionalen Bus, welcher auch die Übertragung von Daten vom Master zum Slave ermöglicht. Dieser Datentransfer vom Steuergerät zum Sensor erfolgt über Spannungsimpulse auf der Versorgungsleitung, welche vom Steuergerät gespeist werden. Durch die Variation der Länge der Spannungsimpulse, oder das Auslassen dieser, werden die Daten kodiert. Die Datenrate ist jedoch aufgrund der höheren Periodendauer der Spannungsimpulse, um ein vielfaches geringer als die Datenrate vom Sensor zum Steuergerät.
Der PSI5-Bus liefert im Vergleich zu ebenfalls in der Automobilindustrie eingesetzten Bussystemen eine geringe Datenrate (siehe Abbildung 1.1). Der Fokus dieses Bussystems liegt jedoch in der Integration von einzelnen Sensoren mit geringer bis mittlerer Datenrate

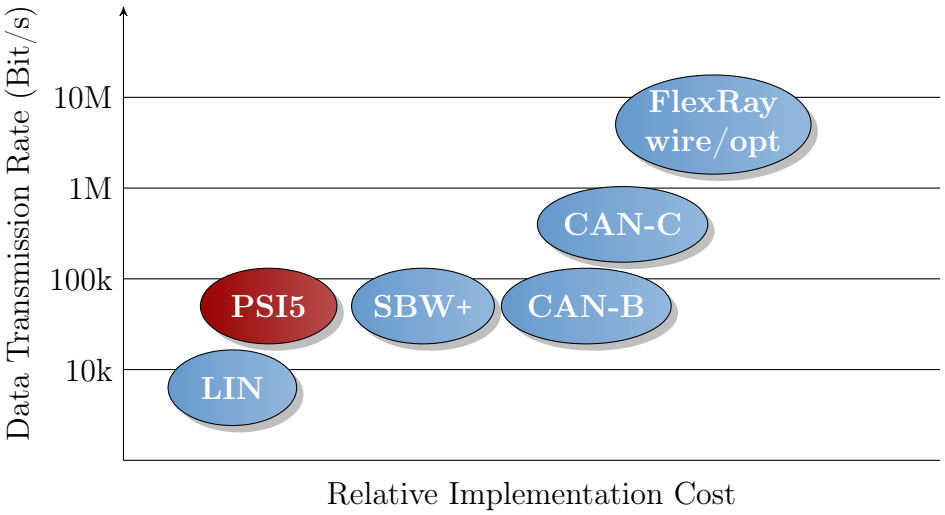


Abb. 1.1: Vergleich unterschiedlicher Bussysteme, modifiziert nach [PSI16]

bei geringen Implementierungs- und Verdrahtungskosten. Zum Vergleich kommt der Local Interconnect Network (LIN)-Bus mit nur einer dedizierten Datenleitung theoretisch mit weniger Verdrahtungsaufwand aus, benötigt jedoch eine separate Spannungsversorgung. Außerdem bietet er bei diesem bedingt geringerem Verdrahtungsaufwand eine Datenrate, die um Faktor zehn geringer ist als die des PSI5-Busses.
Eine ähnliche Datenrate wie die des PSI5-Busses bietet der Controller Area Network (CAN)-BUS, welcher jedoch mit zwei Signalleitungen, sowie den zwei Versorgungsleitungen, einen deutlich größeren Kabelbaum benötigt und einen komplexeren Protokollstack verwendet. Dies führt in der Praxis zu einem erhöhten Designaufwand und größeren Chipflächen, welche den wirtschaftlichen Einsatzbereich für einfache Sensoren mit geringem Datenaufkommen beschränken. Zusätzlich ist der Einsatz des CAN-Busses in sicherheitskritischen Systemen, wie Insassenrückhaltesystemen, nicht möglich, da die Datenübertragung nicht deterministisch ist. Die Übertragungsreihenfolge der Daten auf einem CAN-Bus mit mehreren Teilnehmern ist variabel, da sie von der Priorität der auf dem CAN-Bus vorhandenen Datenpakete abhängig ist. Die Verteilung der Zugriffsrechte auf den PSI5-Bus erfolgt hingegen durch ein Zeitmultiplexverfahren, sodass jedem Sensor ein fester Zeitslot zugeordnet ist, innerhalb welchem er seine Daten an das Steuergerät senden darf. Dieses Verfahren gewährleistet, dass es zu keinen Kollisionen auf dem Bus kommt.
Die von der Elmos Semiconductor AG entwickelte PSI5 Zelle steht neben der Implementierung als PSI5-Controller auch als IP-Block für weitere Designs selbiger Firma zur Verfügung. Neben einer erstmaligen Verifikation des PSI5-Controllers müssen bei der Integration dieses IP-Blocks als Bestandteil eines höher integrierten ASICs mit ausgedehnter Funktionalität mögliche Wechselwirkungen untersucht werden. Diese Tests und Verifikationsschritte machen einen erheblichen Teil der Entwicklung eines ASICs aus und müssen für jede Revision oder Anpassung wiederholt werden. Die bisherigen Testsysteme zur

Untersuchung des PSI5-Busses bzw. etwaigen Controller-Implementierungen folgen zwei unterschiedlichen Ansätzen und bestehen aus manuell zu bedienenden Testplatinen für einzelne spezifische Aspekte des PSI5-Busses oder aus automatischen Testmaschinen. Die Testmaschinen haben hohe Anschaffungskosten und sind aufgrund ihrer Bauform, insbesondere den Abmessungen und dem Gewicht, nicht geeignet für den portablen Einsatz, wie er z.B. zur Unterstützung eines Applikations-Ingenieurs beim Kunden erfolgt. Ebenso ist der Aufwand für die Einrichtung eines einmaligen schnellen Test eines spezifischen Verhaltens durch den komplexen Aufbau der Testmaschine und deren Software hoch. Im Gegensatz zu den Testmaschinen sind die Testplatinen für einen portablen Einsatz geeignet, deren Bedienung und die Durchführung von Tests ist jedoch aufgrund des geringen Automatisierungsgrad sehr zeitintensiv.
Das neue Testsystem, dessen Hardware im Rahmen dieser Arbeit entwickelt wird, soll diese Defizite ausgleichen und eine portable Lösung mit einem höheren Automatisierungsgrad bieten. Damit soll ein guter Kompromiss zwischen den bestehenden Ansätzen erreicht werden.

1.1 Anwendung des PSI5-Busses

Der PSI5-Bus findet seinen primären Anwendungszweck im Bereich von Insassenrückhaltesystemen für Kraftfahrzeuge. Er ersetzt in modernen Fahrzeugen analoge oder einfache digitale pulsweitenmodulierte Signalwege, sowie die älteren Peripheral Acceleration Sensor (PAS)3 / PAS4 Protokolle. Diese wurden eingesetzt, um Insassenrückhaltesysteme durch zusätzliche dezentrale Sensoren, insbesondere Beschleunigungssensoren, zu erweitern.[ZS14, S. 46 ff.]
Um den gestiegenen Anforderungen an die Wirksamkeit von aktiven Sicherheitssystemen im Kraftfahrzeug zu entsprechen, werden von Automobilherstellern komplexe Systeme entworfen, welche eine Vielzahl an verschiedensten Sensoren, wie Beschleunigungssensoren, Drucksensoren und Aufprallsensoren, an verschiedenen verteilten Montagepositionen im Kraftfahrzeug benötigen. Als Anforderungen an das Bussystem wurde vor allem eine kostengünstige Realisierung gestellt, welche mit zwei Leitungen auskommt und so den Verdrahtungsaufwand, sowie das Gewicht im Kraftfahrzeug reduziert. Neben den Faktoren zur Wirtschaftlichkeit müssen jedoch vor allem die sicherheitstechnischen Aspekte gewährleistet werden. Das Bussystem muss unter einer Vielzahl an variierenden Umwelteinflüssen äußerst fehlersicher und deterministisch sein.
Durch ihren Einsatzzweck findet bei einem ASIC mit einer PSI5-Schnittstelle die ISO 26262 Norm („Road vehicles – Functional safety“) häufig Anwendung, welche Methoden für die Entwicklung und Herstellung von sicherheitsrelevanten elektrischen Systemen in Kraftfahrzeugen definiert. Typischerweise wird ein Insassenrückhaltesystem nach der ASIL-Risikoklassifikation mit dem höchsten (ASIL-D) oder zweithöchsten Wert (ASIL-C) bewertet. Dementsprechend aufwändig fallen auch die Test- und Verifikationsaufgaben während der Entwicklung aus, um für den späteren Betrieb eine geringe Ausfallwahrscheinlichkeit und deterministisches Verhalten im Fehlerfall garantieren zu können.
Während der PSI5-Standard ursprünglich vom PSI5 Steering Committee für den Einsatz in Insassenrückhaltesystemen entwickelt wurde, gibt es mittlerweile eine Trennung der Standards in eine Basisspezifikation, sowie verschiedene applikationsspezifische Erweiterungen. Diese Trennung soll den unterschiedlichen Einsatzbedingungen der neuen Bereiche, wie Antriebs- und Fahrwerkstechnik, gerecht werden. Diese Bereiche haben in den letzten Jahren eine kontinuierliche Steigerung der eingesetzten Sensorik erfahren und der PSI5-Bus stellt eine geeignete Möglichkeit dar bei geringem Verdrahtungsaufwand Sensorsatelliten in das Gesamtsystem des Kraftfahrzeugs zu integrieren.[WGO10] Gemein bleibt den Anwendungsbereichen jedoch, dass der PSI5-Bus äußerst zuverlässig sein muss, da dieser oftmals die Übertragung von Daten für sicherheitskritische Anwendungen übernimmt.

1.2 Ziel der Arbeit

Ziel der Arbeit ist die Entwicklung eines Systems, welche die PSI5-Schnittstelle von bis zu zehn PSI5-Bussen simultan testen kann. Da bei den vorliegenden ASICs eine maximale Teilnehmeranzahl von vier Teilnehmern je Bus vorgesehen ist, müssen die Schnittstellen von bis zu 40 Sensoren bzw. Slaves für die Tests simuliert werden können. Der Ablauf und die Auswertung der Tests soll dabei automatisiert ablaufen, um den Aufwand für die Durchführung der Tests gering zu halten und eine hohe Reproduzierbarkeit der Ergebnisse zu erreichen.
Die digitalen Funktionalitäten der Sensoren, sowie weitere Funktionen werden in einem FPGA und / oder Mikrocontroller abgebildet. Die notwendige Umwandlung der digitalen Steuer- und Datensignale in das eigentliche PSI5-Bussignal wird von der Hardware des Testsystems durchgeführt. Eine zentrale Anforderungen an das Testsystem besteht darin, die verschiedensten Parameter einer Kommunikation über den PSI5-Bus innerhalb und jenseits der von der PSI5 Spezifikation vorgegebenen Grenzen automatisiert variieren zu können. Zu diesen Parametern gehören unter anderem die Stromaufnahme und der Datenwortstrompegel eines simulierten Sensors, aber auch der digitalen Domäne zugeordneten Parameter, wie die Bitrate und Verzögerungszeiten.
Um die Robustheit der Implementierung der PSI5-Schnittstelle des zu testenden ASICs und mögliche Wechselwirkungen auf andere Komponenten des ASICs zu untersuchen, sollen sämtliche Kanäle entweder einzeln oder simultan gestört werden. Das Testsystem soll mit der für die Simulation der typischen im Einsatz zu erwartenden Störungen, wie Kurzschlüsse gegen Versorgungs- oder Masseleitungen, benötigten Hardware ausgestattet sein und diese Fehler automatisiert einspeisen können. Weitere Fehlermechanismen sollen durch die Einspeisung mittels externer Instrumente untersucht werden können.
Die bisher genannten Anforderungen an die Ziele der Entwicklung eines Testsystems für den PSI5-Bus können auch durch die oben genannten Testmaschinen (ATE) erfüllt werden. Diese sind jedoch nicht portabel, haben hohe Anforderungen an den Aufstellort und benötigen komplexe, spezialisierte Software. Das zu entwickelnde System soll portabel sein, sodass als Randbedingung eine Begrenzung der Größe auf maximal zwei Europakarten und die Versorgung aus einer einfachen Spannungsquelle definiert wurde.
Die Anforderungen an die Portabilität beschränken sich jedoch nicht nur auf die physikalischen Eigenschaften des Systems, sondern auch auf die Softwareschnittstellen des Testsystems. Diese sollen möglichst einfach aufgebaut sein und durch typischerweise bereits vorhandene Software wie einen Texteditor oder Webbrowser bedienbar sein, sodass eine Installation von spezialisierter Software auf dem PC des Anwenders, welche gegebenenfalls mit erheblichen Lizenzkosten verbunden ist, entfällt.

2 Der PSI5-Bus

Der Peripheral Sensor Interface (PSI)5-Standard definiert für den PSI5-Bus einen physical Layer, einen Data-Link-Layer und einen Application Layer. Die zum Zeitpunkt dieser Arbeit aktuellste Version des PSI5-Standards ist die Version 2.3. Neben dem Hauptstandard sind Erweiterungen für "Airbag", "Chassis and Safety" und "Drivetrain" Komponenten verfügbar.[Rob18]
Der Data-Link-Layer wird bei dem Testsystem durch die Schaltungen und Hardwarekomponenten der Hauptplatine implementiert. Er wird durch logische Funktionen in einem Field Programmable Gate Array (FPGA) abgebildet. Der Application Layer spielt für das Testsystem nur eine untergeordnete Rolle, da dieser die Nutzdaten einer Übertragung vorgibt, diese jedoch beim vorliegenden Testsystem direkt durch den Anwender im Testablauf spezifiziert werden. Das Testsystem ist primär auf den Test des physical Layers und des Data-Link-Layers ausgelegt und prüft keine Funktionen des Application Layers automatisch. Der Anwender kann allerdings durch eine geeignete Wahl von zu übertragenden Nutzdaten und dem Vergleich mit entsprechenden Erwartungswerten durch das Testsystem eine Prüfung des Application Layers eines Device Under Test (DUTs) vornehmen. Die Abbildung 2.1 zeigt das Konzept des PSI5-Busses und den schematischen Aufbau des physical Layers für eine Sensor- und Steuergerät-PSI5-Schnittstelle. Die Spezifikation des

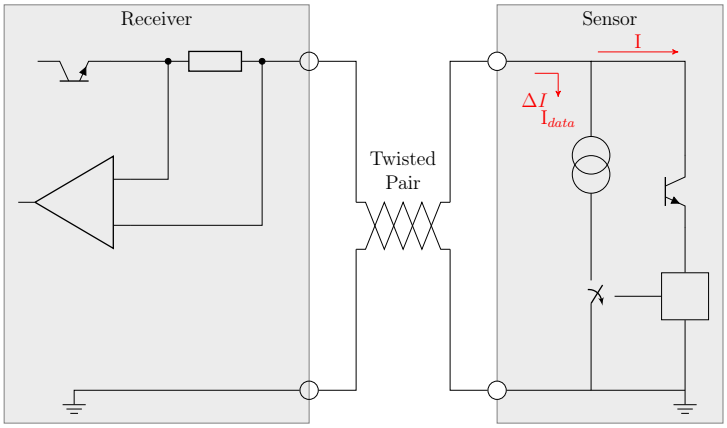


Abb. 2.1: Schematische Darstellung des PSI5-Bus Konzeptes, modifiziert nach [PSI16]

physical Layers des PSI5-Busses ist so ausgelegt, dass bei Verwendung von ebenfalls für Controller Area Network (CAN)-Busse üblichen twisted-Pair-Leitungen die in der Abbildung 2.1 gezeigte Leitung zwischen Sensor und Steuergerät eine Länge von bis zu zwölf Metern aufweisen kann.
Die Länge der Nutzdaten ist variabel und kann in einem einzelnen Frame zwischen 10 und 28 *Bit* betragen, wobei die Nutzdaten noch durch zwei Startbits, ein Parity-Bit oder CRC-Bits ergänzt werden.

Die folgenden Abschnitte dieses Kapitels 2 geben einen Überblick auf die Anforderungen und Eigenschaften der einzelnen Bestandteile eines PSI5-Busses. Dabei enthalten die dargestellten Abschnitte jedoch nur die für die Entwicklung des Testsystems relevanten Eigenschaften der PSI5-Spezifikation, denn auch die zu testenden Application Specific Integrated Circuits (ASICs) oder Application Specific Standard Products (ASSPs) erfüllen nur eine Untermenge des gesamten, im PSI5-Standard definierten, möglichen Funktionsumfangs.

2.1 Aufbau / Physical Layer

Da die Übertragung je nach Datenrichtung sowohl in ihrer Art (Strom oder Spannungsmodulation), als auch in der maximal möglichen Datenrate stark variiert, unterscheidet sich der Aufbau der physical Layer von Sensor und Steuergerät. Die in der Tabelle 2.1 nachfolgenden Daten spezifizieren die Grundeigenschaften des PSI5-Busses, welche von sämtlichen Teilnehmern des Busses eingehalten werden müssen, auch wenn diese je nach Teilnehmer in unterschiedlichster Funktionalität umgesetzt werden.

Tabelle 2.1: Ausgewählte Parameter des PSI5-Busses

Parameter	**Symbol**	**Bemerkung**	**Min.**	**Typ.**	**Max.**	**Einheit**
Supply Voltage	$V_s s$	Standard	5,0		16,5	V
		Low Voltage	4,0		16,5	V
Supply Quiescent Current	I_s	Standard	4,0		19,0	mA
		Extended Mode	4,0		19,0	mA
Current Pulse	$I_\Delta s$	Standard	22,0	26,0	30,0	mA
		Low Power	11,0	13,0	15,0	mA
Bitrate	$f_b it$		125,0		189,0	kHz

2.1.1 ECU / Steuergerät

Der physical Layer des Steuergerätes muss mehrere Funktionen parallel erfüllen. Er ist verantwortlich für die Versorgung der Sensoren mit der benötigten Betriebsspannung, die Modulation selbiger zur Datenübertragung, sowie der Auswertung des Sensorstromes zur Datenrückgewinnung. Neben der eigentlichen Spannungsversorgung verfügt der Versorgungsblock typischerweise über zusätzliche Schutzfunktionen. Diese schützen den Bustreiber gegen Überströme bei einem Kurzschluss der Signalleitungen mit den Betriebsspannungen im Kraftfahrzeug.
Die Modulation der Versorgungsspannung wird eingesetzt, um einen Sync-Impuls zu erzeugen. Dieser bestimmt den Startzeitpunkt der Zeitslots für den Datentransfer der Sensoren. Zusätzlich können über das Verlängern der Impulsdauer oder das selektive Auslassen selbiger, Informationen vom Steuergerät zum Sensor transferiert werden. Siehe dazu auch Abschnitt 2.4.
Kritisch ist dabei der Verlauf der Spannung eines Synchronisationsimpulses, welcher einige Parameter im Bezug auf die Dauer und Amplitude des Spannungshub einhalten muss, vgl. Abbildung 2.2. Die Spannungsamplitude des Sync-Pulses beträgt für einen Standardpuls mindestens $3,5\,V$, kann jedoch bei einem „Reduced sync pulse“, welcher mit der Version 2.0 des PSI5 Standards eingeführt wurde, auf minimal $2,5\,V$ reduziert werden (vgl. Tabelle 2.2). Die maximale mögliche Amplitude ist dabei durch die maximale Versorgungsspannung Vce bzw. den Ruhespannungen ohne Impulse $Vce, base$ begrenzt.

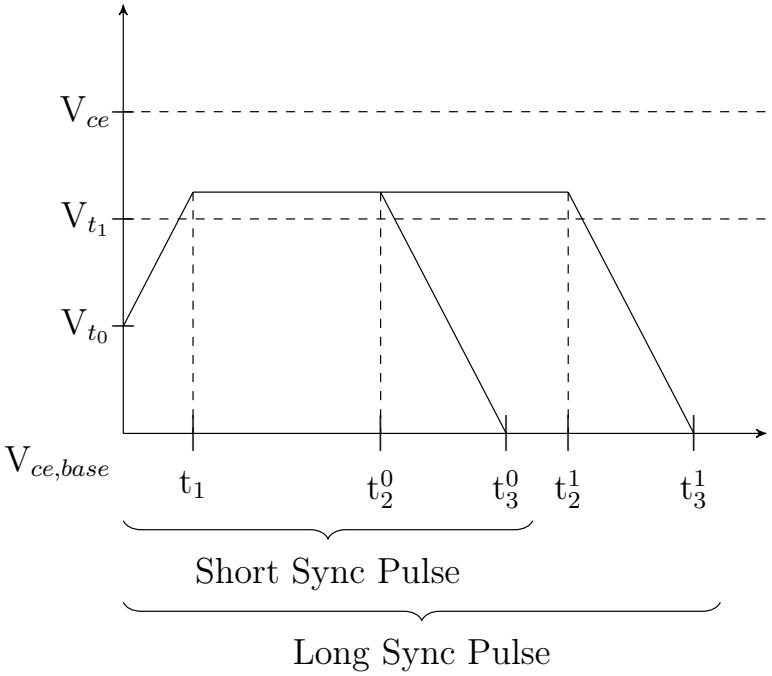


Abb. 2.2: Definition des Sync-Impulses, modifiziert nach [PSI18a, S. 35]

Tabelle 2.2: Parameter des Sync-Impulses, nach [PSI18a, S. 35 f.]

Parameter	**Symbol**	**Bemerkung**	**Min.**	**Typ.**	**Max.**	**Einheit**
Supply Voltage	V_{ce}				16,5	V
Slope Voltage	V_{t0}			V_{ss}+0,5		V
Sustain Voltage	V_{t1}	Reduced Pulse	V_{ss}+2,5			V
		Standard Pulse	V_{ss}+3,5			V
Sustain Start	t_1				7	µs
Sustain End	t_2^0	Short Pulse	16			µs
	t_2^1	Long Pulse	43			µs
Discharge Limit	t_3^0	Short Pulse			35	µs
	t_3^1	Long Pulse			62	µs

Impulse die länger als t_3^0 andauern, werden von den Sensoren, sofern unterstützt, nicht nur als Synchronisationsimpuls, sondern ebenfalls als logische Eins im Rahmen der Datenübertragung angesehen (vgl. Abschnitt 2.4).

Um die von den Sensoren als Stromimpulse modulierten Daten zu empfangen, wird im physical Layer des Steuergerätes die Stromaufnahme des Busses gemessen und ausgewertet. Für die Differenzierung zwischen einer logischen Eins und Null, besitzt der Empfänger eine dynamisch Entscheidungsschwelle. Diese passt sich an die Veränderungen des Ruhestromes an, um auf eine langsame Veränderung des Ruhestromes der Sensoren zu reagieren. Eine Variation der Ruhestromaufnahme entsteht durch die Aktivität der übrigen Elektronik eines Sensors. Die Anstiegs- und Abfallzeiten der Stromimpulse sind durch den Standard auf einen maximalen Wert von $1,8\,\mu s$ am Ausgang des Steuergerätes begrenzt.

Durch die eingesetzten Leitungen und deren Kapazitäten kommt es jedoch noch zu einem weiteren Verschleifen der Signalflanken. Durch die Spezifizierung der maximal zulässigen Anstiegs- und Abfallzeiten, sowie den Eigenschaften der Leitung durch ein Leitungsmodell wird garantiert, dass in den als Stromimpuls modulierten Daten stets genügend hochfrequente Anteile vorhanden sind. [PSI18a, S. 33] Diese werden benötigt, um eine zuverlässige Detektion der Impulse durch den Komparator eines physical Layers gegenüber einer vergleichsweise langsamen Variation der Ruhestromaufnahme zu ermöglichen.
Die Sensordaten sind während der Übertragung zu dem Steuergerät bzw. dem Master des Busses manchesterkodiert, um eine höhere Widerstandsfähigkeit des PSI5-Busses gegen äußere Störungen zu erreichen. Der Manchestercode ist ein Leitungscode, welcher den Takt im Ausgangssignal rekonstruierbar beibehält (vgl. Abbildung 2.3). Die Daten

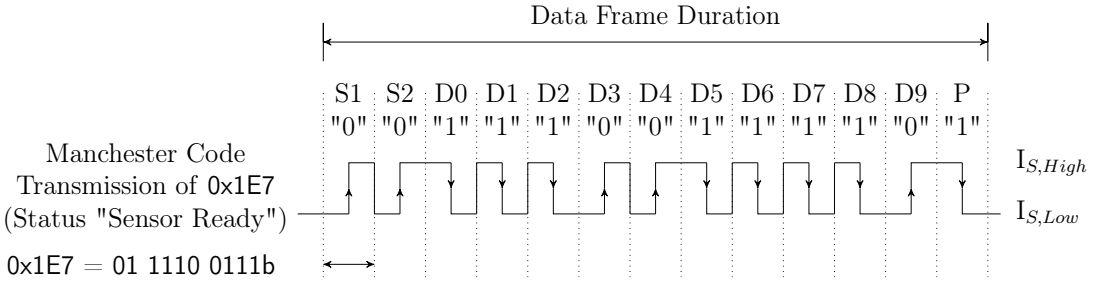


Abb. 2.3: PSI5 Beispielübertragung, modifiziert nach [PSI18a, S.9]

werden dabei bitweise durch die binäre Phasenlage des Taktes moduliert. Die eingesetzte Manchesterkodierung der zu übertragenden Daten bzw. der Stromimpulse ermöglicht die Betrachtung des Einflusses der Datenübertragung auf die Stromaufnahme als ein datenwortunabhängiger Offsetstrom. Die eigentliche Entscheidung zwischen einer logischen eins und null erfolgt anschließend durch einen Komparator, welcher die gegenwärtige Stromaufnahme des PSI5-Busses mit der Entscheidungsschwelle vergleicht. Sämtliche folgenden Schritte der Signalverarbeitung arbeiten auf Basis der resultierenden digitalen binären Signale.
Durch den Einsatz der Manchesterkodierung der Datenübertragung wird neben den eigentlichen Daten des Sensors auch ein Taktsignal von selbigem simultan übertragen. Dem empfangenden Steuergerät ist es daher möglich, sich auf einen aus der Manchesterkodierung zurückgewonnen Takt zu synchronisieren. Diese Synchronisierung verhindert einen Einfluss der Abweichungen in Phase und Frequenz zwischen dem Takt des Sensors und dem des Steuergerätes auf die Erkennung der übertragenen Daten.

2.1.2 Sensor

Der physical Layer des Sensors bildet das Gegenstück des PSI5-Busses zur Hardware des Steuergerätes. Er muss die Fähigkeiten besitzen etwaige Spannungsimpulse des Steuergerätes auszuwerten und im Gegenzug seine zu versendenden Daten als Stromimpulse zu modulieren. Neben dem physical Layer muss auch das restliche Design eines Sensors mit PSI5-Interface bezüglich seiner elektrischen Eigenschaften an die Schnittstelle angepasst werden, da die Spannungsversorgung über den PSI5-Bus erfolgt. Dies gilt insbesondere für die Welligkeit der Stromaufnahme, den absoluten Wert selbiger, sowie die äquivalente Kapazität des Sensors bzw. seines PSI5-Interfaces.
Die Versorgungsspannung des PSI5-Busses ist durch keinen festen Wert definiert, sondern deckt einen Spannungsbereich mit einer Größe von über $5\,V$ ab. Zeitgleich definiert der PSI5-Standard einen Spannungsimpuls, welcher von dem Steuergerät erzeugt wird, nur durch eine minimale Amplitude. Dies hat zur Folge, dass keine feste Entscheidungs-

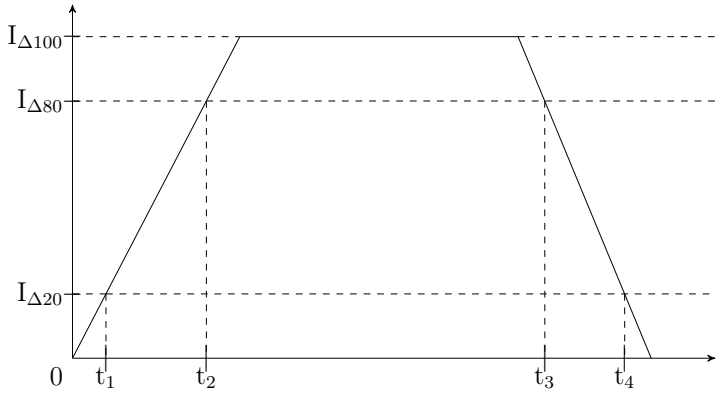


Abb. 2.4: Definition des Stromimpulses, modifiziert nach [PSI18a, S. 33]

schwelle zur Detektierung des Spannungsimpulses verwendet werden kann, da andernfalls die Möglichkeit besteht, dass Schwankungen der Versorgungsspannung irrtümlich als Sync-Impulse interpretiert werden. Die Erkennung des Sync-Pulses muss zeitnah und oh-

Tabelle 2.3: Parameter des Stromimpulses, nach [PSI18a, S. 33]

Parameter	**Symbol**	**Min.**	**Typ.**	**Max.**	**Einheit**
Rise Time	$t_2 - t_1$			1,8	µs
Fall Time	$t_4 - t_3$			1,8	µs

ne großen Jitter erfolgen, damit eine Zuordnung der Sensoren zu den ihnen zugehörigen Timeslots der Übertragung zuverlässig erfolgen kann. Siehe dazu auch Abschnitt 2.2.2.
Die Erzeugung der Stromimpulse erfolgt durch eine steuerbare Stromsenke innerhalb des PSI5 physical Layers des Sensors. Eine Variation der Amplitude des Stromimpulses ist dabei nicht vorgesehen. Die Amplitude ist auf einen nach dem PSI5-Standard definierten Wert voreingestellt (siehe Abbildung 2.4 und Tabelle 2.3). Aufgrund der hohen Amplitude

des Stromimpulses von mindestens $11\,mA$ im low power mode oder $22\,mA$ im common mode wird eine hohe Störfestigkeit und eindeutige Identifizierbarkeit der Daten einer Übertragung zum physical Layer des Steuergerätes gewährleistet.[PSI18a, S. 27]

2.2 Topologie

Der PSI5-Bus unterstützt verschiedenste Topologien, um den vielfältigen Anforderungen eines Einsatzes im Kraftfahrzeug gerecht zu werden. Die Umsetzung dieser ist dabei einhergehend mit unterschiedlichen Betriebsmodi der PSI5-Schnittstelle und Variationen im Protokollablauf. Diese umfassen asynchrone Point-to-Point Verbindungen mit nur einem Sensor, einen Bus mit paralleler Verdrahtung der Sensoren oder einen Bus mit serieller Verschaltung (Daisy Chain). Eine direkte Klassifizierung der Topologie anhand des Betriebsmodus ist dabei nicht möglich, da zum Beispiel im Parallelbusbetrieb Stern- und Linientopologie der Verdrahtung vorliegen können. Im Daisy-Chain-Betrieb des Busses ist von einer Linientopologie auszugehen. Die ASICs, welche mit dem Testsystem, das im Rahmen dieser Arbeit entsteht, geprüft werden sollen, nutzen den synchronen parallelen Betriebsmodus. Das Testsystem kann in seiner gegenwärtigen Auslegung daneben auch den asynchronen Betriebsmodus bedienen. Lediglich der Daisy-Chain-Betrieb ist derzeit nicht möglich und würde weiterer Modifikationen bedürfen.

2.2.1 Asynchroner Betriebsmodus

Im asynchronen Betriebsmodus sind keine Sync-Impulse vorhanden. Der Sensor initiiert den Datentransfer bei Bedarf, oder gegebenenfalls durch einen eigenen Timer. Die vollständige Kontrolle über den Zeitpunkt der Datentransfers liegt beim Sensor. Dieser Betriebsmodus ist daher nur für eine Point-to-Point-Verbindung zwischen dem Steuergerät und einem Sensor geeignet, da andernfalls Kollisionen auf dem Bus durch simultane Datentransfers mehrerer Sensoren nicht ausgeschlossen werden können (siehe Abbildung 2.5).

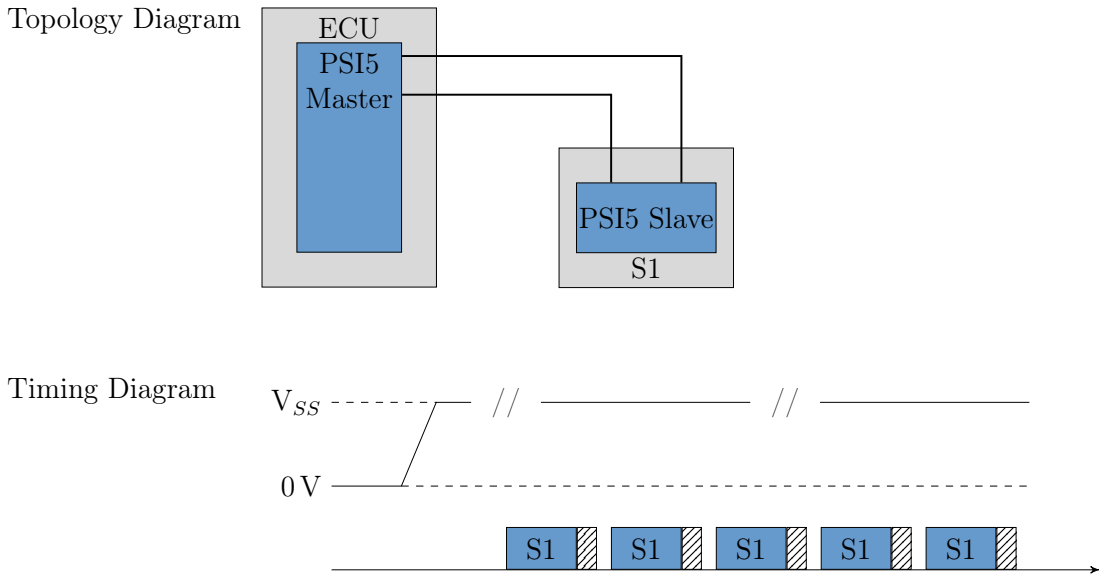


Abb. 2.5: Point-to-Point-Verbindung, modifiziert nach [PSI18a, S. 56]

2.2.2 Synchroner Betriebsmodus

Der synchrone Betriebsmodus verwendet die vom Steuergerät erzeugten Spannungsimpulse, um sämtliche Sensoren zu synchronisieren und ihnen den Startpunkt eines Transferzyklus zu übermitteln (siehe Abbildung 2.6). Die Dauer eines Transferzyklus $Tsync$ kann

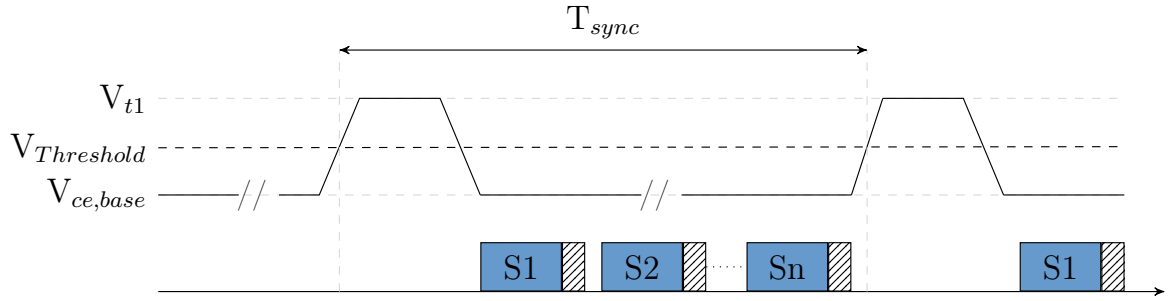


Abb. 2.6: Synchroner Betriebsmodus mit Sync-Impulsen, modifiziert nach [PSI18a, S. 57]

dabei, wie zum Beispiel bei der Sensorik für Insassenrückhaltesysteme verwendet, ein fester Wert sein oder variabel nach jedem Zyklus angepasst werden. Die variable Anpassung ist dabei unter anderem für die Sensorik des Antriebsstranges vorgesehen, um dort die Zyklusdauer an die Drehzahl des Motors anzupassen. Durch den Sync-Impuls kann neben der Synchronisierung der Transferzyklen auch der Samplezeitpunkt der Sensoren synchronisiert werden, wobei dies von der Implementierung des entsprechenden Sensors abhängig ist und separat betrachtet werden muss.
Der synchrone Betriebsmodus erlaubt theoretisch den Anschluss einer unbegrenzten Anzahl an Sensoren an ein Steuergerät mit nur einem PSI5-Interface. Mit jedem zusätzlichen

Sensor erhöht sich jedoch die minimal mögliche Zyklusdauer, sodass in der Praxis neben der akkumulierten Stromaufnahme der Sensoren, welche das Interface des Steuergeräts bereitstellen muss, ein weiteres begrenzendes Kriterium existiert. Der Sync-Impuls markiert den Startzeitpunkt eines Transferzyklus, innerhalb dessen die Sensoren zu definierten Zeitpunkten ihre Daten übertragen. Die Einteilung erfolgt dabei in einem synchronem Zeitmultiplexverfahren (Time Division Multiple Access, TDMA), welches jedem Sensor einen Zeitslot für den Transfer bereitstellt. Diese Zeitslots können dabei in der Länge variieren. Typischerweise sind jedoch, durch die applikationsspezifischen Erweiterungen des PSI5-Standards, identische Längen der Zeitslots vorgesehen. Eine Variation ihrer Länge wird im Betrieb nicht unterstützt.
Sensoren, die eine größere Payload erzeugen, können diesen, sofern die vollständige Übertragung innerhalb eines Transferzyklus gewünscht ist, über zwei Zeitslots verteilt an das Steuergerät senden. Eine Überlappung der beiden Zeitslots eines Sensors ist dabei möglich, ohne dass es zu Kollisionen der Daten kommt, da die beiden Zeitslots vom selben Sensor kontrolliert werden.

Parallele Verdrahtung

Bei der parallelen Verdrahtung der Sensoren an das PSI5-Interface des Steuergeräts wie in Abbildung 2.7 dargestellt, ist der synchrone Betriebsmodus vorgeschrieben. Typischerwei-

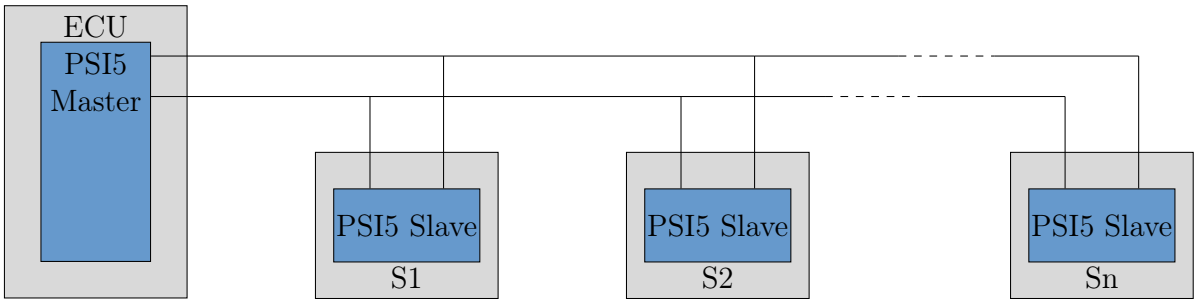


Abb. 2.7: Parallele Busverdrahtung, modifiziert nach [PSI18a, S. 57]

se wird z.B. gemäß der Unterspezifikation für Airbag Systeme eine Zykluszeit von $500\,\mu S$ verwendet, was bei einer Nutzdatengröße von $10\,Bit$ und einer Datenrate von $125\,kBit/s$ maximal drei Zeitslots bzw. bei $189 kBit/s$ vier Zeitslots entspricht.[PSI18b, S. 5] Die Adressierung der einzelnen Sensoren, sowie deren Zuordnung zu einem Zeitslot, muss dabei von den Sensoren organisiert werden. Sie müssen daher über einen internen Speicher für die Buskonfiguration oder eine externe Schnittstelle zur Konfiguration verfügen. Durch die festen Zeitslots der Sensoren innerhalb des TDMA wird gewährleistet, dass mit Beendigung eines Zyklus die aktuellen Daten eines jeden Sensors zum Steuergerät transferiert wurden.

2.3 Bit Coding / Data Link Layer

Der folgende Abschnitt bezieht sich ausschließlich auf den Datentransfer vom Sensor zum PSI5-Interface des Steuergerätes. Die Daten sind nach dem Manchester-II-Verfahren kodiert, deren fallende Flanke einer logischen Eins entspricht. Die steigende Flanke repräsentiert eine logische Null. Der Start der Übertragung eines Frames wird durch zwei aufeinanderfolgende logische Nullen eingeleitet. Anschließend folgt die Payload von mindestens 10 *Bit* bis maximal 28 *Bit*. Den Abschluss des Frames bildet die Prüfsumme, welche aus einem Paritätsbit oder einer 3-Bit-CRC bestehen kann. Die sich ergebende maximale Gesamtlänge einer Übertragung beträgt 33 *Bit*.

2.3.1 PSI5-Frame

Der Frame besteht neben den Pflichtfeldern der Startbits, dem Datenbereich A und der Prüfsumme gegebenenfalls aus weiteren Status- und Datenbits (siehe Abbildung 2.8). Nach den Startbits kann der Frame um zwei Bits für den Serial Messaging Channel, wel-

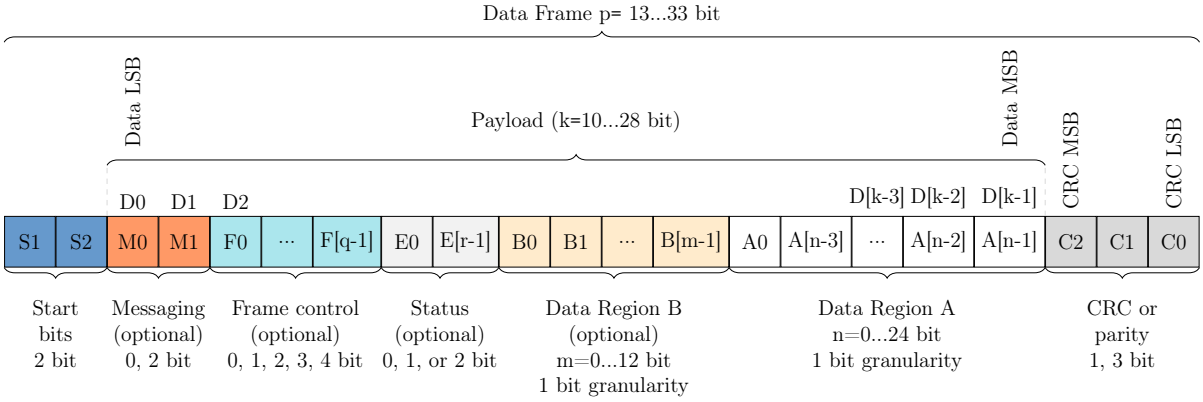


Abb. 2.8: PSI5-Frame, modifiziert nach [PSI18a, S. 10]

cher Daten über mehrere sequenzielle Nachrichten verteilt, erweitert werden. Das nächste optionale Feld ist das bis zu vier Bit umfassende Frame Control Feld, welches Informationen über den Typ der Nutzdaten oder die Adresse des Sensors beinhaltet. Das Statusfeld, welches ebenfalls nicht zu den Pflichtfeldern gehört, kann mit bis zu zwei Bit Status- und Fehlerinformationen des Sensors abbilden. Das Pflichtdatenfeld A kann in der Länge zwischen 10 *Bit* und 24 *Bit* variiert werden. Zusätzlich kann das optionale Datenfeld B in den Frame integriert werden. Dieses kann zwischen 1 *Bit* und 12 *Bit* in dem Frame vor dem Pflichtdatenfeld A transferiert werden.

Die Granularität jeglicher Felder der Nutzdaten, d.h. auch der optionalen Status- und Datenfelder, ist, mit Ausnahme der Bits für den Serial Messaging Channel, ein Bit groß. Die minimale Größe einzelner Felder ist jedoch weiterhin gültig. Ebenfalls darf die maximale Länge der Nutzdaten 28 *Bit* wie bisher nicht überschreiten, sodass bei zusätzlichen Feldern die Datenfelder gegebenenfalls in ihrer Größe beschnitten werden müssen.

2.3.2 CRC / Parity

Die Prüfsumme zum Schutz der Daten kann entweder aus einem einzelnen Paritätsbit oder einer 3-Bit-CRC bestehen. Wenn die Prüfsumme aus einem Paritätsbit besteht, muss ein gerades Paritätsbit verwendet werden, welches die Startbits nicht einschließt, sondern nur die Payload absichert.
Die zyklische Redundanzprüfung (CRC) verwendet das Generatorpolynom $g(x) = x^3 + x + 1$, welches mit einem Initialisierungswert von `0x7` vorbelegt ist. Die Daten werden in ihrer Transferreihenfolge in das Schieberegister geschoben, d.h. beginnend mit dem niederwertigsten Bit. Zusätzlich werden die Daten durch drei abschließende Nullen erweitert. Ist die letzte abschließende Null in das Schieberegister geschoben, so ist die Berechnung der CRC abgeschlossen. Die drei für die Berechnung des Polynoms notwendigen Speicherstellen enthalten die drei Bit der CRC. Diese drei Bits werden, im Gegensatz zu den Nutzdaten, mit dem höchstwertigen Bit zuerst an den Frame angehängt.

2.3.3 Data Range

Dieser Abschnitt, welcher die Gültigkeitsbereiche, die vom PSI5-Standard definiert werden, darstellt, gehört formell gesehen zum Applikation Layer. Er wird hier exemplarisch aufgeführt, um einen Einblick in die typischen Nutzdaten zu geben, welche im Normalfall in den entsprechenden Regionen des Data-Link Layers zu finden sind. Für das eigentliche Testsystem bedeuten diese Gültigkeitsbereiche keine weiteren Einschränkungen oder Anforderungen, da der Anwender über seine Testvorgaben die Nutzdaten selbst festlegt und damit für den korrekten Einhalt der Gültigkeitsbereiche verantwortlich ist.
Die höchstwertigsten zehn Bits werden immer als Grundlage für die Wandlung des Gültigkeitsbereichs herangezogen. Die restlichen Bits des Datenfeldes A stehen dabei frei zur Verfügung. Die weiteren Felder des Frames unterliegen ebenfalls nicht der Bereichsbegrenzung. Dieser feste Gültigkeitsbereich ermöglicht eine Standardisierung und bessere Interoperabilität verschiedener Sensoren.

Tabelle 2.4: Data Range des PSI5-Protokolls, nach [PSI18a, S. 45f.]

Wert		Datentyp	Beschreibung
Dec	**Hex**		
+511	0x1FF	Status & Error Code	Statische Meldungen
.	.	.	.
.	.	.	.
+481	0x1E1	Status & Error Code	Statische Meldungen
+480	0x1E0	Sensor Output Data	Maximaler Sensordatenwert
.	.	.	.
.	.	.	.
-480	0x220	Sensor Output Data	Minimaler Sensordatenwert
-481	0x21F	Status Data 0b1111	Initialisierungsdaten
.	.	.	.
.	.	.	.
-496	0x210	Status Data 0b0000	Initialisierungsdaten
-497	0x20F	Block ID 16	Block IDs
.	.	.	.
.	.	.	.
-512	0x200	Block ID 1	Block IDs

Die zehn Bits ermöglichen die Darstellung von 1024 verschiedenen Werten. Der gültige Wertebereich für physikalische Sensordaten wird jedoch auf den Bereich Data Range 1, welcher zwischen -480 und $+480$ liegt, begrenzt. Werte größer als $+480$ werden als Data Range 2 bezeichnet und enthalten standardisierte oder applikationsspezifische Status- und Fehlermeldungen. Der Wertebereich ab -480 bis -512 wird als Data Range 3 für den Versand von Daten zur Initialisierung nach dem Einschalten des Busses verwendet, wobei nach dem Standard eine zusätzliche Trennung in Initialisierungsdaten und deren Identifikationsnummer erfolgt.

2.4 ECU-to-Sensor

Wie bereits im Überblick zum PSI5-Bus beschrieben, verwendet die Kommunikation vom Steuergerät zum Sensor spannungsmodulierte Signale. Dabei stehen zwei unterschiedliche Verfahren zur Verfügung. Das Tooth Gap Verfahren verwendet den periodischen Sync-Impuls zum Kodieren von Daten. Dabei wird eine logische Eins durch einen vorhandenen Sync-Impuls abgebildet, eine Null hingegen durch das Auslassen des Impulses. Dieses Verfahren ist nur bei einer konstanten Periode des Sync-Impulses möglich, da ansonsten eine Differenzierung zwischen einem ausbleibenden Sync-Impuls und einer verlängerten Periode nicht möglich ist.
Das zweite Verfahren nutzt verschiedene Sync-Puls Längen (Pulse Width Verfahren), um

die Informationen zu kodieren. Die Unterscheidung erfolgt dabei durch die Längen der Impulse. Eine logische Eins ist im Vergleich zur logischen Null durch einen längeren Impuls definiert. Eine Kodierung von mehr als einem Bit in einem Symbol bzw. Impuls ist durch den PSI5-Standard nicht vorgesehen. Im Gegensatz zum Datentransfer durch das Tooth Gap Verfahren, muss bei Verwendung des Puls Width Verfahrens keine Startbedingung für die Initialisierung eines Datentransfers erfüllt sein.
Beide Verfahren werden jedoch von dem aktuell zu untersuchenden DUT nicht unterstützt, sodass das Testsystem gegenwärtig keine Funktionen besitzt, um diese Art der Datenübertragung zu prüfen. Die Hardware des Testsystem ist jedoch so gestaltet, dass es in Zukunft möglich ist das System um diese Funktionalität zu erweitern. Dazu bedarf es einer Erweiterung der programmierbaren Logik und der Software des Testsystems zur weitergehenden Auswertung der Synchronisationsimpulse und deren Zuordnung.

3 Testsystem

Das Testsystem soll eingesetzt werden, um einen PSI5-Controller, welcher als anwendungsspezifisches Standardprodukt oder als Komponente einer höher integrierten anwendungsspezifischen Schaltung vorliegen kann, zu prüfen. Die Parameter, welche mit diesem Testsystem geprüft werden können, umfassen verschiedenste digitale und analoge Parameter des PSI5-Standards. Neben den Konformitätstests des Controllers gegenüber dem Standard, ist das Fehlerverhalten, welches der PSI5-Controller bei einer Verletzung der Spezifikation zeigt, eine bedeutsame Information bei dessen Untersuchung und unabdingbar, um die Robustheit des Gesamtsystems bestimmen zu können.
Das Testsystem muss damit nicht nur ausreichende Freiheiten bieten, um Situationen abseits der Spezifikation zu prüfen, sondern auch Fehlerquellen simulieren können. Diese Fehlerquellen sind ein elementarer Bestandteil, um eine Robustheitsprüfung eines einzelnen PSI5-Controllers, oder Untersuchungen von Wechselwirkungen zwischen mehreren PSI5-Controllern in einem ASIC durchführen zu können.
Betrachtet werden soll mit diesem Testsystem vor allem, ob der Controller etwaige Fehlerfälle, wie einen Kurzschluss der Datenleitungen gegen Masse oder die positive Versorgungsspannung, sowie zwischen den Datenleitungen einzelner Kanäle zuverlässig erkennt und dementsprechend reagiert. Die Bewertung der Testergebnisse kann dabei automatisch durch die Informationen, die das DUT bereitstellt, oder durch externe Messinstrumente, wie z.B. ein Oszilloskop erfolgen. Ein hoher Grad an Automatisierung verringert dabei die benötigte Zeit für die Wiederholung von Tests und ermöglicht die einfache Umsetzung von Regressionstests zwischen zwei Versionen des gleichen ASSPs oder ASICs.

3.1 Konzept

Die Testumgebung zum Prüfen des PSI5-Controllers eines Steuergerät-ASICs erfolgt über die Nachbildung von Sensoren mit einer PSI5-Schnittstelle. Diese werden im Testsystem in simulierter Form mit dem zu testenden PSI5-Controller verbunden. Das Testsystem kann dann vorprogrammierte Frames über den PSI5-Bus an den Controller senden, wenn es von diesem dazu aufgefordert wird. Anschließend werden die vom PSI5-Controller empfangenen Datenworte und Statusinformationen über eine separate Serial Peripheral Interface (SPI)-Schnittstelle ausgelesen und mit einem Erwartungswert verglichen.
Es können mehrere Sensoren, die sich einen PSI5-Kanal mit ihren Zeitslots teilen, durch eine einzelne Hardwareimplementierung eines Kanals nachgebildet werden, sodass die Topologie des PSI5-Busses nicht beachtet werden muss. Die für die Simulation des PSI5-Busses notwendige Implementierung eines Kanals besteht aus einem physical Layer, welcher die Auswertung und Erzeugung der Strom- und Spannungsimpulse übernimmt, sowie der

notwendigen Steuerlogik, die das Protokoll bzw. die Pattern des PSI5-Busses erzeugt. Die Simulationshardware muss in der Lage sein, den kombinierten Ruhestrom mehrerer virtueller Sensoren aufzunehmen und bis zu vier Zeitslots eines Kanals zu bedienen. Darüber hinaus werden mehrere Kanäle gleichzeitig, jedoch unabhängig voneinander betrieben, sodass das Testsystem eine parallele Verarbeitung vornehmen muss.
Die Implementierung des physical Layers für die Generierung der Ruhestromaufnahme und der Stromimpulse eines Datentransfers besteht aus einer Stromsenke, welche durch eine Digital-Analog-Wandlung ihre Werte erhält. Diese integrierte Digital-Analog-Wandlung ermöglicht eine Nachbildung unterschiedlichster Sensoren und ihrer typischen Ruhe- und Datenströme. Die Auswertung der Synchronisations-Spannungsimpulse erfolgt im physical Layer durch einen Komparator, dessen digitales Ausgangssignal direkt in der digitalen Logik, welche die Protokollnachbildung übernimmt, verarbeitet wird.
Eine messtechnische Erfassung und Bewertung der Signalverläufe des PSI5-Busses ist in diesem Testsystem nicht vorgesehen. Wird diese Art der Untersuchung benötigt, kann sie durch ein externes Messmittel, wie z.B. ein Oszilloskop, durchgeführt werden. Das Testsystem verfügt dazu über einen Ausgang, der ein geeignetes Signal zur Synchronisation der externen Messinstrumente bereitstellt.
Da insbesondere komplexere ASICs, welche den PSI5-Controller nur als eine Komponente enthalten, eine komplexe Konfiguration nach Systemstart benötigen und gegebenenfalls auch noch andere Funktionen des ASICs im Betrieb bedient werden müssen, ist die reine Nachbildung von PSI5-Sensoren bzw. dem PSI5-Protokoll in einem FPGA nicht ausreichend. Das Testsystem besteht daher aus zusätzlichen Softwarekomponenten, die den ASIC bedienen und die von ihm empfangenen Datenworte auswerten.
Zu den Softwarekomponenten gehört außerdem die Ablaufsteuerung, welche die vom Anwender spezifizierten Testschritte aus einer Datei ausliest und die entsprechenden Anweisungen an das Testsystem und das DUT gibt. Dazu können in jedem Testschritt eine Konfiguration, Anweisung, SPI-Datentransfer oder Warteoperationen definiert und gegebenenfalls mit einem Erwartungswert versehen werden, gegen welchen die Ablaufsteuerung die erhaltenen Daten vergleicht. Sollte es zu Abweichungen gegenüber dem Erwartungswert kommen, kann die Ausführung weiterer Testschritte unterbrochen werden oder nur ein Logeintrag über die Abweichung bzw. den Fehler angefertigt und die Ausführung der Testschritte fortgesetzt werden.
Die Bedienung der Ablaufsteuerung des Testsystems erfolgt über ein Userinterface, welches unabhängig von der eigentlichen Testanwendung ist, welche die einzelnen Testschritte durchführt. Das Userinterface erlaubt das Laden und Ausführen von Testprozeduren direkt über ein Webinterface, sodass von jedem gewöhnlichen Webbrowser über TCP/IP bzw. HTTP auf das Testsystem zugegriffen werden kann, ohne das spezielle Software benötigt wird.

3.1.1 FPGA / CPU Architektur

Das Testsystem setzt auf die Kombination eines FPGAs und eines Mikrocontrollers in einem System-On-Chip (SoC), um den unterschiedlichsten Anforderungen, die an das Testsystem gestellt werden, gerecht zu werden. Die Ablaufsteuerung und das Userinterface lassen sich gut durch sequentielle Software beschreiben und profitieren von bestehenden Softwarestrukturen, wie Betriebssystemen, Webservern usw., um den Entwicklungsaufwand gering zu halten.
Selbiges gilt jedoch nicht für die PSI5-Sensorsimulation und andere Hardwareschnittstellen, wie z.B. die SPI-Schnittstellen des Testsystems. Da das Testsystem in seiner aktuellen Auslegung bis zu zehn PSI5-Kanäle simultan und unabhängig voneinander prüfen soll, ist es sinnvoll diese Funktionalität in der programmierbaren Logik des FPGAs abzulegen. Durch die Abbildung der PSI5-Sensorsimulation in der programmierbaren Logik anhand entsprechender Schaltungsstrukturen, kann eine vollständige parallele Verarbeitung der einzelnen Kanäle stattfinden. Eine etwaige Realisierung in Software würde die Anforderungen an den Mikrocontroller um ein vielfaches steigern und gegebenenfalls die Untersuchung von Timing bezogenen Parametern unmöglich machen.
Die Programmierbarkeit der Logik lässt eine Anpassung der Anzahl der Schnittstellen an den jeweiligen ASIC zu, indem die entsprechenden Blöcke innerhalb des FPGAs dupliziert oder entfernt werden. Ebenso können beliebig weitreichende Modifikationen der logischen Funktionen der Schnittstelle durchgeführt werden, ohne die externen Hardwarekomponenten des Testsystems anpassen zu müssen. Sofern die Schnittstellen zwischen dem FPGA und dem Prozessor des SoCs unverändert bleiben, beschränken sich die notwendigen Änderungen auf das Modifizieren des Logikdesigns mittels einer Hardwarebeschreibungssprache, sodass der benötigte Aufwand einer Anpassung gering gehalten wird. Diese Schnittstellen, welche, wie in Abbildung 3.1 erkennbar, als Advanced eXtensible Interface (AXI)-Interface umgesetzt sind, sind die Brücken zwischen der Software, welche das Userinterface, die Ablaufsteuerung und das Verwalten der Testergebnisse übernimmt, und der programmierbaren Hardware-Logik. Die Auswertung der Daten des Testsystems ist, ebenso wie das Userinterface, nicht echtzeitkritisch, sondern kann im Gegensatz zur eigentlichen Testdurchführung zeitversetzt erfolgen. Dies senkt im Zusammenspiel mit der programmierbaren Logik die Anforderungen an die Verarbeitung und ermöglicht den Einsatz des nicht echtzeitfähigen Linux Kernels für einen Großteil der Aufgaben, die von der Software übernommen werden.
Die verbleibenden Aufgaben zur Ansteuerung des DUTs, welche echtzeitkritisch sind, sich jedoch nicht für eine Implementierung in der programmierbaren Logik anbieten, werden durch ein Echtzeitbetriebssystem bearbeitet. Im Rahmen dieser Arbeit findet dazu FreeRTOS Anwendung, um Konfigurationsinformationen an das DUT zu senden und dessen Watchdog zu bedienen. Diese Umsetzung setzt jedoch eine Trennung der Central Proces-

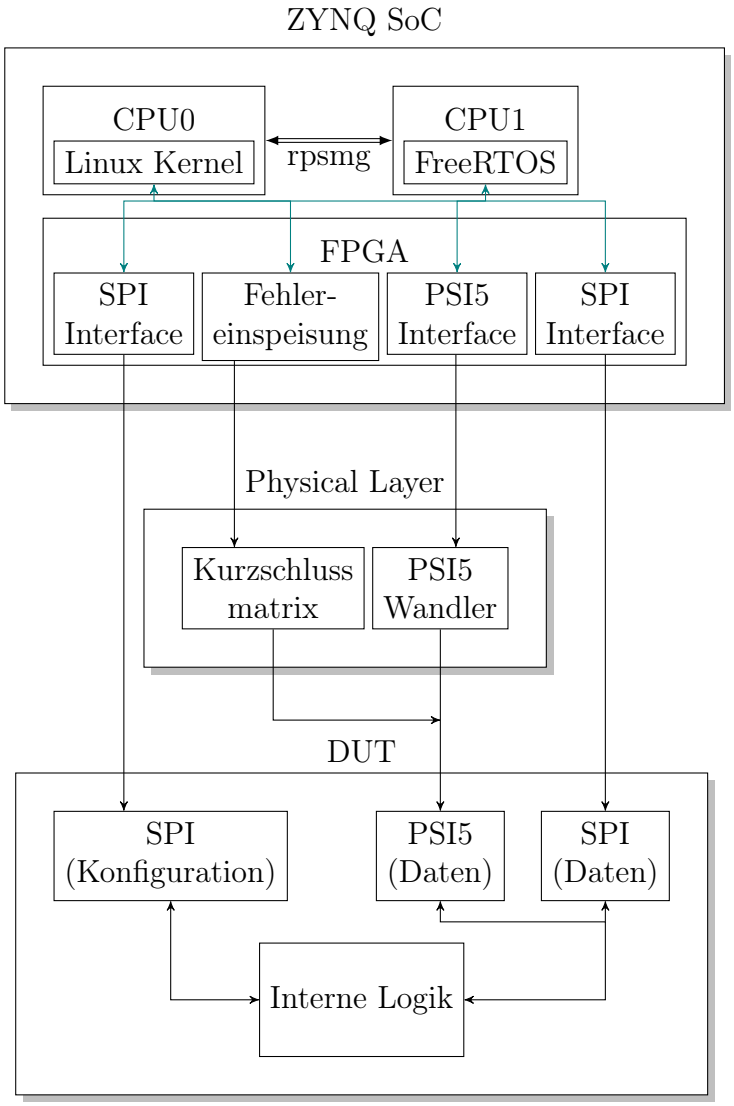


Abb. 3.1: Blockschaltbild der Testsystem Architektur

sing Unit (CPU)-Kerne und die explizite, exklusive Zuweisung von Ressourcen, insbesondere von Speicherbereichen, an die beiden ausgeführten Betriebssysteme voraus.
Die gewünschte Architektur, aus einem FPGA und einem Mikrocontroller mit mindestens zwei Kernen für die beiden Betriebssysteme, um die unterschiedlichen Aufgaben des Testsystems optimal bearbeiten zu können, führte zur Auswahl eines Xilinx ZYNQ SoCs als Kernstück des Testsystems. Dieser vereint die programmierbare Logik und CPU in einem SoC und stellt bereits intern Interfaces zwischen beiden Blöcken bereit.

3.2 ZYNQ SoC

Die Xilinx ZYNQ-7000 ist eine SoC-Reihe, welche, wie im Blockschaltbild der Abbildung 3.2 zu sehen ist, aus einem ARM Cortex A9 Prozessor und einer vollintegrierten 28 nm programmierbaren Logik besteht. Die Application Processor Unit (APU) bzw.

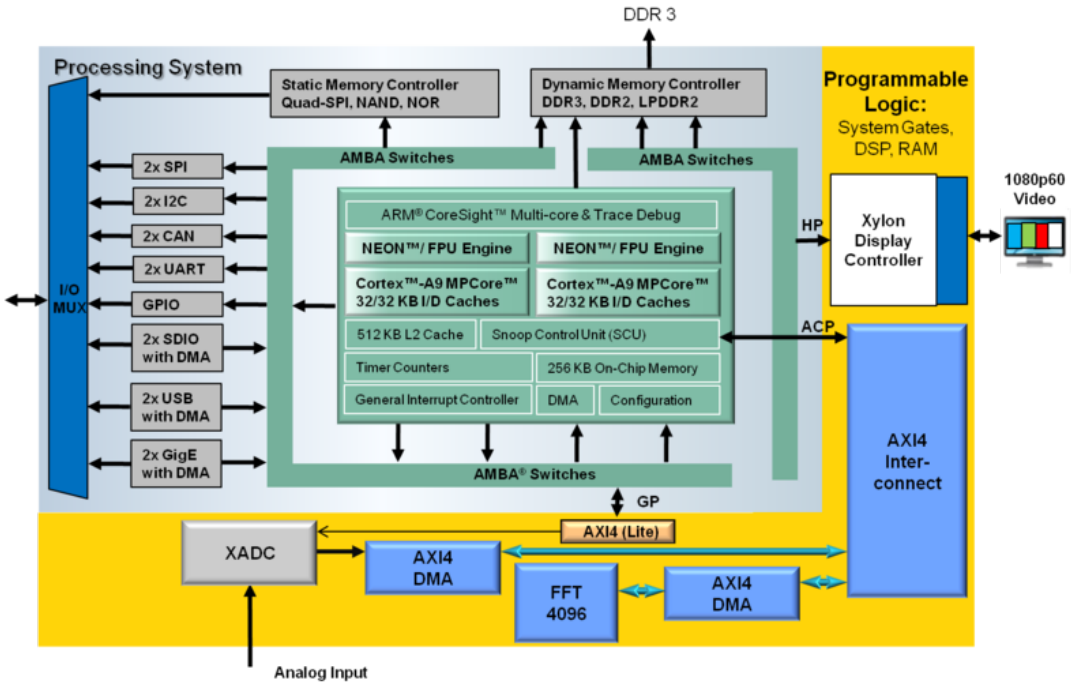


Abb. 3.2: Blockschaltbild der ZYNQ Architektur [Xil14]

CPU, die auf dem ARM Cortex A9 basiert, arbeitet mit einer Taktfrequenz von bis zu 1000 MHz, je nach eingesetzter Variante und besitzt 32 kB L1 Cache, sowie 512 kB L2 Cache, welchen sich beide Prozessorkerne teilen. Die CPU hat Zugriff auf eine Auswahl von I/O Peripheriecontrollern, unter anderem ein USB-Host-Controller, sowie Client-Controller, Gigabit-Ethernet, SD-Controller, UART, SPI und I2C. Die Verbindung mit den externen Pinnen des SoCs erfolgt, je nach gewünschter Konfiguration, über einen I/O Multiplexer, sodass ein größtmöglicher Freiheitsgrad des Hardwaredesigns erreicht wird. [Xil16]

Die programmierbare Logik enthält neben den eigentlichen Logikzellen, Digitalen Signalprozessor (DSP)-Blöcken und 36 kB Dual-Port Block-RAM Blöcken, 256 kB SRAM, der über einen Interconnect angesprochen wird. Ist dieser für die Anwendung nicht ausreichend, kann die programmierbare Logik über ein AXI-Interface auf den Speichercontroller für den externen DDR3-RAM zugreifen, welcher von der programmierbaren Logik und APU geteilt wird. Die im ZYNQ-7000 verwendeten AXI 4-Interfaces werden sowohl für die Kommunikation zwischen dem Prozessor und der programmierbaren Logik verwendet, als auch zur Kommunikation mit den I/O-Peripheriecontrollern. Die programmierbare Logik besitzt universelle 3,3 V und 1,8 V high-speed I/Os, welche von der APU unabhängig

sind, sowie einen integrierten 12 *Bit* Analog-To-Digital Converter. Dieser ADC kann über einen analogen Multiplexer für externe Aufgaben verwendet werden oder interne Parameter wie Spannungen und Temperaturen überwachen. Sowohl der Prozessor als auch die programmierbare Logik werden über konfigurierbare Taktgenerationsblöcke, welche Phase Locked Loop (PLL)-Taktgeneratoren und Taktteiler enthalten, mit einem entsprechenden Taktsignal versorgt.[Xil16]
Für die Entwicklungen von Systemen die einen ZYNQ-SoC enthalten, werden von Xilinx abgestimmte Werkzeuge bereitgestellt, um die Entwicklung und Integration von der programmierbaren Logik, den Bootloadern, Betriebssystemen und Softwareapplikationen zu vereinfachen. Dabei wird eine Trennung zwischen der Entwicklungsumgebung für die programmierbare Logik und den Softwarewerkzeugen vorgenommen, da erstere von Xilinx für den Einsatz von reinen FPGAs vorgesehen ist.
Die Xilinx Vivado Design Suite ist eine Entwicklungsumgebung für Hardwaredesigns der SoC- und FPGA-Familien von Xilinx. Die Funktionen umfassen dabei insbesondere Simulation, Synthese und Analyse von entsprechenden Designs, die in Form einer Hardwarebeschreibungssprache vorliegen. Unterstützt werden dabei VHDL, Verilog und SystemVerilog, wobei der Umfang der Konstrukte für Synthese und Simulation gegenüber dem Gesamtumfang der Sprache deutlich beschnitten ist und je nach Sprache variieren kann.[Xil18a, S. 274ff.]
Weitere Funktionen ermöglichen Analyse und Optimierung von Timing und Ressourcennutzung, Clock- und Data-Management und den Export von Hardwaredefinition der Schnittstellen zwischen der programmierbaren Logik und der CPU bei aufwändigen SoC-Designs für die Softwareentwicklung. Die Entwicklung von Systemen kann durch die Verwendung des von Vivado mitgelieferten IP-Katalog beschleunigt werden. Während einige mitgelieferte IP-Blöcke generische Funktion enthalten, besteht der Großteil der IP-Blöcke aus speziell angepassten Funktionen, welche die Ausnutzung der SoC-Architekturen von Xilinx vereinfachen soll. Neben den Funktionen zum Erstellen und Implementieren von Hardwaredesigns enthält Vivado auch Funktionen zum Flashen und Debuggen der SoCs und FPGAs über unter anderem Joint Test Action Group (JTAG) oder TCP/IP. Für die Debug-Operationen werden gegebenenfalls zusätzliche IP-Blöcke, wie ein interner Logikanalysator oder die Ethernet Debug-Bridge benötigt, welche zunächst per Synthese in das Design der programmierbaren Logik integriert werden müssen.[Xil19b] Neuere Versionen von Vivado enthalten Werkzeuge um eine high-level Synthese von System-C Programmcode durchzuführen.
Für die Softwareentwicklung stellt Xilinx das Xilinx Software Development Kit (XSDK) bereit, dessen integrierte Entwicklungsumgebung (Integrated Development Environment, IDE) auf Eclipse und den Eclipse CDT Werkzeugen basiert. Neben der IDE bringt das SDK sämtliche notwendige Toolchains für das Cross-Kompilieren und Linken von Anwendungen und ganzen (einfachen) Betriebssystemen bereit. Die Toolchain greift dabei

auf generische ARM GCC Compiler zurück, die mitgelieferten Treiber und Bibliotheken sind jedoch stark an die SoC-Produkte von Xilinx angepasst. Das XSDK unterstützt die Entwicklung von Baremetal, FreeRTOS und Linux-Applikationen. Für die Entwicklung von FreeRTOS-Projekten wird eine vollständige angepasste FreeRTOS Version als Projekttemplate mitgeliefert. Etwaige Bibliotheken für Linux-Anwendungen müssen dagegen extern erzeugt und dann in die IDE eingebunden werden. Xilinx bietet dazu eine Option in seinen Petalinux-Tools an.
Neben dem Erstellen von Anwendungen bietet das XSDK umfangreiche Debug-Werkzeuge, welche über JTAG bei Baremetal- oder FreeRTOS-Anwendungen oder einen TCF-Server (TCP/IP) bei Linux-Projekten genutzt werden können. Ebenfalls vorhanden sind Werkzeuge zum Erstellen und Flashen von binären Bootdateien, welche in einem nicht-flüchtigen Speicher abgelegt werden, um den SoC beim Systemstart mit den notwendigen Einstellungen und dem Bitstream für die programmierbare Logik zu konfigurieren.
Die Komplexität eines vollständigen Embedded Linux inklusive Kernels ist durch die Vielzahl an Abhängigkeiten und Konfigurationsmöglichkeiten so hoch, dass das Kompilieren eines Kernels und z.B. etwaiger Module nicht in einer IDE durchgeführt wird. Typischerweise erfolgt diese Kompilierung durch einen rekursiven Aufruf verschiedenster Konfigurations- und Makefiles. Neben den Makefiles werden zusätzlich noch Modifikationen und Treiber benötigt, um den Kernel an die vorhandene Hardware anzupassen, insbesondere wenn diese, wie z.B. bei der ZYNQ-Architektur gegeben, deutlich von den typischen Hardwarekonfigurationen eines Mikrocontrollers mit ARM-CPU-Kern abweicht. Xilinx liefert mit den Petalinux Tools eine vom XSDK unabhängige Toolchain für das Kompilieren eines speziell angepassten Linux Kernels. Der Petalinux Kernel, dessen Quelldateien zusammen mit der Toolchain ausgeliefert werden, ist an die Hardware des ZYNQs angepasst und nutzt die vorhandene Floating-Point-Einheit für Fließkommaoperationen. [Xil19a] Durch die zahlreichen Anpassungen ist die Version des Petalinux Kernels immer an eine spezifische Version der Toolchain gebunden, mit welcher der Kernel zusammen ausgeliefert wird. Das Kompilieren von neueren Kernelquellen wird in der Regel von älteren Petalinux Toolchain Versionen nicht unterstützt. Die in dieser Arbeit genutzte Version des Petalinux Kernels basiert auf der Kernel Version 4.14 (Petalinux 2018.2).
Neben dem Kernel und der Toolchain enthält das Petalinux die weiteren benötigten Komponenten, um aus dem Petalinux Kernel ein funktionsfähiges Embedded Linux zu machen, mit einem Root-Dateisystem, Treibern und einem Bootloader. Der eingesetzte Bootloader ist U-Boot, welcher das Booten des Linux-Images nach Übergabe durch einen First-Stage-Bootloader übernimmt.[DEN19]
Der First-Stage-Bootloader sorgt für die essentielle Konfiguration der CPU, Taktgeneratoren, einfacher Kommunikationsschnittstellen und das Programmieren des FPGAs mit einem Bitstream. Ein Eingreifen in den Boot-Prozess oder eine Änderung der Konfiguration des First-Stage-Bootloaders ist nur durch das Neukompilieren und Aufspielen auf die

Hardware möglich, da zum Ausführungszeitpunkt des First-Stage-Bootloaders wichtige Speicher- und Kommunikationsschnittstellen noch nicht initialisiert sind.
Mit Abschluss der Ausführung des First-Stage-Bootloaders sind diese soweit initialisiert, dass U-Boot weitreichende Konfigurationsoptionen bieten kann, welche über eine serielle Schnittstelle modifiziert werden können. Diese Einstellungen können permanent in einem für den Bootloader dedizierten SPI-Flash gespeichert werden, welcher auch die auszuführenden Binärdaten von U-Boot enthält. Weiterhin ist es möglich, Kernel-Images über Trivial File Transfer Protocol (TFTP) zu Testzwecken in den RAM zu laden. Im normalen Betrieb werden die Kernel-Images aus dem primären, nichtflüchtigen Speicher in den RAM geladen und dann von dort ausgeführt, wobei über die Konfiguration von U-Boot der Speicherbereich und etwaige Kernel-Argumente zu spezifizieren sind.
Das von den Petalinux Tools erzeugte Root-Dateisystem ist in der Anzahl der vorhandenen Programme sehr begrenzt. Soll ein Linux-System genutzt werden, welches bereits viele Programme und Funktionen integriert und gegebenenfalls eine Paketverwaltung mitbringt, so empfiehlt sich der Einsatz des Root-Dateisystems aus einer entsprechenden Distribution. Geeignet hierfür sind die meisten Linux-Distributionen, welche für die ARM-Architektur mit einer Hardware-Fließkommaeinheit gebaut wurden und einen headless Modus besitzen, d.h. ohne Display und X-Server auskommen. Im Rahmen dieser Arbeit wurde ein abgespecktes Debian (headless) verwendet.

3.2.1 Trenz Electronic TE0720 & TE0703

Das Testsystem nutzt einen ZYNQ-SoC nicht als individuelle Komponente, sondern ein Trenz Electronic TE0720 Modul und eine TE0703 Trägerplatine. Diese Architektur ist den hohen Anforderungen an das PCB-Design, welche ein ZYNQ-SoC stellt, geschuldet. Allein um den ZYNQ-SoC betreiben zu können, werden viele externe Komponente benötigt, was noch einmal um weitere gesteigert werden muss, um das Potential der ZYNQ-Platform optimal auszunutzen.
Zu den notwendigen Komponenten zählen Spannungswandler, um die Spannungsebenen von $3,3\,V$, $1,8\,V$, $1,5\,V$ und $1,0\,V$ bereitzustellen. Des Weiteren benötigt der SoC, sowie eventuelle physical Layer für Ethernet und USB, einen Oszillator als Taktgenerator. Außerdem müssen verschiedene externe Speicher, wie der DDR-RAM und der Flash für den Bootloader und das Betriebssystem, an den SoC angebunden werden. Dies führt dazu, dass der Aufwand für die Entwicklung und Montage des Systems um ein vielfaches steigt, um den Anforderungen des SoCs zu entsprechen. Nicht zuletzt auch, weil gerade der SoC und die Speicherchips mit ihren vielen Pins nur in einem feinen Ball Grid Array (BGA)-Package erhältlich sind, welche eine Platine mit mehr als vier Layern und eine komplexe (maschinelle) Bestückung erfordern.

Der Einsatz des TE0720 Moduls von Trenz Electronic erleichtert die Entwicklung des Systems dahingehend, dass alle notwendigen externen Komponente für den Betrieb des SoCs bereits auf dem Modul enthalten sind. Die Hauptplatine des Testsystems muss sich daher nur um die für das Testsystem spezifischen Funktionen kümmern, sodass als Schnittstelle zwischen dem TE0720 Moduls nur noch eine Versorgungsspannung und diverse I/Os, welche mit der programmierbaren Logik des SoCs verbunden sind, verbleiben.
Das für diese Arbeit gewählte Modul der TE0720 Serie ist ein Trenz Electronic TE0720-03-1CFA, welches über einen ZYNQ XC7Z020-1 SoC verfügt. Dieser ZYNQ besteht aus einer Dual-Core ARM A9 CPU, welche mit einer maximalen Taktfrequenz von $667\,MHz$ betrieben werden kann. Die programmierbare Logik dieser Variante besteht aus $85\,k$ Zellen $53\,k$ Look-Up-Tabellen und $106\,k$ Flipflops, ist damit ausreichend groß für das Logikdesign dieses Systems und erhält die Möglichkeit zu einem späteren Zeitpunkt das Design um weitere Funktionalität zu erweitern, ohne den SoC bzw. das Modul tauschen zu müssen.[Tre15][Xil16, S. 2ff.] Das Modul verfügt über $1\,GB$ DDR3-RAM, welcher direkt an den Speichercontroller des ZYNQs angebunden ist und $8\,GB$ embedded Multimedia Card (eMMC) Flashspeicher, welcher im Rahmen dieser Arbeit als Boot- und Rootpartition für ein Linux Betriebssystem genutzt wird. Der Bootloader wird in den $32\,MB$ QSPI Flash des Modules abgelegt, alternativ ist auch ein Booten eines Images von dem SD-Kartenslot des Trägermoduls möglich. Der SD-Kartenslot findet für dieses Testsystem jedoch keine Verwendung, sodass dieser gegebenenfalls zum einfachen Übertragen von Dateien unter Linux genutzt werden kann. Selbiges gilt für den USB-Controller. Dieser wird aktuell nicht aktiv genutzt, jedoch vom Linux Kernel unterstützt, sodass dieser zukünftigen Erweiterungen z.B. durch einen USB-WIFI-Adapter dienen kann, um die Portabilität zu erhöhen. Eine zusätzliche Komponente des Moduls, die in dieser Arbeit aktiv genutzt wird, ist die I2C-Real-Time Clock (RTC), welche als Zeitgeber für das Betriebssystem dient. Die Zeit dieser Uhr wird unter anderem für Datumsstempel in Log- und Testdateien genutzt, um eine bessere Zuordnung und Übersicht der Darstellungen zu erreichen.
Das TE0720 Modul wird mit der TE0703 Trägerplatine über drei fine-pitch Steckverbinder verbunden, welche insgesamt 260 Pins aufweisen, von welchen 152 als I/Os eine direkte Verbindung zur programmierbaren Logik des SoCs aufweisen. Die TE0703 Trägerplatine wird als Adapterplatine zwischen dem SoC-Modul und der Hauptplatine des Testsystems eingesetzt. Sie bietet einige Komponente, welche vor allem für die Entwicklung vorteilhaft sind. So befinden sich auf der Trägerplatine ein USB-Client-Controller von FTDI Ltd., welcher eine UART-Schnittstelle, sowie eine JTAG-Debug-Schnittstelle über ein Complex Programmable Logic Device (CPLD) zum SoC bereitstellt, sodass auf externe Debughardware während der Entwicklung verzichtet werden kann.[Tre18] Neben der Debughardware und einem einzelnen Schaltwandler, der die $5\,V$-Eingangspannung in $3,3\,V$ wandelt, zeichnet sich die Trägerplatine vor allem durch Steckverbinder in Standardbauform aus. Der Ethernetanschluss ist als RJ45 ausgeführt, der USB-Client Steckverbinder ist eine Mini-B

Buchse, der USB-Host Steckverbinder ist eine gewöhnliche Typ-A Buchse und der SD-Kartenslot akzeptiert microSD-Karten. Die vielen I/Os werden von der Trägerplatine zur Hauptplatine zusammen mit der Spannungsversorgung über zwei VG96-Steckverbinder geführt, welche einen einfach zu verarbeitenden Pinabstand von $2,54\,mm$ besitzen.
Ist zu einem späteren Zeitpunkt gewünscht die Abmessungen der Platine weiter zu reduzieren, kann die Trägerplatine ersetzt werden, und das SoC-Modul direkt auf der Hauptplatine aufgebracht werden. Für den reinen Betrieb ohne Debugging werden von der Trägerplatine lediglich der Ethernetanschluss, dessen physical Layer sich bereits auf dem SoC-Modul befindet, und der $3,3\,V$ Spannungswandler benötigt. Diese beiden Komponenten lassen sich unproblematisch auf die Hauptplatine integrieren.

3.3 Sensorsimulator

Der Sensorsimulator des Testsystems, welcher das Verhalten eines PSI5-Sensors imitieren soll, muss nicht nur die Funktionen der PSI5-Schnittstelle eines realen Sensors nachbilden, sondern auch den Einfluss des eigentlichen Sensors und dessen Auswerteelektronik. Zu dem Einfluss auf den PSI5-Bus ist vorrangig der Strombedarf des Sensors zu nennen, welcher zu einem fließenden Ruhestrom auf dem Bus führt. Das typische ideale Modell des Aufbaus der physical Layer des PSI5-Busses ist in Abbildung 3.3 zu sehen. Der phy-

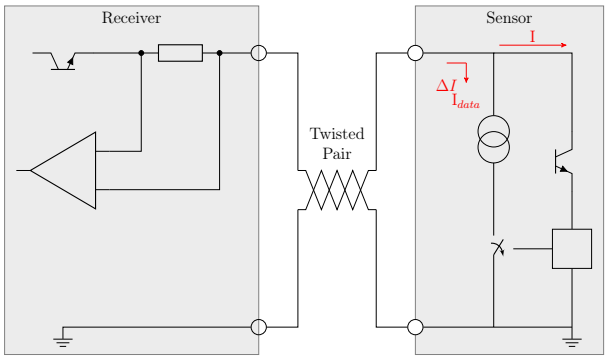


Abb. 3.3: Schematische Darstellung des physical Layers eines PSI5-Busses

sical Layer des Sensors besteht aus einer Detektorschaltung für die Spannungsimpulse des Steuergerätes und einer zuschaltbaren Stromsenke. Diese beiden Schaltungselemente bilden ebenfalls die Grundlage der Hardware des Sensorsimulators.
Der Funktionsblock zur Erkennung eines Synchronisationsimpulses kann in seinem Funktionsumfang unverändert in die Hardware des Testsystems übernommen werden. Seine einzige Aufgabe, die Auswertung der Spannungsimpulse auf dem PSI5-Bus und die Wandlung in ein digitales Steuersignal, bedarf keiner Modifikation. Die Stromsenke zur Erzeugung der Stromimpulse zur Datenübertragung vom Slave zum Master hingegen kann nicht übernommen werden, sondern muss in ihrem Funktionsumfang erweitert werden, um den Anforderungen des Testsystems zu entsprechen. Die Stromsenke eines echten PSI5 physcial Layers modelliert nur die Stromimpulse, welche zur Datenübertragung genutzt werden. Der Ruhestrom entsteht in einem tatsächlichen Sensor durch den Strombedarf der Sensorik und Auswerteelektronik und muss in dem Testsystem separat nachgebildet werden. Die Umsetzung kann daher entweder aus zwei separaten Stromsenken bestehen oder alternativ aus einer einzelnen Stromsenke, welche nie vollständig ausgeschaltet wird, sodass stets ein bestimmter Ruhestrom fließt.

Die Umsetzung der Hardware des Testsystems besteht aus einem Dual-Digital-To-Analog Converter (DAC) mit zwei Kanälen, mit nachgeschaltetem Spannungs-Strom-Wandler zur Implementierung der Sendeseite des Sensorsimulators. Die Topologie des Testsystems nutzt dabei eine Mischung der beiden möglichen Konzepte zur Simulation von Ruhe- und Impulsstrom. So existiert nur eine Stromsenke, welche jedoch durch das analog summierte Signal zweier separater DAC-Kanäle gespeist wird. Mittels der DACs wird die Wandlung der digitalen Vorgabewerte für den Ruhestrom und den Impulsstrom des PSI5-Protokolls durchgeführt, welche durch den Testablauf spezifiziert und gegebenenfalls während des Testes variiert werden.
Die beiden gewandelten Signale werden in einem Netzwerk summiert, gefiltert und in der Amplitude angepasst, sodass sie dem Spannungs-Strom-Wandler zugeführt werden können. Die Anpassung der Amplitude des Eingangssignals des Spannungs-Strom-Wandlers dient der Anpassung des Bereichs, in welchem die Stromsenke Ströme erzeugen kann. Die Hardware ist dabei so ausgelegt, dass sowohl der Ruhestrom als auch der Impulsstrom bei dem Maximalwert des DACs $50\,mA$ entspricht.
Um den Impulsstrom zur Datenübertragung aktivieren und deaktivieren zu können, befindet sich in dessen Signalpfad zwischen dem DAC und dem Eingangsnetzwerk des Operationsverstärkers der Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) T1. Dieser wird durch ein digitales Signal der im FPGA enthaltenen Logik des PSI5-Pattern-Generators angesteuert. Dazu wird, wenn nur der Ruhestrom von der Logik angefordert wird, die Ausgangsspannung des DACs für die Impulsstromvorgabe auf ein Massepotential gelegt, sodass das summierte Signal für die Wandlung in den eigentlichen Strom des Busses den Anteil des Impulsstromes nicht enthält.
Der Schaltplan in Abbildung 3.4 zeigt diese Umsetzung eines Sensorsimulators, mit dem geschalteten Netzwerk. Die Komponente U1 ist der oben erwähnte Dual-DAC, der auf der

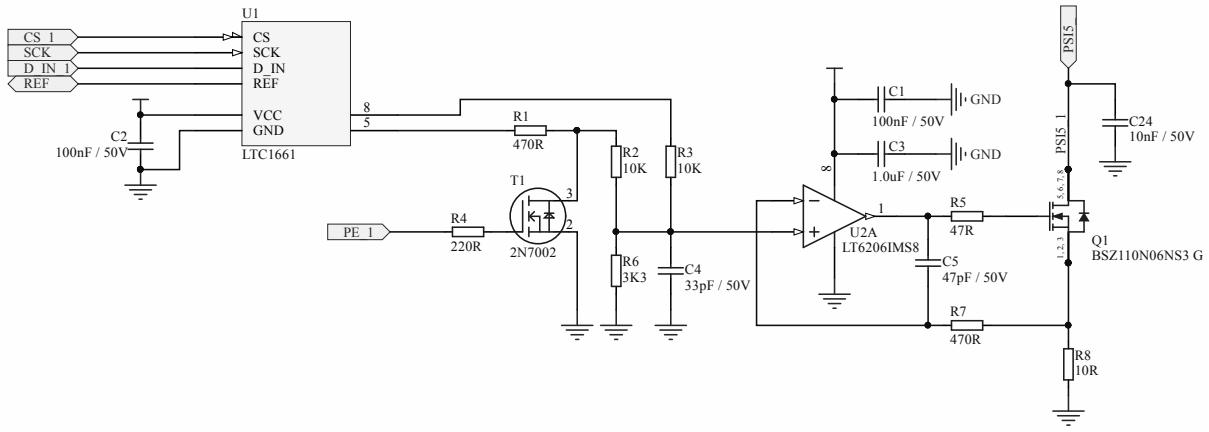


Abb. 3.4: Schaltplan der Stromsenke des Sensor Emulators

linken Seite eine SPI-Schnittstelle besitzt, über welche die neuen Werte zur Wandlung von dem FPGA übertragen werden. Die Auflösung der Digital-Analog-Wandler beträgt

$10\,Bit$, was bei der gegenwärtigen Auslegung des Testsystems mit einem maximalen möglichen Strom von $50\,mA$ einer Auflösung von ca. $0,5\,mA/LSB$ entspricht.
Die Summierung der beiden DAC-Spannungen als Sollwertvorgabe für eine einzelne Stromsenke hat neben der Einsparung einer unabhängigen Zweiten zusätzlich den Vorteil, dass der MOSFET der Stromsenke durch das Vorhandensein eines Ruhestroms kontinuierlich leitend ist. Dies begrenzt die Gate-Umladeströme und benötigte Slew-Rate am Ausgang des Operationsverstärkers, da für die Stromimpulse nur eine geringe Spannungsänderung am Gate des N-Kanal-MOSFETs Q1 benötigt wird.
Der Spannungs-Strom-Wandler besteht aus einem Operationsverstärker U2, einem N-Kanal-MOSFET Q1 und einem Messshunt R8, welcher für die Messung des Laststroms verwendet wird. Die Vorgabespannung ist am nicht-invertierenden Eingang des Operationsverstärkers angeschlossen und gibt dem Wandler den gewünschten Strom vor. Der mit dem Ausgang des Operationsverstärkers verbundene N-Kanal-MOSFET wird durch die entstehende Gatespannung eingeschaltet. Der durch die Stromsenke fließende Strom erzeugt einen Spannungsabfall über dem Messshunt R8, welcher zwecks Gegenkopplung mit dem invertierenden Eingang des Operationsverstärkers verbunden ist. Die Gegenkopplung sorgt dafür, dass sich die über den Messshunt abfallende Spannung auf denselben Wert wie die Spannung am nicht-invertierenden Eingang des Operationsverstärkers einstellt. Da die Spannung über dem Messshunt direkt proportional zu dem den Messshunt durchfließenden Strom abhängig ist, wird der Strom, welcher in der Stromsenke fließt, geregelt. Der Zusammenhang zwischen der summierten Vorgabespannung am Eingang des Operationsverstärkers und dem sich einstellenden Strom ist dabei durch den $10\,\Omega$ großen Messshunt definiert als:

$$I_{PSI5} = \frac{U_{OP_{in+}}}{10\,\Omega} \tag{3.1}$$

Gleichung 3.1: Berechnung des Laststroms in Abhängigkeit der Vorgabespannung

Die Größe des Messshunts ist mit $10\,\Omega$ so dimensioniert, dass der Spannungsabfall nicht zu groß wird. Andernfalls können große geforderte Ströme auch bei geringem Innenwiderstand des MOSFETs nicht erreicht werden. Zeitgleich ist der Wert von $10\,\Omega$ ausreichend groß, dass der Einfluss der Offsetspannung des Operationsverstärkers und weitere Störgrößen in ihrem Einfluss auf den letztendlich fließenden Strom des Sensorsimulators begrenzt wird.

3.3.1 Sync Pulse Erkennung

Die zweite Hardwarekomponente des Sensorsimulators ist eine Detektionsschaltung zur Erkennung der Synchronisationspulse auf dem PSI5-Bus. Diese wandelt den analogen Spannungsimpuls des PSI5-Busses in ein diskretes, digitales Signal, welches als Eingangsgröße für den Patterngenerator beziehungsweise die Nachbildung des PSI5-Protokolls genutzt wird.
Die Detektionsschaltung für die Spannungsimpulse des Steuergerätes zur Synchronisation besteht aus einem Komparator mit vorgeschaltetem Tiefpass (siehe Abbildung 3.5). Dieser

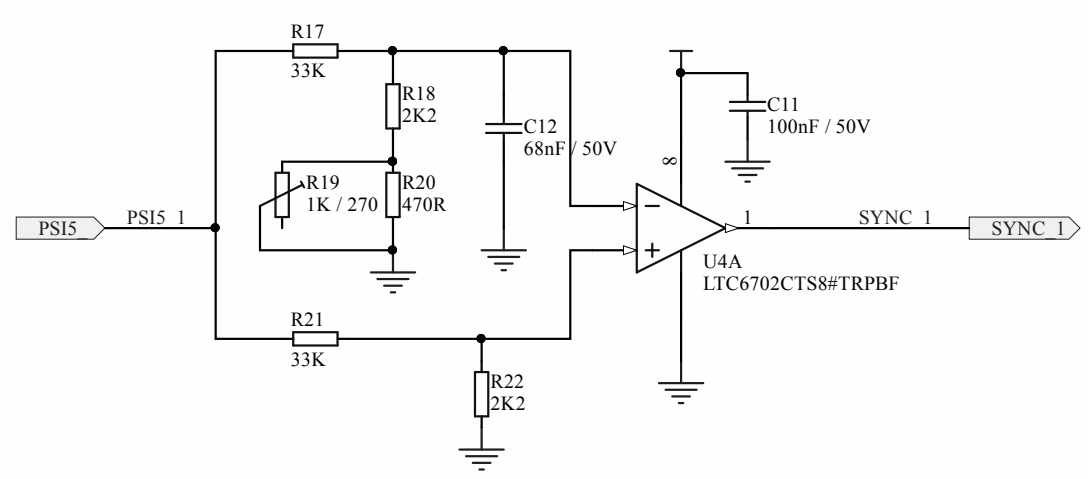


Abb. 3.5: Schaltplan der Sync-Impuls Detektion

filtert die Spannungsimpulse aus dem Signal des PSI5-Busses, sodass nur der Gleichanteil bzw. die Betriebsspannung übrig ist. Dieses Signal wird zum Vergleich mit dem tatsächlichen Signal des PSI5-Busses durch den Komparator herangezogen. Erzeugt das Steuergerät einen Spannungsimpuls, steigt die Spannung bei Einhaltung des PSI5-Standards durch den Controller mit einer Slew-Rate zwischen $0.43\,V/\mu s$ und $1,5\,V/\mu s$. Diese Spannungserhöhung wird von dem positiven Eingang des Komparators registriert, während das Signal am negativen Eingang durch die Tiefpassfilterung verzögert wird und auch in seiner Amplitude reduziert bleibt. Ist die Differenz zwischen dem tiefpassgefilterten und dem ungefilterten Signalen groß genug, gibt der Komparator dies als logische Eins am Ausgang weiter, sodass die nachfolgende digitale Logik dieses Signal zum Start einer Datenübertragung nutzen kann.
Die Grenzfrequenz des Tiefpasses liegt bei $935\,Hz$. Dieser Wert ist so gewählt, dass einerseits Schwankungen der Ruhestromaufnahme und die eigentliche Datenübertragung nicht zur fehlerhaften Detektion eines Sync-Impulses führen und andererseits eine ausreichend hohe Phasendifferenz zwischen den beiden Eingangssignale des Komparators erzeugt wird, um echte Sync-Impulse zuverlässig erkennen zu können.
Jeder der beiden Pfade zu den Eingängen des Komparators verfügt über einen Spannungs-

teiler, der verhindert, dass die Betriebsspannungen des PSI5-Busses den Eingangsspannungsbereich des Komparators verletzen und diesen beschädigen. Dieser liegt bei maximal $2,1\,V$, da sich die Eingangsspannung des Komparators immer mindestens $1,2\,V$ unter seiner Betriebsspannung von $3,3\,V$ befinden muss. Die Betriebsspannung des Komparators beträgt $3,3\,V$, da dieser einen Push-Pull-Ausgang besitzt, welcher direkt mit den I/O-Zellen des FPGAs verbunden werden soll, die eine maximale Eingangsspannung von $3,3\,V$ aufweisen.
Eine Anpassung der Schaltschwelle durch den FPGA bzw. die Steuersoftware ist nicht vorgesehen. Die Schaltschwelle ist statisch und ergibt sich aus dem Verhältnis der eingesetzten Widerstände. Dabei bilden der Widerstand R20 und das Potentiometer R19 eine alternative Bestückungsvariante, mit welcher eine manuelle Anpassung der Schaltschwelle des Komparators durchgeführt werden kann.

3.4 Fehlereinspeisung

Um die Robustheit und Unabhängigkeit mehrerer PSI5-Kanäle in einem ASIC zu testen, werden einzelne Kanäle gezielt gestört. Die Störung verschiedener PSI5-Kanäle kann dabei sequenziell oder gegebenenfalls auch simultan mit einer variablen Anzahl an betroffenen Kanälen entsprechend der Vorgaben des Testprogramms erfolgen. Im Extremfall werden sämtliche Kanäle mit einem externen Fehler beaufschlagt, und der zu testende ASIC muss dabei diesen Fehlerzustand zweifelsfrei identifizieren, wobei weitere Funktionsblöcke des ASICs nicht beeinträchtigt werden dürfen. Die Untersuchung von Fehlern funktioniert dabei nach dem grundsätzlichen Prinzip, welches auch der Rest des Testsystem nutzt. Das Testsystem stellt einen in diesem Fall fehlerbehafteten PSI5-Frame oder Zustand wie einen Kurzschluss bereit und liest über die SPI-Schnittstelle die Statusinformationen des DUTs aus.
Die Fehlerarten können in zwei Gruppen kategorisiert werden, Fehler in dem PSI5-Protokoll bzw. dem Frame und Fehler der eigentlichen Datenübertragung durch den physical Layer. Die Fehler im Protokoll können einfache Fehler, wie ein fehlerhaftes Bit in einem Frame, eine fehlerhafte Prüfsumme oder ungültige Manchesterkodierung sein oder aus komplexeren Verletzungen des Protokolltimings, z.B. der Zeitslots, sein. Sämtliche dieser Fehler im eigentlichen Protokoll werden anhand entsprechender Konfigurationen des PSI5-Patterngenerators in der digitalen Domäne erzeugt. Für diese Art von Fehlern wird, mit Ausnahme von zusätzlichen Logikblöcken in dem FPGA, keine zusätzliche Hardware benötigt. Die Störungen des eigentlichen physikalischen PSI5-Busses können dabei geringe Überströme in der Ruhestromaufnahme oder während der Datenimpulse, große Überströme in Folge eines Kurzschlusses der Leitungen des PSI5-Busses mit den Versorgungsleitungen, oder Kurzschlüsse zwischen den Leitungen des PSI5-Busses umfassen. Erstere können durch den einstellbaren Strombereich des Sensorsimulators abgedeckt werden. Letztere benötigen zusätzliche Hardware auf der Hauptplatine des Testsystems, siehe Abschnitt 3.4.1 und 3.4.2.
Weitere Störquellen können Fremdspannungen oder Impulse sein, welche in das Bussystem eingespeist werden. Das Testsystem bietet hierfür keine internen Komponente für deren Erzeugung, sodass es zur Simulation dieser Fehlerfälle weiterer externer Geräte oder Schaltungen bedarf. Die Tests solcher Fehlerfälle dienen vor allem der Untersuchung des Einflusses von Substratströmen auf die Funktionalität und möglichen Wechselwirkungen zwischen den einzelnen Kanälen eines ASICs oder ASSPs, die in einem sperrschichtisolierten Prozess gefertigt wurden.
Die Robustheitstests, welche Kurzschlüsse und Fehlspannungen simulieren, sind primär auf das Prüfen des Verhaltens des ASICs oder ASSPs gegenüber denen durch mechanische Beschädigungen oder durch eintretende Feuchtigkeit in den Kabelbaum, in die Steuergeräte, Sensorik oder den Steckverbindern entstehenden Fehlern ausgelegt. Ein vollständiger

Ausfall der Funktion des ASICs oder eine permanente Beschädigung ist dabei durch die Schutzbeschaltung des ASICs zu verhindern. Insbesondere für das System mit Sensorik für Insassenrückhaltesysteme sind diese Test von enormer Bedeutung, da im Falle eines Unfalles mit einem Kraftfahrzeug von einer Beschädigung des Kabelbaums und etwaiger Sensorik ausgegangen werden muss. Vor allem ein Kurzschluss gegen die Masseleitung ist wahrscheinlich, da diese, mit wenigen Ausnahmen, bei allen Kraftfahrzeugen als Fahrzeugmasse über die Karosserie ausgeführt wird.

3.4.1 Kurzschluss gegen Versorgungsspannungen oder Masse

Das Testsystem kann über MOSFETs einen Kurzschluss eines oder mehrerer PSI5-Kanäle gegen eine der beiden Versorgungsleitungen im Kraftfahrzeug (KFZ), die positive Versorgungsspannung bzw. Batteriespannung $VBAT$ oder die Masseleitung bzw. Karosserie, nachbilden. Dazu ist das Testsystem an jedem Kanal mit einer Halbbrücke aus jeweils einem N-Kanal und P-Kanal-MOSFET ausgestattet. Diese werden von der Ablaufsteuerung zu den gewünschten Zeitpunkten eines Tests eingeschaltet, um die Fehler zu simulieren. Für den Kurschluss gegen die Masseleitung wird ein N-Kanal-MOSFET verwendet, welcher den PSI5-Bus gegen Masse schaltet. Der MOSFET wird von den I/Os des FPGAs respektive der programmierbaren Logik gesteuert. Die in Abbildung 3.6 dargestellte Schaltung für den Kurzschluss gegen die positive Versorgungsspannung $VBAT$ nutzt einen P-Kanal-MOSFET und vorgeschaltete Levelshifter aus einem Widerstand und N-Kanal-MOSFET, um die benötigten Gate -Spannungen aus dem Logiksignal des FPGAs zu erzeugen.

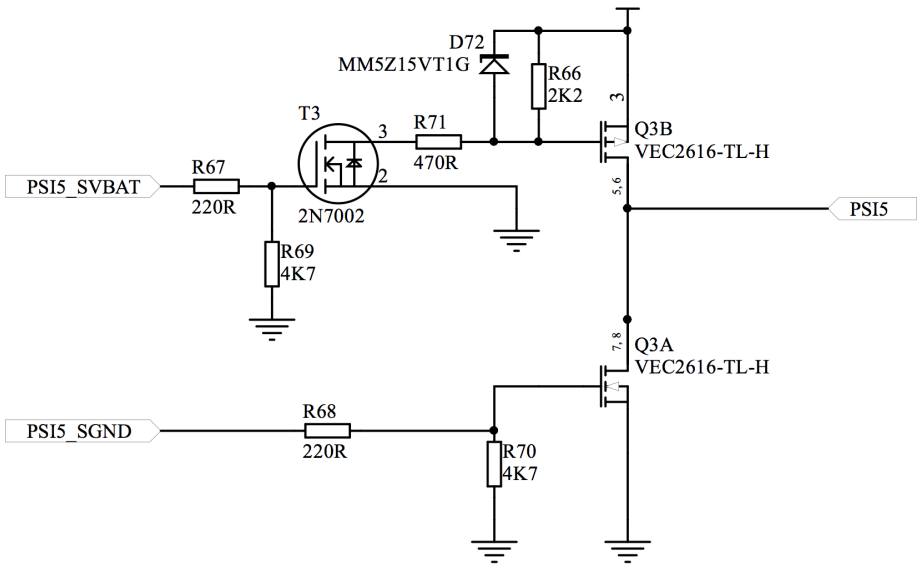


Abb. 3.6: Schaltplan der Kurzschluss Fehlereinspeisung

Da die Gates der beiden MOSFETs ohne weitere intelligente Treiber an die I/Os des FPGAs angeschlossen sind, ist die programmierbare Logik für die Kontrolle des Einschaltzeitpunkts der MOSFETs verantwortlich. Das Design der programmierbaren Logik muss daher sicherstellen, dass zu keinem Zeitpunkt beide MOSFETs eingeschaltet werden, da ansonsten die positive Versorgungsspannung des Testsystems direkt mit der Masse kurzgeschlossen würde.

3.4.2 Kurzschluss zwischen Leitungen

Neben den Kurzschlüssen der Leitungen des PSI5-Busses gegen die Versorgungsleitung sind Kurschlüsse zwischen den einzelnen PSI5-Kanälen eine Steuergerätes bei der Beschädigung des Kabelbaums im Kraftfahrzeug ein realistisches Szenario. Das Testsystem besitzt an jedem Kanal zwei Photo-Relais, um Kurzschlüsse zwischen zwei PSI5-Bussen simulieren zu können. Das Photo-Relais eines PSI5-Kanals ist durch eine Stromschiene bzw. Bus mit den Photo-Relais aller anderen Kanäle verbunden (vgl. Abbildung 3.7. Die zwei separaten Stromschienen bzw. Busse zur Erzeugung von Kurzschlüssen zwischen den PSI5-Kanälen ermöglichen die Abbildung zweier unabhängiger Kurzschlüsse. So können z.B. Kanal 1, 3, 4 und 5 über den einen Bus miteinander kurzgeschlossen werden, währen der zweite Bus genutzt wird, um Kanal 2 und 6 kurzzuschließen, ohne dass es eine leitende Verbindung zwischen den beiden Gruppen gibt.

Die eingesetzten Photo-Relais bestehen aus einer *GaAs* Infrarot LED auf der Eingangs-

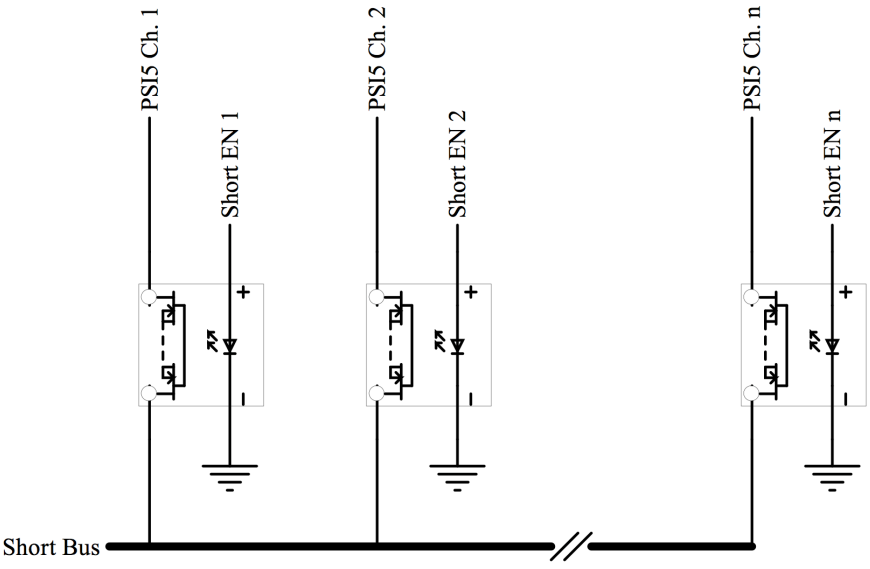


Abb. 3.7: Schaltplan eines Kurzschlussbusses (zwei Unabhängig im Testsystem vorhanden)

seite und einer Photozelle, sowie antiseriellen MOSFETs auf der Ausgangsseite. Durch den optische Signalweg zwischen der Infrarot LED und der Photozelle entsteht eine gal-

vanische Trennung von Ein- und Ausgang. Diese galvanische Trennung der Steuersignale von den PSI5-Kanälen sorgt dafür, dass, im Gegensatz zu einer Implementierung mit MOSFETs ohne optische Trennung, kein Bezug zwischen dem sich einstellenden Potential der beiden kurzgeschlossenen Kanäle und der Masse des Testsystems vorhanden sein muss.
Die antiserielle Verschaltung der MOSFETs am Ausgang des Photo-Relais ermöglicht einen bidirektionalen Stromfluss durch das Photo-Relais, welcher benötigt wird, da von dem Testsystem die Stromrichtung beim Kurzschließen zweier Kanäle nicht vorhergesehen werden kann. Die Stromrichtung in einem Kurzschluss ist abhängig von vielen, dem Testsystem unbekannten, Faktoren, wie dem Betriebszustand der PSI5-Schnittstelle, der Treiberstärke, Versorgungs- und Busspannung, sowie etwaigen Leitungs- und Übergangswiderständen.
Die beiden Kurzschlussbusse sind über Steckverbinder aus dem Testsystem herausgeführt. Über diese kann ein Kurzschlussbus zur Simulation von Fehlspannungen, die nicht Masse- oder Versorgungsspannungspotential haben, mit einer externen Spannungsquelle verbunden werden. Über die Photo-Relais kann dann ausgewählt werden, welche der PSI5-Kanäle mit der Fehlspannung beaufschlagt werden sollen.

4 FPGA Design

Das FPGA-Design ist nach Funktionszugehörigkeit in einzelne Blöcke unterteilt. Diese Blöcke werden in einem grafischen Top-Level instanziiert, welches schematisch in Abbildung 4.1 dargestellt ist. Diese Darstellung enthält zur Verbesserung der Übersichtlichkeit einige Auslassungen. Das vollständige grafische Top-Level ist in Anhang A.1 enthalten. Die Nutzung von einzelnen hierarchischen Blöcken im Gegensatz zu einem flachen Design bietet eine verbesserte Übersichtlichkeit und die Möglichkeit auf einfache Art mehrere parallele Instanzen eines Blocks zu erzeugen. In der gegebenen Struktur des Testsystems

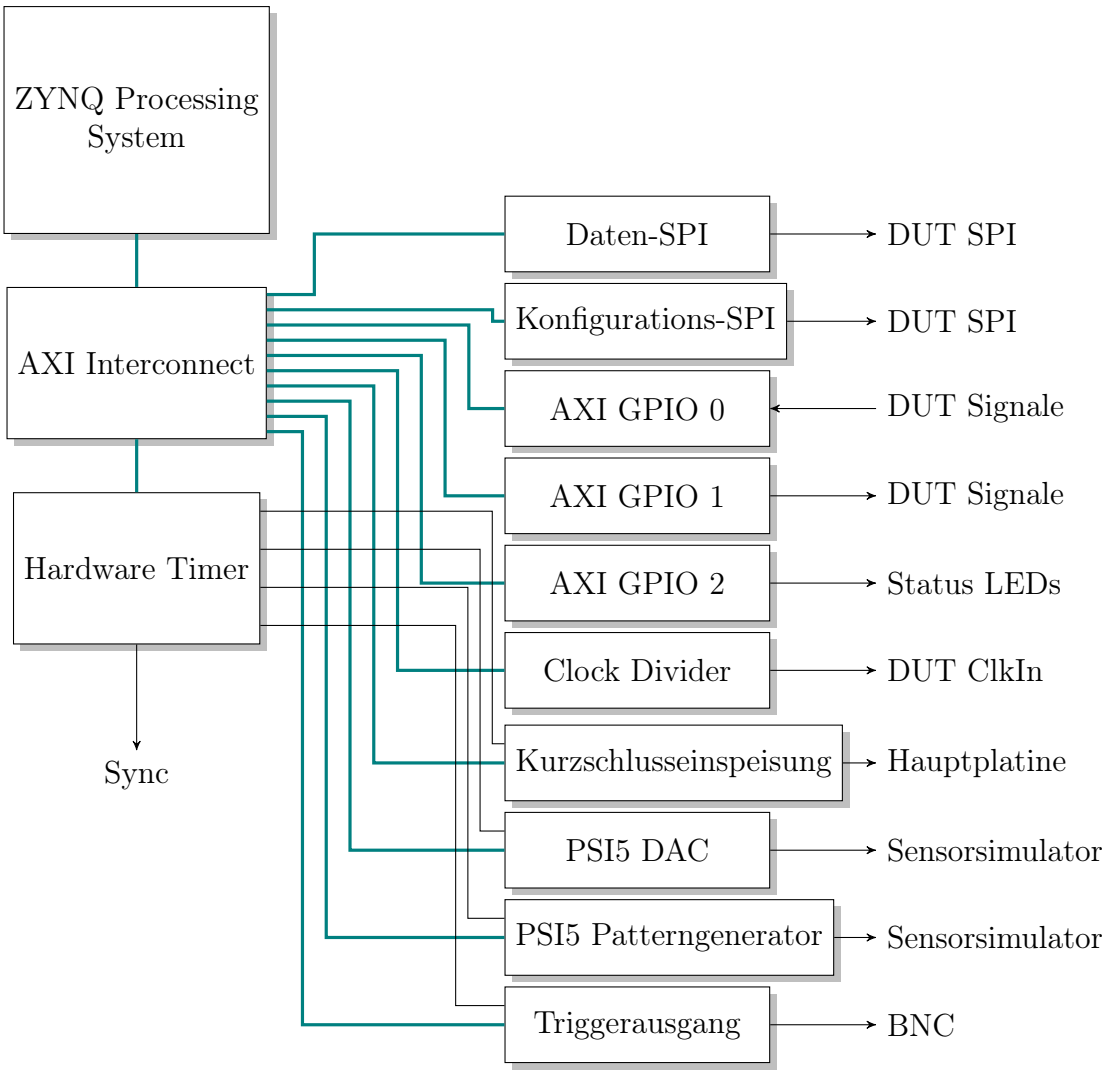


Abb. 4.1: Überblick über das Top-Level

sind sämtliche Blöcke über ein Interface mit der CPU des SoCs verbunden, da diese die Konfigurationen und Aktionen sämtlicher Blöcke organisiert. Eine Sonderstellung nimmt hierbei der Timer ein, da dieser nicht nur eine Verbindung zur CPU hat, sondern auch das Steuersignal für andere Blöcke bereitstellt.
Die Blöcke des Testsystems umfassen:

- **SPI-Interface:** Das Testsystem verfügt über zwei SPI-Master-Interfaces, welche genutzt werden, um das DUT zu konfigurieren und Nutzdaten zu empfangen. Das SPI-Interface ist dabei eine eigene Implementierung und kein fertiger IP-Block von Xilinx, da diese keine SPI-Blöcke kostenlos bereitstellen, die bis zu 32 *Bit* Nachrichtenlängen unterstützen. Die beiden SPI-Interfaces sind zwei unabhängige Instanzen des selben Blocks. Sie verfügen jeder über ein AXI, das über einen Interconnect mit der CPU verbunden ist. Die Software der beiden CPU-Kerne kann dabei jeweils auf beide Interfaces zugreifen. In der gegenwärtigen Auslegung des Testsystems ist jedoch vorgesehen, dass ein Interface nur von einer CPU bzw. einem Betriebssystem angesprochen wird. Das SPI-Interface übernimmt auch die Ansteuerung des Chip-Select (CS)-Signals, sodass kein separater Block zur Ansteuerung der General Purpose Outputs (GPOs) benötigt wird.

- **DAC-Ansteuerung:** Die Ansteuerung der DACs der Sensorsimulatoren des Testsystems erfolgt über einen SPI-Bus. Für sie wird jedoch nicht das oben genannte SPI-Master-Interface und Software genutzt, sondern es existiert ein extra Block. Dieser implementiert den notwendigen Data Link Layer und das Protokoll, welches die DACs benötigen. Der Block zur DAC-Ansteuerung nutzt das selbe SPI-Interface, wie der SPI-Interface Block, jedoch ohne das AXI-Interface. Die Statemachine des DAC-Blocks kommuniziert direkt über interne Signale mit dem zugehörigen SPI-Interface. Die Ansteuerung der DACs wurde in Hardware umgesetzt, um die CPU bzw. Software zu entlasten und die Latenz bis zur Aktualisierung der Ströme des Sensorsimulators zu verringern. Des weiteren kann die Aktualisierung so durch den Hardware Timer zu einem genau definierten Zeitpunkt ausgeführt werden, was bei einer Implementierung in Software nur begrenzt möglich wäre.

- **Xilinx GPIO-Block:** Besitzt das DUT weitere Steuereingänge, wie z.B. ein Enable-Signal, so ist das Testsystem durch diesen Block in der Lage die General Purpose Input/Outputs (GPIOs) des FPGAs anzusteuern, welche mit den Steuersignalen des DUTs verbunden sind. Die GPIO-IP-Blöcke des Testsystems stammen aus dem IP-Katalog von Xilinx, welcher mit Vivado ausgeliefert wird und besitzen ein AXI-Interface, über welches die Software die GPIOs setzten oder lesen kann. In dem Testsystem sind drei unabhängige GPIO-IP-Blöcke verbaut. Der erste wird für die Ansteuerung der Diagnose-LEDs des Testsystems genutzt. Die Software kann so bei Bedarf LEDs auf der Hauptplatine einschalten, um Zustände oder Fehler visuell darzustellen. Die beiden weiteren Blöcke dienen dem Setzen und Lesen von Steuersignalen des DUTs. Diese sind dabei nach Signalrichtung aufgeteilt, sodass der eine nur Ausgangssignale besitzt und der zweite GPIO-IP-Block nur Eingangssignale des Testsystem auswertet.

- **PSI5 Patterngenerator:** Der PSI5 Patterngenerator ist das Kernstück des Testsystems. Er ist für die Erzeugung der Signalformen der Sensorsimulatoren verantwortlich, sodass diese eine Nachricht entsprechend dem PSI5-Standard ergeben. Die Frames der Timeslots und deren Transfer sind in allen wichtigen Parametern variabel, sodass das Testsystem dem Anwender die größtmöglichen Freiheiten bietet verschiedenste Aspekte des DUTs zu untersuchen. Neben der Variation der Parameter einer Übertragung ist der Patterngenerator auch in der Lage ungültige Frames z.B. mit Prüfsummenfehler durch eine Fehlereinspeisung zu erzeugen. Das Testsystem verfügt gegenwärtig über einen Patterngenerator der bis zu zehn PSI-5 Kanäle je maximal vier Timeslots parallel bedienen kann.

- **Fehlereinspeisung:** Dieser Block erzeugt das Steuersignal für die Fehlereinspeisung von Kurzschlüssen gegen Versorgungspotentiale oder zwischen PSI-5 Kanälen entsprechend der Vorgaben der Software, welche diese über das AXI-Interface an den Block sendet. Diese Funktionalität ist nicht durch einen GPIO-Block umgesetzt, da der Block der Fehlereinspeisung ungültige Konfigurationen erkennen und verhindern muss. Diese könnten ansonsten zu ungewollten Kurzschlüssen im Testsystem und gegebenenfalls zu einer Beschädigung des Testsystems oder dem DUT führen.

- **Hardware Timer:** Das Testsystem enthält einen $32\,Bit$ Timer, um die primär in Software umgesetzte Ablaufsteuerung der Tests zu unterstützen. Der Timer ermöglicht die Untersuchung von timing-kritischen Aspekten des PSI5-Busses, indem er Steuersignale für Aktionen des Testsystems zu exakten Zeitpunkten ohne die Latenz der Software bereitstellt. Die Software ist abhängig von der Auslastung der CPU bei der Verwendung von nicht-echtzeitfähigen Betriebssystemen. Und auch echtzeitfähige Betriebssysteme haben Limitierungen bezüglich der Auflösung des Ausführungszeitpunkts und der Latenzen für Bus-Zugriffe innerhalb des SoCs. Der Timer stellt zur Unterstützung der Ablaufsteuerung nur Steuersignale bereit, welche Aktionen von anderen Blöcken des Testsystems zu exakten Zeitpunkten auslösen. Die Konfiguration der anderen Blöcke mit der auszuführenden Aktion erfolgt weiterhin durch die Software über das AXI-Interface des entsprechenden Blocks.

- **Taktteiler:** Dieser Taktteiler des Testsystems ist vorgesehen, um das DUT mit einem Takt zu versorgen, wenn dieses keinen internen Oszillator besitzt oder eine zusätzliche externe Taktquelle benötigt. Über das AXI-Interface dieses Blocks kann im Betrieb die Ausgangstaktfrequenz angepasst werden. Da es sich um einen reinen Taktteiler und keine PLL handelt, können keine höheren Frequenzen als der den Block speisende Haupttakt erreicht werden und das Verhältnis zwischen den Fre-

quenzen des Ein- und Ausgangstakts muss ein ganzzahliger Faktor sein. Abgesehen von der Einstellung der Ausgangstaktfrequenz sind keine weiteren Einstellungsmöglichkeiten vorgesehen.

- **Triggerausgang:** Der Triggerausgang des Testsystems wird durch einen extra Block angesteuert. Dieser beinhaltet einen Multiplexer, sodass verschiedene Aspekte des Testsystems als Quelle für das Triggerausgangssignal genutzt werden können. Dadurch lässt sich die Triggerquelle des Testsystems für andere externe Instrumente an die verschiedensten Testabläufe anpassen. Neben der Auswahl der Systemgröße für das Triggerausgangssignal kann der Trigger-Block auch Ausgangssignale mit einer einstellbaren Dauer erzeugen, sodass auch kurze interne Impulse an externe Instrumente sicher weitergeleitet werden können.

Zur Kommunikation der einzelnen Blöcke mit der ARM CPU wird ein AXI4-Bus verwendet. Dieser ist Teil der Arm Advanced Microcontroller Bus Architecture (AMBA) Spezifikation von ARM und wird ebenfalls von einem großen Teil der IP-Blöcke von Xilinx benutzt. Die in diesem System verwendete Variante des AXI4-Busses arbeitet mit 32 Bit breiten Registern und Signalen. Das AXI-Interface existiert in drei verschiedenen Versionen: Einem AXI4-Stream Interface, welches einen schnellen unidirektionalen Datentransfer ermöglicht, dem Memory Mapped AXI4-Interface und dem abgespeckten Memory Mapped AXI4-Lite Interface. Die Memory Mapped Varianten nutzen eine Abbildung in den Speicherraum der CPU zur Kommunikation zwischen dem IP-Block mit AXI-Interface und der CPU. Es werden keine speziellen Instruktionen benötigt, um mit den externen Blöcken in der programmierbaren Logik zu kommunizieren, sondern es können die üblichen Speicherzugriffsroutinen der CPU genutzt werden. Einziger Unterschied aus Softwareseite ist der Speicherbereich, welcher vom Adressraum des DDR-Hauptspeichers abweicht.
In Tabelle 4.1 sind die Basisadressen der AXI-Interface der eingesetzten Blöcke aufgelistet. Das nullte Register des Interfaces belegt dabei die angegebene Basisadresse, während für den Zugriff auf die weiteren Register eines Interfaces ein Offset zur jeweiligen Basisadresse benötigt wird. Jedes Register besitzt einen Offset von 0x04 zum vorherigen Register. Das dritte Register eines AXI-Interfaces kann daher z.B. durch die Basisadresse + 0x0C erreicht werden.
Die im Rahmen dieser Arbeit entwickelten IP-Blöcke verfügen über das abgespeckte AXI4-Lite Interface, da die Ansprüche der Blöcke an den Datentransfer zu der CPU vergleichsweise gering sind und die Taktfrequenz des Busses ausreichend hoch ist, sodass die von der normalen AXI4-Interface Variante unterstützten Burst-Zugriffe auf die Register bzw. den Speicherbereich des Interfaces nicht benötigt werden.
Das AXI-Lite Interface besteht aus fünf Kanälen: Read Address, Read Data, Write Address, Write Data und Write Response. Während eines lesenden Zugriffs wird zunächst

Tabelle 4.1: AXI Interface Basisadressen der IP-Blöcke

Name	**Funktion**	**Adresse**
gpio 0	DUT Input Signals	0x41200000
gpio 1	DUT Output Signals	0x41210000
clkdiv	Optionale DUT Taktquelle	0x41220000
spi 0	DUT Konfiguration SPI Interface	0x41230000
spi 1	DUT Nutzdatentransfer SPI Interface	0x41230000
gpio 2	Testsystem Diagnose Signale (LEDs)	0x41250000
ext trigger	Konfigurationsblock des Triggerausgangs	0x41260000
shortmap	Fehlereinspeisung / Kurzschlussmatrix	0x41270000
trigger timer	Hardware Timer der Ablaufsteuerung	0x41280000
psi5 dac	Ansteuerung der DACs der Sensorsimulatoren	0x41290000
psi5 pg	Konfiguration der PSI5 Patterngeneratoren	0x41290000

vom Master die gewünschte Adresse mit im Read Address Kanal abgelegt. Der Slave nutzt daraufhin den Read Data Kanal, um die gewünschten Daten an den Master zu senden. Bei einem schreibenden Zugriff stellt der Master über den Write Address und Write Data Kanal sowohl die Adresse, als auch die neuen Daten für das Register des Slaves bereit. Der Slave speichert die Daten des Write Data Kanals und kann über den Write Response Kanal dem Master bestätigen, dass der Schreibzugriff erfolgreich war. Die Übertragung von Daten und der Adresse eines Schreibzugriffes muss nicht gleichzeitig passieren. Jedoch müssen beide übertragen und vom Slave auswertetet worden sein, bevor dieser antworten kann.[Xil11, S. 6ff.]
Die eigentliche Datenübertragung sämtlicher Kanäle erfolgt über einen Handshake in zwei Schritten. Im ersten Schritt setzt der Sender ein Valid-Signal, welches die Gültigkeit der aktuell anliegenden Daten zeigt. In einem zweiten Schritt setzt der Empfänger ein Ready-Signal, welches seine Bereitschaft zum Empfang der Daten signalisiert. Sind beide Signale gesetzt, ist der Handshake abgeschlossen und mit der nächsten steigenden Taktflanke werden die Daten übernommen.
Der Takt der in diesem Design von den AXI-Interfaces genutzt wird, ist der selbe, welcher auch von den eigentlichen Blöcken als Haupttakt genutzt wird, sodass das gesamte System mit einem einzigen Haupttakt arbeitet. Der Vorteil dieses synchronen Designs aller Komponenten mit nur einer Taktquelle ist, dass innerhalb der programmierbaren Logik keine Taktdomänenwechsel stattfinden und daher keine Synchronisierer zwischen den einzelnen Komponenten benötigt werden. Lediglich externe Signal, welche in den FPGA über dessen GPIOs geführt werden, benötigen eine Synchronisiererstufe.
Der Haupttakt des Systems besitzt eine Frequenz von $100\,MHz$ und wird von einer PLL des SoCs erzeugt. Deren Konfiguration erfolgt durch den First-Stage-Bootloader, welcher auch die Logikblöcke des FPGAs mit einem Bitstream konfiguriert. Neben dem Haupttakt wird auch das Reset-Signal des AXI-Interfaces von den Interna der IP-Blöcke genutzt.

Führt der SoC einen Reset der Peripherie durch, werden die IP-Blöcke vollständig zurück gesetzt. Unabhängig davon ermöglichen einige Blöcke auch einen Reset ihrer selbst durch das Setzten eines Bits in ihrem Interface, ohne das andere Blöcke beeinflusst werden.
Die einzelnen Blöcke mit ihren AXI-Interfaces, welches allesamt Slave-Instanzen sind, sind durch einen Interconnect mit dem einzigen Master des Systems, dem SoC, verbunden. Der AXI-Interconnect ist vereinfacht dargestellt ein Multiplexer, welcher die Anfragen des Masters auswertet und entsprechend der verlangten Adresse die Verbindung einem Slave zuordnet.

4.1 Serial Peripheral Interface

Zwischen dem DUT und dem Testsystem, sowie innerhalb des Testsystems zur Kommunikation zwischen der programmierbaren Logik und den DACs der Hauptplatine, wird ein SPI-Bus benötigt. Die programmierbare Logik verfügt daher über mehrere Instanzen einer SPI-Master-Schnittstelle. Der SPI-Bus ist ein von Motorola entwickelter, synchroner, serieller Datenbus. Der SPI-Bus nutzt für die serielle Übertragung der Daten im Regelfall drei gemeinsame Leitungen, welche an jeden Teilnehmer des Busses angeschlossen sind.

- Serial Clock (SCK) - Takt zur Synchronisierung
- Master Input, Slave Output (MISO) - Datenleitung
- Master Output, Slave Input (MOSI) - Datenleitung

Dazu kommt jeweils eine CS-Leitung vom Master zu jedem der Slaves, mit welcher der Master vor dem Start der eigentlichen Datenübertragung auswählen kann, welcher der Slaves aktiv sein soll. Hat der Master durch die Aktivierung einer CS-Leitung den gewünschten Slave ausgewählt, kann er durch anlegen des SCK-Taktes die Datenübertragung starten. Mit jeder Taktperiode wird in beide Datenrichtungen simultan ein Bit übertragen. Ist keine bidirektionale Kommunikation erwünscht, so muss die andere Datenrichtung mit Dummy-Daten aufgefüllt werden. Eine unidirektionale Datenübertragung findet häufig zu Beginn einer Datenübertragung mit mehreren Nachrichten statt, da der Master dem Slave zunächst das Register der gewünschten Daten mitteilt. Slaves deren CS-Leitung nicht aktiviert wurden, dürfen ihre MOSI-Leitung nicht nutzen, da es ansonsten zu Kollisionen auf dem Bus kommen würde.
Den SPI-Bus gibt es in verschiedensten Auslegungen, was sowohl die Taktfrequenz und damit einhergehend die Übertragungsrate, als auch die Nachrichtenlänge betrifft. Typische Nachrichtenlängen umfassen ein Vielfaches von 8 Bit im Bereich zwischen 8 Bit und 32 Bit. Das Testsystem nutzt intern Nachrichtenlängen von 16 Bit zur Kommunikation mit den DACs, sowie 32 Bit mit dem DUT.Letztere wird durch den eingesetzten Safe-SPI-Standard festgelegt. Dieser besteht hauptsächlich aus einem logical Layer, sodass sich sein Einfluss in dem Aufbau und der Zugehörigkeit der zu übertragenden Nutzdaten widerspiegelt. Die Umsetzung des Safe-SPI-Standards erfolgt in diesem Testsystem durch Software, welche die Datenworte einer Nachricht bestimmt. Siehe dazu auch Abschnitt 5.4. Die hier eingesetzte SPI-Schnittstelle muss daher keine zusätzlichen Funktionen für den Safe-SPI-Standard implementieren.
Der Zusammenhang zwischen dem Takt und den Datenleitungen wird über zwei Eigenschaften beschrieben. CPOL definiert die Polarität des Taktes, während CPHA das Timing zwischen dem Takt und den Daten festlegt. Jeder dieser beiden Eigenschaften kann zwei

Werte annehmen, sodass der SPI-Bus, wie in Tabelle 4.2 dargestellt, insgesamt vier verschiedene mögliche Betriebsmodi besitzt. [HJH89]

Tabelle 4.2: SPI Betriebsmodi

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Hat die Polarität des Taktes CPOL den Wert null, so besitzt die SCK-Taktleitung einen Low-Pegel während der Bus im Ruhezustand ist. Die einzelnen Taktpulse bestehen aus jeweils einem High-Pegel. Daraus folgt, dass die erste Taktflanke zu Beginn einer Übertragung eine steigende Flanke ist. Für eine Konfiguration, bei welcher CPOL eins ist, gilt das Inverse. Ein High-Pegel ist während des Ruhezustandes des Busses vorhanden und die Taktimpulse bestehen aus Low-Pegeln. Die Übertragung beginnt daher, wie in Abbildung 4.2 zu sehen ist, mit einer fallenden Taktflanke und erst die zweite Flanke ist eine steigende Flanke. Der Wechsel zwischen CPOL=0 und CPOL=1 ist durch einen einfachen

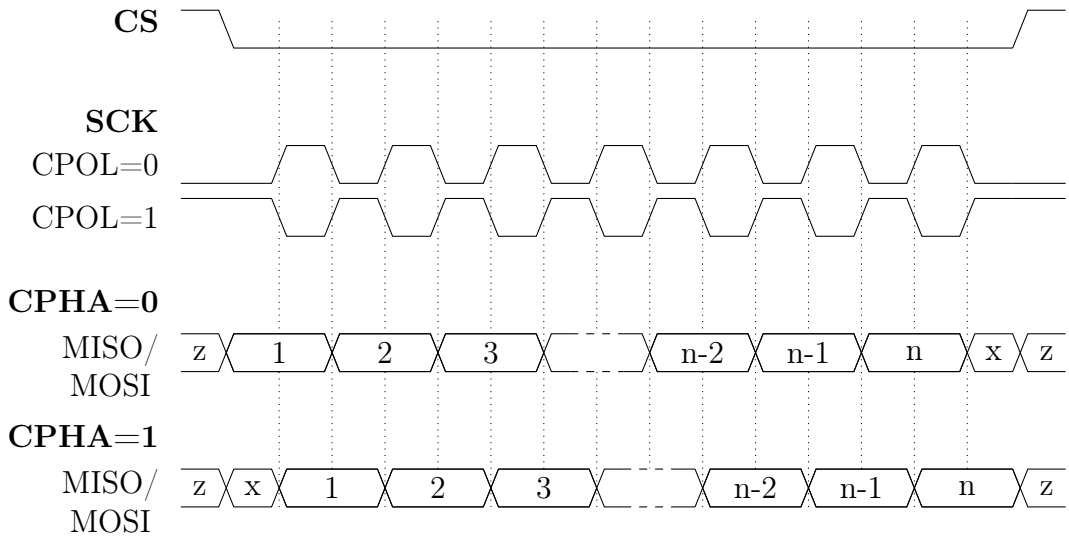


Abb. 4.2: Timingdiagramm der Betriebsmodi der SPI-Schnittstelle

Inverter zu realisieren.
Mittels CPHA wird die Phase zwischen den Taktflanken und dem Zeitpunkt, an welchem die Daten auf den Bus geschoben oder von diesem gelesen werden, bestimmt. Ist CPHA null, so werden zum Zeitpunkt der ersten Flanke die Daten von den beiden Datenleitungen gelesen und bis zur zweiten Taktflanke gehalten. Erst mit der zweiten Taktflanke des Taktimpulses werden die Daten auf dem Bus aktualisiert. Dies führt dazu, dass wenn CPHA

null ist, das erste Bit bereits ausreichend lange vor der ersten Taktflanke auf dem Bus anliegen muss, sodass es zum Zeitpunkt dieser gelesen werden kann. Typischerweise erfolgt das Setzen des ersten Bits auf den Bus zusammen mit dem Auswählen der CS-Leitung, sodass eine ausreichende Setup-Zeit zwischen dem Auswählen der CS-Leitung und der ersten Taktflanke eingehalten werden muss. Ist CPHA eins, werden die Daten zum Zeitpunkt der ersten Taktflanke auf dem Bus aktualisiert. Die Zustände der Datenleitungen werden damit erst gültig, wenn der Master die erste Taktflanke erzeugt hat. Vorher befinden sich die Datenleitungen in einem undefinierten Zustand (in der Abbildung 4.2 als x definiert) und sollten daher von Master und Slave ignoriert werden. Das Lesen der Daten vom Bus erfolgt mit der zweiten Taktflanke eines Taktimpulses. Dies führt dazu, dass der Zustand der Datenleitungen beim Übertragen des letzten Bits ausreichend lange gehalten werden muss, da er noch mit der allerletzten Taktflanke gelesen wird.
Die in diesem System eingesetzte Umsetzung einer SPI-Schnittstelle unterstützt alle vier Betriebsmodi zwischen denen mittels Konfigurationsbits im Betrieb gewechselt werden kann. Die Entwicklung einer eigenen SPI-Schnittstelle war notwendig, da die von Xilinx erhältlichen, kostenfreien IP-Blöcke keine 32 *Bit* Nachrichtenlängen unterstützen und für die DAC-Ansteuerung eine SPI-Schnittstelle benötigt wurde, die mehrere Kanäle simultan mit einer SCK-Leitung bedienen kann.
Die Hauptbestandteile, aus denen der SPI-Block besteht, sind zwei Schieberegister, welche die Daten der MISO- und MOSI-Datenrichtung beinhalten, einem Taktteiler für die Erzeugung des SCK-Taktes und eine Statemachine. Die Statemachine stellt als zentrales Element die Steuersignale für die restlichen Komponente bereit. Der SPI-Block ist dabei so entworfen, dass er eine beliebige Anzahl an SPI-Kanälen simultan bedienen kann. Die Kanäle teilen sich dabei die selbe Statemachine sowie den selben Taktgenerator und verfügen nur über eigene Schieberegister für die Daten. Dies schränkt die Kanäle dahingehend ein, dass alle Kanäle eines SPI-Blocks identische Einstellungen, wie z.B. Betriebsmodi, Datenlänge und Datenrate besitzen müssen.
Über verschiedene Parameter können die in Tabelle 4.3 dargestellten Aspekte der SPI-Schnittstelle zum Zeitpunkt der Synthese verändert werden. Der Parameter SCKDIV be-

Tabelle 4.3: Durch Synthese veränderbare SPI-Parameter

Parameter	**Funktion**
CHANNELS	Anzahl der SPI-Kanäle
LENGTH	Anzahl der Datenbits pro Nachricht
DELAY	Anzahl der Taktzyklen zw. CS-Auswahl und ersten SCK-Taktflanke
SCKDIV	/2 Teilfaktor zwischen Haupttakt und SCK-Takt
CS_NO	Anzahl von CS-Ausgängen

zeichnet den Teilfaktor zwischen dem Haupttakt, welcher den SPI-Block speist und dem von diesem erzeugten SCK-Takt. Neben diesem Teilfaktor wird der Haupttakt design-

bedingt zusätzlich innerhalb des SPI-Blockes halbiert, sodass z.B. bei einer Eingangsfrequenz von $10\,MHz$ und einem Teilfaktor von 5 die Ausgangsfrequenz $1\,MHz$ beträgt.
Ebenso ist zu beachten, dass die Anzahl der CS-Ausgänge unabhängig von der Anzahl der SPI-Kanäle ist. Es ist daher notwendig die Anzahl der CS-Ausgänge zu erhöhen, wenn mehrere SPI-Kanäle gewünscht sind und diese sich den CS-Ausgang nicht teilen sollen.
Die unter anderem in Tabelle 4.4 dargestellten Ein- und Ausgangssignale des eigentlichen SPI-Busses, die aus dem FPGA herausgeführt werden, sind Vektoren, welche ihre Größe entsprechend der Anzahl der Kanäle verändern. Ein SPI-Block mit fünf Kanälen hat

Tabelle 4.4: Ein- und Ausgangssignale des SPI-Blocks

Parameter	**Datenrichtung**	**Funktion**
clk	Input	Haupttakt des Blocks
reset	Input	Reseteingang des Blocks
enable	Input	Enable des Blockes
cpol	Input	Polarität des SCK-Signals
cpha	Input	Phase der Datenleitungen
data_in	Input	Zu übertragende Daten (Array)
data_out	Output	Empfangene Daten (Array)
cs_select	Input	Auswahl des CS-Signals (falls mehrere vorhanden)
done	Output	Statussignal
SCK	Output	SCK-Taktausgang
CS	Output	CS Ausgangssignal (Vektor)
MOSI	Ouput	MOSI Ausgangssignal (Vektor)
MISO	Input	MISO Eingangssignal, unsynchronisiert (Vektor)

dementsprechend auch fünf MISO- und MOSI-Datenleitungen. Die Taktleitung ist nur einfach ausgeführt, da diese, wie auch die internen Konfigurationssignale, für alle Kanäle identisch ist. Die internen Signale für die zu übertragenden und empfangenden Daten (data_in bzw. data_out) sind Arrays, die in der ersten Dimension von der Anzahl der Bits pro SPI-Nachricht abhängen und deren zweite Dimension durch die Anzahl der vorhanden Kanäle bestimmt wird.
Die Schieberegister eines SPI-Kanals haben, neben dem Reset- und Taktsignal, drei Steuereingangssignale. Das erste ist das preload-Signal, welches dem MOSI-Shiftregister signalisiert, dass es vorgeladen werden soll. Dabei werden die Daten, welche am data_in-Eingang anliegen vollständig und unverändert in das MOSI-Register übernommen. Wird das shift_en-Signal des MOSI-Shiftregisters aktiviert, so wird, so lange das Signal aktiv ist, mit jedem Takt ein Bit auf die MOSI-Leitung des SPI-Busses geschoben und die restlichen Bits rücken im Schieberegister entsprechend nach. Das letzte Steuersignal ist das sample_en-Signal, mittels dessen das MISO-Shiftregister den aktuellen Zustand der MISO-Leitung des SPI-Busses in das Register übernimmt.
Da die MISO-Leitung von außen in den FPGA geführt wird und deshalb nicht synchron zum Haupttakt des SPI-Blocks ist, wird dieses zunächst durch ein zusätzliches Flip-Flop

gepuffert. Zusammen mit dem Flip-Flop des Schieberegisters bildet das zusätzliche Flip-Flop einen Synchronisierer, der Glitches und metastabile Zustände verhindert.
Der Taktteiler für die Erzeugung des Taktes generiert das SCK-Ausgangssignal nicht direkt, sondern besteht nur aus einem Zähler. Die Auswertung des Zählerstandes und die Erzeugung des Taktsignals wird von der Statemachine übernommen. Der Zähler verfügt daher neben dem Reset- und Haupttakteingang nur über ein clk_enable-Signal. Ist dieses Signal gesetzt, zählt der Counter runter. Ist das clk_enable-Signal nicht gesetzt, der Reset aktiviert, oder erreicht der Counter null, wird der Zähler auf den vorbestimmten Wert zurückgesetzt. Der vorbestimmte Wert kann dabei über den Parameter SCKDIV angepasst werden. Der SPI-Block verfügt über einen identischen zweiten Zähler, welcher für die Verzögerungszeit zwischen dem CS-Signal und der ersten bzw. letzten Taktflanke genutzt wird. Dieser Zähler nutzt ein mit delay_enable bezeichnetes Signal als Steuersignal und wird über den Parameter DELAY angepasst.
Die Zustandsmaschine des SPI-Blocks besteht aus fünf Zuständen (siehe Abbildung 4.3), wobei der Transfer-Zustand auch als zwei separate Zustände betrachtet werden könnte. Im Ruhezustand bzw. Idle-Zustand, sind die Ausgänge der Statemachine und damit teil-

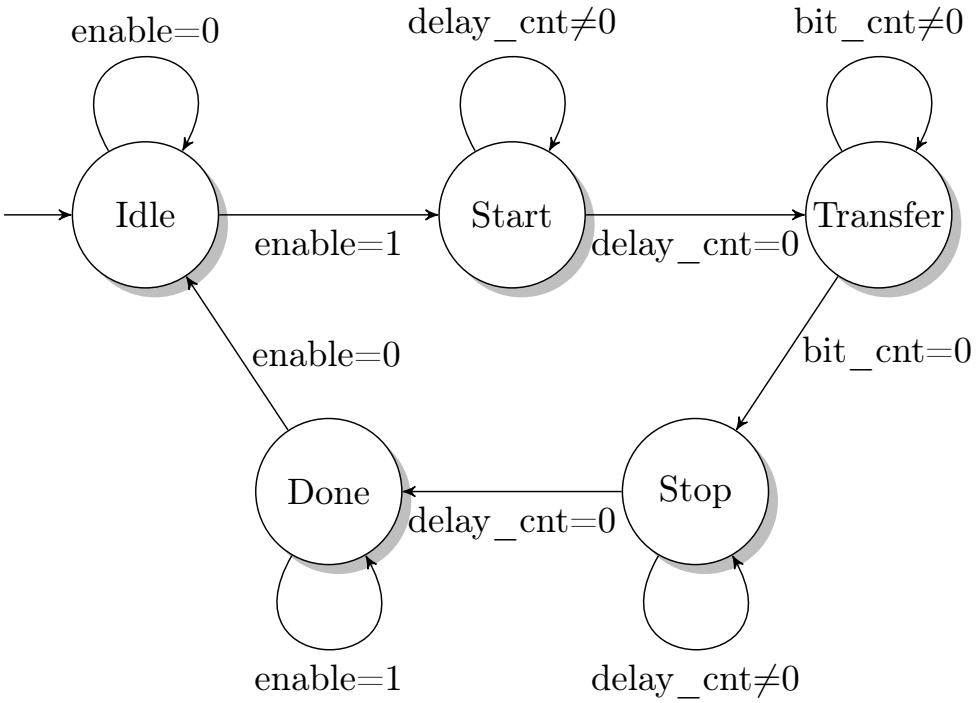


Abb. 4.3: Zustandsübergangsdiagramm der SPI Statemachine

weise auch die Ausgangssignale des Blockes auf definierte Werte zurückgesetzt. Es wird kein SCK-Takt erzeugt und auch keins der CS-Signale ist gesetzt. Dadurch dass die beiden Zähler des Blocks inaktiv sind, werden diese automatisch auf ihren jeweiligen vorbestimmten Wert zurückgesetzt. Die Konfigurationsbits wie cpha und cpol werden kontinuierlich in ein Register übernommen, so lange sich die Statemachine in dem Idle-Zustand befindet.

Selbiges gilt auch für die zu übertragenden Daten, denn da das preload-Signal im Idle-Zustand gesetzt ist, werden die Daten, die am Eingang des Blocks anliegen, permanent mit jedem Takt in das MOSI-Shiftregister übernommen.
Der Idle-Zustand kann nur verlassen werden, indem das enable-Signal gesetzt wird. In diesem Fall wechselt die Statemachine in den Start-Zustand. In diesem werden die Daten nicht mehr von den Eingängen übernommen und auch das MOSI-Shiftregister wird nicht mehr vorgeladen. Änderungen an den Eingangsignalen haben nun keinen Einfluss mehr auf die Übertragung. So wird verhindert, dass eine Änderung der Eingangssignale des SPI-Blocks während der Übertragung zu undefinierten Zuständen oder ungültigen Datenworten führt.
Im Start-Zustand werden die Werte von cpha und cpol ausgewertet und die Hilfsvariable clk_state auf die erste durchzuführende Aktion gesetzt. Ist cpha eins, ist die erste Aktion, die im folgenden Transfer-Zustand ausgeführt werden muss, eine Schiebeoperation der Daten auf den Bus. Ist cpha hingegen null, wird mit der ersten Taktflanke bzw. der ersten Aktion vom Bus gelesen. Daher muss bereits im Start-Zustand eine Schiebeoperation neben dem Aktivieren der CS-Leitung eingefügt werden. Die Schiebeoperation wird durch das Setzen des shift_en-Signals des MOSI-Shiftregisters für einen Takt erreicht. Neben diesen Aktionen wird im Start-Zustand der Zähler delay_cnt für die Verzögerung zwischen CS-Leitung und dem Start der Übertragung aktiviert. Der Start-Zustand wird automatisch verlassen und in den Zustand Transfer gewechselt, wenn der Zähler den Wert null erreicht hat. Dies stellt sicher, dass die definierten Setup-Zeiten des SPI-Busses eingehalten werden.
Der Transfer-Zustand kann als zweigliedrig betrachtet werden, da in diesem, abhängig von der Hilfsvariable clk_state, zwei unterschiedliche Abläufe stattfinden. Zeigt die Hilfsvariable clk_state an, dass als nächstes eine Schiebeoperation ausgeführt werden soll, so wird das shift_en-Signal des MOSI-Shiftregisters aktiviert und anschließend die Hilfsvariable auf eine Leseoperation gesetzt. Andernfalls wird das sample_en Signal des Shiftregisters gesetzt und der gegenwärtige Zustand der MISO-Leitung gelesen.
Beiden Abläufen ist gemein, dass sie in Abhängigkeit der Einstellungen von cpha und cpol das SCK-Ausgangssignal setzten. Damit zwischen den beiden abwechselnden Operationen eine Verzögerung entsteht und auch das SCK-Ausgangssignal die korrekte Periodendauer hat, wird zwischen dem Wechsel der beiden Operationen der Zähler des Taktteilers aktiviert und die nächste Aktion erst nach Ablauf dieses Zählers ausgeführt. Die Verzögerung zwischen den beiden Aktionen entspricht dabei genau der halben Periodendauer des erzeugten SCK-Taktsignals. Ist der Wert von cpha eins und die nächste auszuführende Aktion eine Schiebeoperation, wurde ein Bit korrekt übertragen. Selbiges gilt dementsprechend auch wenn cpha null und die nächste Aktion eine Leseoperation des Busses ist. Mit jedem übertragenen Bit wird ein Zähler, welcher die Anzahl der verbleibenden Bits zählt (bit_cnt), dekrementiert. Verbleiben keine zu übertragenden Bits mehr, wird das SCK-

Signal auf seinen Ruhewert entsprechend der Konfiguration durch cpol gesetzt. Ebenfalls wird der Zähler für die Verzögerung zwischen SCK und CS (delay_cnt), welcher bereits am Start der Übertragung genutzt wurde, erneut gestartet und es erfolgt der Wechsel in den Stop-Zustand.
Die Statemachine verbleibt so lange im Stop-Zustand, bis der Zähler delay_cnt null erreicht. Das CS-Signal wird erst mit dem Verlassen des Stop-Zustands zurückgesetzt. So wird auch, wenn cpha eins ist, sichergestellt, dass dem Slave genügend Zeit zum Samplen des letzten Bits bleibt.
Anschließend erfolgt der Wechsel in den Done-Zustand, in welchem der Block über das done-Signal anderen Komponenten den Abschluss einer Übertragung anzeigen kann. Ist in diesem Zustand das enable-Signal bereits zurückgesetzt, so wechselt die Statemachine mit dem nächsten Takt direkt wieder in den Ausgangszustand Idle. Ist das enable-Signal nicht zurückgesetzt, so verbleibt die Statemachine für unbegrenzte Zeit im gegenwärtigen Zustand. Dieses Verhalten stellt sicher, dass eine Übertragung nicht versehentlich wiederholt wird, sondern durch das Setzen und Zurücksetzen des enable-Signals explizit angefordert werden muss.

4.1.1 Serial Peripheral Interface AXI Wrapper

Der SPI-Block wird nicht nur als ein Bestandteil anderer Blöcke, wie z.B. der DAC-Ansteuerung genutzt, sondern auch als eine eigenständige Schnittstelle. In diesem Testsystem sind zwei unabhängige SPI-Schnittstellen eingebaut, die jeweils von einem CPU-Kern des SoCs angesprochen werden und mit dem DUT kommunizieren. Damit die CPU mit dem SPI-Block Daten austauschen kann, muss der SPI-Block mit einem zu der CPU kompatiblem AXI-Interface ausgestattet sein.
Der SPI-Block ist daher in einer Variante mit nur einem Kanal mit einem AXI4-Lite-Interface ausgestattet, welches über ein entsprechendes Interconnect mit dem AXI-Interface der CPU verbunden ist. Da der entwickelte SPI-Block unter anderem in der Anzahl der Kanäle und damit auch der Größe der Arrays und Vektoren durch Parameter zum Zeitpunkt der Synthese angepasst werden kann, befindet sich zwischen der AXI-Schnittstelle und dem SPI-Block ein Wrapper. Dieser ist notwendig, da die Werkzeuge von Xilinx zur Erstellung des Interfaces zum gegenwärtigen Zeitpunkt nur begrenzt mit dynamischen SystemVerilog Konstrukten umgehen können. Der SPI-Block ist in SystemVerilog geschrieben, der Wrapper hingegen, welcher die Anzahl der Kanäle festlegt und die Größen der Signalvektoren bzw. Arrays statisch macht, ist, ebenso wie das eigentliche AXI4-Lite-Interface, in Verilog umgesetzt. Andere Parameter wie z.B die Nachrichtenlänge in Bits oder der SCK-Taktteilungsfaktor, die nicht die Größe der Signale ändern, sind weiterhin über Parameter anpassbar und können daher auch beim Integrieren in das Top-Level

durch automatisch von Vivado generierte Einstellungsmöglichkeiten in den Dialogfenstern der Komponente angepasst werden.
Das AXI-Interface des SPI-Blocks hat vier 32 *Bit* Register. Wie Tabelle 4.5 zu entnehmen ist, sind von den ersten beiden Registern nur ein Bruchteil aller verfügbaren Bits genutzt. Es wäre daher theoretisch möglich die ersten beiden Register in einem Register zusam-

Tabelle 4.5: Register des AXI-Interfaces des SPI-Blocks

Register	**Bit(s)**	**Parameter**	**Funktion**
0	11-8	CS Select	Auswahl des genutzten CS-Ausgangs
0	2	Cpha	Phase der Datenleitungen
0	1	Cpol	Polarität der Taktleitung
0	0	Enable	Enable Signal des SPI-Blockes
1	0	Done	Übertragung abgeschlossen
2	31-0	Data In	Zu übertragenden Daten (MOSI-Daten)
3	31-0	Data Out	Empfangene Daten (MISO-Daten)

*Alle nicht explizit aufgeführten Bits der Register sind unbenutzt

menzuführen. In diesem Fall sind beide Register jedoch getrennt, sodass auf das nullte und zweite Register nur schreibend und auf die anderen beiden ausschließlich lesend zugegriffen werden muss.
Über die CS Select Bits kann die Software bestimmen, welcher der vorhanden CS-Ausgänge am Start einer Übertragung aktiviert werden soll. Die Kodierung erfolgt binär, sodass immer nur ein CS-Ausgang ausgewählt werden kann. Die vier Bit erlauben dabei bis zu 16 CS Ausgänge bzw. 16 verschiedene SPI-Slaves. Die tatsächliche Anzahl hängt dabei immer von der Anzahl der im Design angeschlossen CS-Leitungen ab, sodass bei einer geringeren Anzahl gegebenenfalls nur die unteren Bits Verwendung finden.
Über die Bits cpha und cpol kann durch die Software eingestellt werden, welcher SPI-Betriebsmodi genutzt werden soll. Dabei ist zu beachten, dass die Einstellungen erst mit dem Start der nächsten Übertragung verwendet werden. Findet eine Veränderung der Polarität der Taktleitung statt, ändert sich die Polarität im Ruhezustand des Busses nicht. Erst mit Beginn der nächsten Übertragung wird diese invertiert und bleibt dann auch während weiterer Businaktivitäten bestehen.
Durch das Setzten des Enable Bits des nullten Registers wird die Übertragung gestartet. Die Software muss dazu bereits in vorhergehenden Zugriffen auf das AXI-Interface des SPI-Blocks die Daten, welche über den SPI-Bus gesendet werden sollen, im Register zwei gesetzt haben. Der Abschluss einer Übertragung wird von dem Block durch das Setzen des niederwertigsten Bits im Register eins angezeigt und die zugehörigen empfangenen Daten im Register drei bereitgestellt.

4.2 DAC Ansteuerung

Der auf der Hauptplatine verbaute LTC1661 ist ein Digital-To-Analog Converter (DAC) mit zwei unabhängigen Kanälen. Für jeden der simulierten PSI5-Kanäle ist einer dieser DACs verbaut, sodass das System insgesamt über 20 unabhängige DAC-Kanäle verfügt, welche mit den entsprechenden Daten versorgt werden müssen. Jeder DAC besitzt dazu eine SPI-Schnittstelle, welche 16 Bit Nachrichten erwartet. Da der DAC keine Daten zurückliefert, sondern nur Konfigurationen empfängt, verfügt dieser über keine MISO-Datenleitung.[Lin16]
Die beiden Kanäle eines DACs haben eine identische Auflösung von jeweils 10 Bit. Daher ist es nicht möglich mit einer einzelnen SPI-Nachricht die Ausgangswerte der beiden Kanäle simultan zu ändern. Es werden mindestens zwei sequenzielle Nachrichten benötigt. Um dennoch die Möglichkeit zu bieten, die Ausgangsspannung der beiden Kanäle simultan aktualisieren zu können, besitzt der DAC zwei Register, von denen jeweils eins einem Kanal fest zugeordnet ist. In diese Register können die neuen gewünschten Werte der Ausgangsspannung über zwei SPI-Nachrichten geladen werden. Anschließend kann über einen separaten Befehl die simultane Wandlung der beiden Werte angestoßen werden. Alternativ ist es auch möglich das Laden des zweiten Registers direkt mit einem Wandlungsbefehl zu kombinieren. Der DAC lädt dann zunächst das zweite Register und beginnt automatisch nach Abschluss des Ladens die Wandlung. Die Ausgangsspannung der entsprechenden Ausgänge ist direkt proportional zu dem Wert, welcher zuvor in das Register geladen wurde. Dabei entspricht der Wert 0 einer Ausgangsspannung von 0 V und der Maximalwert 1023 einer Spannung von $2,5\,V$, da der Referenzeingang des DACs von einer $2,5\,V$ Spannungsreferenz gespeist wird.
Das gesamte entsprechende Verhalten des DACs wird über Befehle, die als SPI-Nachrichten nach Abbildung 4.4 an den DAC gesendet werden, konfiguriert. Die dargestellten Befehle bestehen aus einem 4 Bit Befehlscode, den 10 Bit Registerdaten und 2 Dummybits, um eine 16 Bit SPI-Nachricht aufzufüllen. In Tabelle 4.6 sind auszugsweise Befehlscodes und

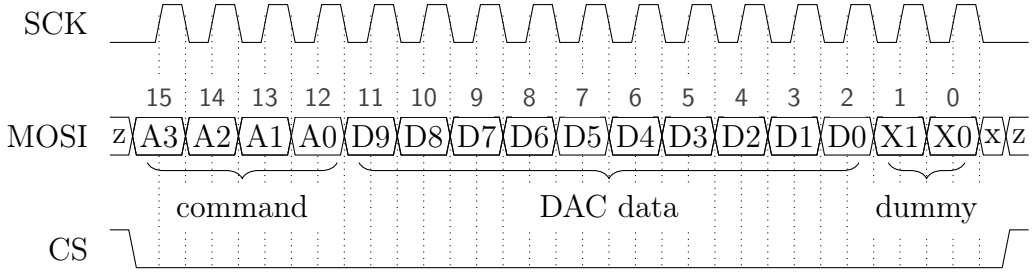


Abb. 4.4: Transferdiagramm der DAC SPI-Nachrichten, nach [Lin16, S. 8]

ihre Funktion aufgelistet. Dabei werden von dem DAC-Block des Testsystems der erste und dritte dargestellte Befehl genutzt.

Tabelle 4.6: Ausgewählte DAC SPI-Befehle, [Lin16, S. 9]

Befehl (Bit 15-12)	**Daten** (Bit 11-2)	**Dummy** (Bit 1-0)	**Funktion**
0b0001	Wert Kanal A	0b00	Lade Register A mit Wert A (kein Update der Ausgänge)
0b0010	Wert Kanal B	0b00	Lade Register B mit Wert B (kein Update der Ausgänge)
0b1010	Wert Kanal B	0b00	Lade Register B mit Wert B u. aktualisiere Ausgang A und B
0b1000	–	0b00	Aktualisiere Ausgang A und B

Der DAC-Block besteht aus einer Statemachine, welche die Erzeugung der Befehle koordiniert und eine Instanz der in Kapitel 4.1 beschriebenen SPI-Schnittstelle zum Transfer der Befehle an die DACs instantiiert. Die SPI-Schnittstelle verfügt über zehn Kanäle, sodass alle zehn DACs simultan mit neuen Werten versorgt werden. Das Aktualisieren eines einzelnen DACs ist nicht vorgesehen. Da sich jedoch die gewandelte Ausgangsspannung eines DACs nicht verändert, wenn er den selben Wert erneut erhält, kann ein einzelner Kanal verändert werden, indem alle DACs neue Daten erhalten, wobei sich nur die Daten des betroffenen Kanals unterscheiden. Durch diese Designentscheidung wird für alle DACs nur eine DAC-Statemachine und ebenfalls im SPI-Block nur eine SPI-Statemachine benötigt. Lediglich die Anzahl der Register im DAC-Block für die Zusammenstellung der Befehle und die Anzahl der Schieberegister im SPI-Block verändert sich mit der Anzahl der eingesetzten DACs.
Die MISO-Leitung des SPI-Blocks ist fest mit einem Low-Pegel verdrahtet, da die DACs nicht über eine solche Leitung verfügen. Dies führt dazu, dass während der Synthese durch Optimierungsverfahren die zugehörige Logik und Schieberegister aus dem Design entfernt werden.
Die Statemachine besteht aus acht Zuständen von denen zwei sich zur Implementierung einer Verzögerung einen zusätzlichen externen Zähler teilen (siehe Abbildung 4.5). Im Idle-Zustand der Statemachine werden für jeden DAC die Werte für seine beiden Kanäle in die Register des DAC-Blocks der programmierbaren Logik übernommen. Gleichzeitig wird das enable-Signal des zugehörigen SPI-Blocks und alle anderen Steuersignale des DAC-Blocks zurückgesetzt. Die Statemachine befindet sich damit in ihrem Ausgangszustand und ist bereit für eingehende Befehle. In diesem Zustand verbleibt die Statemachine so lange, bis das enable-Signal des DAC-Blocks gesetzt wird. Ist dies gesetzt, werden die am Eingang des DAC-Blocks anliegenden Daten nicht mehr in die internen Register übernommen, sodass die Daten, welche übertragen werden, nicht mehr von Änderungen am Eingang beeinflusst werden. Diese Speicherung der Daten verhindert, dass durch nachträgliche Änderungen der Daten am Eingang des Blocks ungültige Nachrichten versendet werden.

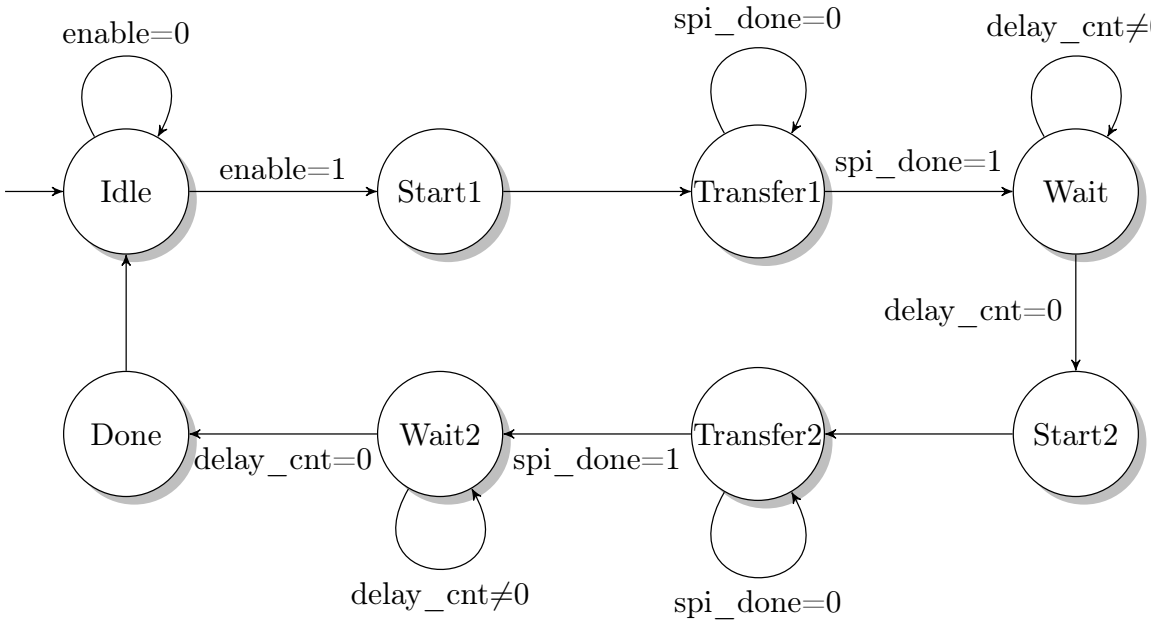


Abb. 4.5: Zustandsübergangsdiagramm der DAC Statemachine

Nach der Aktivierung des enable-Signals wechselt die Statemachine in den Start1-Zustand. In diesem wird die Erzeugung des ersten DAC-Befehls durchgeführt. Dazu wird in einem Vektor die 16 *Bit* Nachricht zusammengesetzt, welche zum Versand über die SPI-Schnittstelle an den Eingang des SPI-Blocks angelegt wird. In der ersten Übertragung soll der Befehl zum Laden des Registers A des DACs gesendet werden. Der Befehlscode besteht daher entsprechend der Tabelle 4.6 aus dem Wert 0x1. Anschließend werden in dem 16 *Bit* Vektor die 10 *Bit* des Ausgangswerts und zwei Dummybits angehängt. Das Zusammensetzen der Nachricht erfolgt dabei für jeden der vorhandenen DACs simultan, sodass bei diesem Testsystem zehn Nachrichten erzeugt werden. Zusätzlich wird in dem Start1-Zustand das enable-Signal des SPI-Blocks gesetzt, damit diesem mitgeteilt wird, dass die Nachricht zusammengesetzt ist und den Transfer selbiger auf den Bus starten kann.

Anschließend wechselt die Statemachine mit dem nächsten Takt in den Transfer1-Zustand. In diesem Zustand passiert außer der Rücknahme des enable-Signals des SPI-Blocks keine weitere Aktion. Die Statemachine verbleibt jedoch in diesem Zustand, bis ihr durch das spi_done-Signal vom SPI-Block mitgeteilt wird, dass der Transfer der Nachricht über die SPI-Schnittstelle abgeschlossen ist. Daraufhin geht die Statemachine in den Wait-Zustand über.

Im Wait-Zustand wird der Zähler zur Erzeugung einer Verzögerung gestartet. Ist er nicht aktiviert, so wird er automatisch mit einem Wert vorgeladen, welcher einer SCK-Taktperiode entspricht. Diese Verzögerung ist notwendig, da der SPI-Block zwar das Timing zwischen dem CS-Signal und den Taktimpulsen des SCK-Signals sicherstellt, nicht jedoch die Einhaltung der minimalen Ruhezeiten des Busses beachtet. Diese müssen von

dem darüberliegenden DAC-Block eingehalten werden, da ansonsten das Timing des DACs verletzt wird, welches eine minimale Impulslänge auf der CS-Leitung von $100\,ns$ vorschreibt (vgl. [Lin16, S. 3]).
Erreicht der Zähler delay_cnt für die Verzögerung den Wert null, so geht die Statemachine in den Zustand Start2 über. Die nun folgenden Zustände entsprechen von der Struktur den vorherigen Start1, Transfer1 und Wait Zuständen, da auch im zweiten Teil der DAC-Statemachine ein Befehl an den DAC versendet werden soll. Einzig der Aufbau des Befehls, der in dem Zustand Start2 zusammengesetzt wird, unterscheidet sich von den vorherigen Schritten. In diesem Fall soll dem DAC mitgeteilt werden, dass er sowohl das Register B laden soll, als auch die Wandlung für die beiden Ausgänge starten soll. Der Anfang der Nachricht besteht daher aus dem Befehlscode 0xA (vgl. Tabelle 4.6), an welchen noch die $10\,Bit$ des Wertes, welcher die Spannung des Ausgangs B bestimmt, und zwei Dummybits angehängt werden.
Ist auch der zweite Befehl versendet worden und ist die notwendige Wartezeit abgelaufen, geht die Statemachine in den Done-Zustand über. Dieser existiert, um das gleichnamige Signal zu setzen, welches anderen Blöcken oder auch der Software signalisiert, dass die Aktualisierung der DACs bzw. eine Änderung der Ausgangsspannung abgeschlossen ist. Aus dem Done-Zustand wechselt die Statemachine direkt mit dem nächsten Takt in den Idle-Zustand. Ein Prüfen, ob das enable-Signal bereits zurückgenommen wurde, wie es bei dem SPI-Block erfolgt, findet nicht statt, da eine ungewollte erneute Übertragung der selben Werte an die DACs keinen Einfluss auf deren Ausgangsspannung hat.

4.2.1 DAC AXI Interface

Der DAC-Block wird nicht als eine Komponente anderer Blöcke, sondern nur als eigenständiger Block genutzt. Dazu ist der DAC-Block mit einem AXI-Interface ausgestattet, welches durch die Software der CPU angesprochen wird. Die DACs werden wie in Abschnitt 3.3 beschrieben genutzt, um den Strom der PSI5-Kanäle zu definieren. Eine andere Verwendung fällt den DACs und damit auch dem Block, der diese ansteuert, nicht zu.
Der DAC-Block kann ähnlich wie der SPI-Block über einen Parameter zum Synthesezeitpunkt in der Anzahl der anzusprechenden DACs variiert werden. Da jedoch das AXI-Interface statisch ausfallen muss und das Testsystem in seiner gegenwärtigen Auslegung über zehn PSI5-Kanäle und damit zehn DACs verfügt, beträgt die Anzahl der unterstützten Kanäle ebenfalls zehn.
Das zugehörige AXI-Interface besteht aus elf Registern, wobei die letzten zehn Register identisch und jeweils einem DAC bzw. PSI5-Kanal zugeordnet sind (siehe Tabelle 4.7). Das Done-Bit signalisiert, dass die Werte, die in den Registern des AXI-Interfaces von der Software abgelegt wurden, an die DACs übertragen worden sind und diese jetzt die neue

Tabelle 4.7: Register des AXI-Interfaces des DAC-Blocks

Register	**Bit(s)**	**R/W**	**Parameter**	**Funktion**
0	3	R	Done	DAC Aktualisierung abgeschlossen
0	2	R/W	Update	Starte Aktualisierung
0	1	R/W	Reset	Reset DAC IP-Block
1	9-0	R/W	DAC 1 Wert A	Ruhestrom PSI5 Kanal 1
⋮	⋮	⋮	⋮	⋮
10	9-0	R/W	DAC 10 Wert A	Ruhestrom PSI5 Kanal 10
1	25-16	R/W	DAC 1 Wert B	Impulsstrom PSI5 Kanal 1
⋮	⋮	⋮	⋮	⋮
10	25-16	R/W	DAC 10 Wert B	Impulsstrom PSI5 Kanal 10

*Alle nicht explizit aufgeführten Bits der Register sind unbenutzt

Ausgangsspannung bereitstellen bzw. der Strom der PSI5-Kanäle angepasst wurde.
Mittels des Update-Bits teilt die Software dem DAC-Block mit, dass dieser die gegenwärtig in den Registern 1-10 vorhandenen Werte an die DAC weiterleiten soll. Dazu sind die jeweils gültigen untersten zehn Bits, sowie die Bits 25-16 der Register des AXI-Interfaces, mit den entsprechenden Werteeingängen des DAC-Blocks verbunden. Der zweite Wert beginnt mit dem Bit 16, sodass der 32 Bit Registerinhalt durch die Software durch das Aneinanderreihen von zwei 16 Bit Variablen, welche den Ruhe- und Impulsstrom beinhalten, bestimmt werden kann. Dabei ist zu beachten, dass nur jeweils 10 Bit gültig sind und die höherwertigen Bits ignoriert werden.
Neben dem Update-Bit kann auch der separate Update-Eingang des Blocks genutzt werden, um eine Aktualisierung der Ströme des PSI5-Busses anzustoßen. Dieser wird in dem vorliegenden Testsystem mit dem Timer-Block verbunden und genutzt, um eine zeitgenauere Aktualisierung der Ströme vorzunehmen. Dazu werden die neuen Werte zunächst durch die Software über das AXI-Interface an den DAC-Block gesendet, der zugehörige Aktualisierungsbefehl wird jedoch durch den Timer ausgelöst. Dabei ist zu beachten, dass zwar die Latenz der Software und des AXI-Busses umgangen wird, die Laufzeiten des SPI-Busses aber weiterhin vorhanden sind.
Wird das Reset-Bit gesetzt, wird der DAC-Block zurückgesetzt. Die Statemachine bricht die aktuelle Übertragung ab und die internen Register werden zurückgesetzt. Das Reset-Bit hat die höchste Priorität. Solange dieses Bit gesetzt ist, ignoriert der DAC-Block das Update-Bit. Der Inhalt der Register des AXI-Interfaces wird dabei nicht gelöscht. Sollen die Register im AXI-Interfaces, welche Ruhe- und Impulsstrom aller Kanäle speichern, ebenfalls zurückgesetzt werden, müssen diese manuell überschrieben oder das AXI-Interface über seinen separaten Reseteingang zurückgesetzt werden. Dies erfolgt z.B. beim Systemstart oder einem Reset des AXI-Busses durch den SoC.

4.3 PSI5 Patterngenerator

Der PSI5 Patterngenerator besteht aus drei Gruppen an Funktionsblöcken, welche unterschiedliche Aufgaben übernehmen: den Framegeneratoren, den Schieberegistern zusammen mit dem Taktteiler, sowie einer Statemachine (siehe Abbildung 4.6). Analog dazu

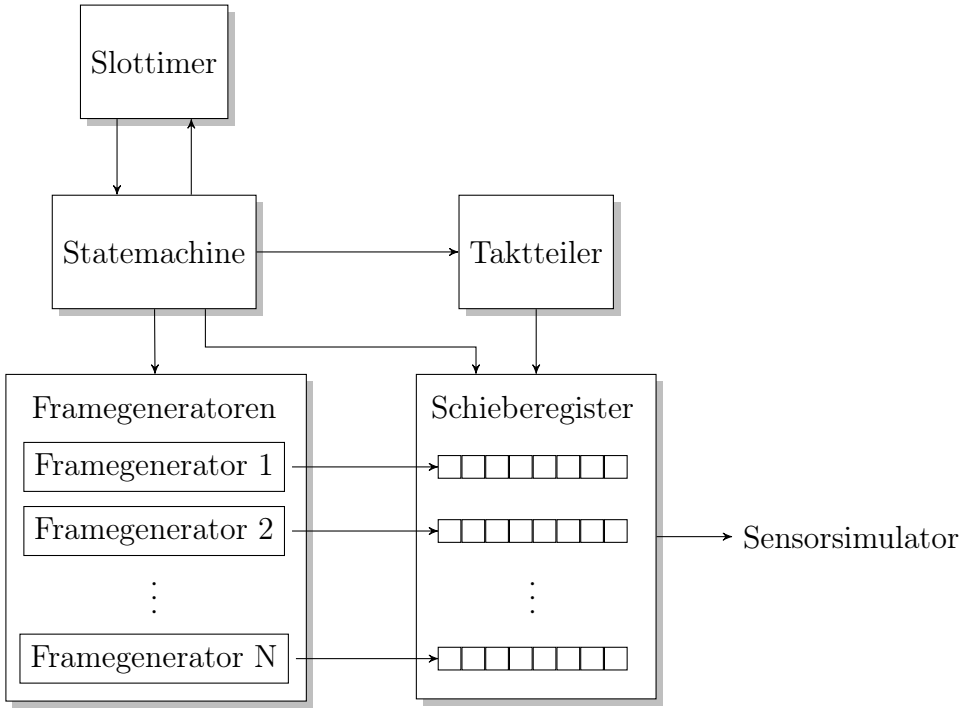


Abb. 4.6: Funktionsblöcke des Patterngenerators

erfolgt der Ablauf der Übertragung von Daten durch den Patterngenerator stufenweise durch die Funktionsblöcke. Zunächst wird ein Frame erzeugt, welcher dann in die Schieberegister geladen wird und anschließend mittels der Hardware des Sensorsimulators über den PSI5-Bus transferiert wird.
Der Framegenerator erstellt einen Frame aus den Daten und Vorgaben, welche die Software bereitstellt. Dabei existieren in einem PSI5 Patterngenerator-Block mehrere Framegeneratoren, sodass für jeden der Timeslots eines Kanals ein separater Framegenerator vorhanden ist. Die Frames werden, sobald der entsprechende Timeslot beginnt, sequentiell auf den Bus übertragen. Dazu werden die Schieberegister und ein Taktteiler zur Erzeugung des Bittaktes des Busses genutzt. Durch das Schieben der Daten aus dem Schieberegister mit einem definierten Bittakt, entstehen auf dem Bus Signalformen, welche einer Übertragung nach dem PSI5-Standard entsprechen.
Die Ablaufsteuerung der gesamten Übertragung wird von der primären Statemachine durchgeführt, welche auch, je nach Zustand, die Befehle der Software entgegen nimmt. Die primäre Statemachine ist das zentrale Element des PSI5 Patterngenerators, da sie mit allen weiteren Blöcken des Patterngenerators verbunden ist und deren Steuersignale bereitstellt, sowie die Auswertung selbiger durchführt. Der Patterngenerator ist dabei mit

nur einer primären Statemachine pro Kanal ausgestattet, welche die Verwaltung sämtlicher Timeslots eines Kanals übernimmt.
Das Design des PSI5 Patterngenerators ist, wie auch das Design anderer Blöcke, in der Anzahl der Kanäle variabel gestaltet. Um entsprechende Sprachkonstrukte nutzen zu können, welche die dynamische Anpassung von Komponenten erlauben, ist der PSI5 Patterngenerator in SystemVerilog geschrieben. Das vollständige Design des Blocks erhält jedoch einen statischen Wrapper und ein statisches AXI-Interface, welche die Anzahl der Kanäle und Timeslots pro Kanal festlegen. Im Testsystem ist daher passend zur Hardware gegenwärtig eine Konfiguration des PSI5 Patterngenerators genutzt, welche zehn Kanäle und bis zu vier Timeslots je Kanal besitzt. Eine Erweiterung um zusätzliche Kanäle ist zukünftig durch eine Erweiterung des AXI-Interfaces um weitere Register möglich.
Die folgenden Abschnitte beschreiben die Funktionsweise der einzelnen Komponenten des PSI5 Patterngenerators im Detail.

4.3.1 PSI5 Patterngenerator AXI Interface

Das AXI-Interface des PSI5 Patterngenerators besteht aus 180 Registern. Diese große Anzahl ergibt sich aus den zehn vorhandenen unabhängigen PSI5-Kanälen. Jedem dieser Kanäle sind 18 Register zugeordnet, wobei sich der Aufbau der Registerstruktur für die einzelnen Kanäle nicht unterscheidet, sondern nur für jeden weiteren Kanal dupliziert wurde. Auch innerhalb eines Kanals wiederholt sich der Aufbau der Registerstruktur, da jeder Kanal über mehrere identische Timeslots und deren Einstellungen verfügt. Die Tabelle 4.8 zeigt daher nur einen Registersatz, dessen Angaben in Abhängigkeit des Kanals und des Timeslots erfolgen.

Tabelle 4.8: Register des AXI-Interfaces des Patterngenerators

Register	**Bit(s)**	**R/W**	**Parameter**	**Funktion**
$K \cdot 18$	31	R	Error	Fehler - Timeout erreicht.
$K \cdot 18$	4	R/W	Trigger Mode	Sync-Impulsquelle
$K \cdot 18$	3	R	Done	Transferzyklus abgeschlossen
$K \cdot 18$	2	R/W	Trigger	Externer Trigger
$K \cdot 18$	1	W	Reset	Reset PCI5 Patterngenerator
$K \cdot 18$	0	R/W	Enable	Starte Patterngenerator
$K \cdot 18 + 1$	31-0	R/W	Timeout	Sync-Impuls Timeout Kanal [K]
$K \cdot 18 + S \cdot 4 + 2$	20-16	R/W	Payload Length	Nutzdatenlänge Kanal [K] Slot [S]
$K \cdot 18 + S \cdot 4 + 2$	3	R/W	Menc. Error	Manchester Fehlereinspeisung Kanal [K] Slot [S]
$K \cdot 18 + S \cdot 4 + 2$	2	R/W	Chskm. Error	Prüfsummen Fehlereinspeisung Kanal [K] Slot [S]
$K \cdot 18 + S \cdot 4 + 2$	1	R/W	Chksm. Mode	Prüfsummen Typenauswahl Kanal [K] Slot [S]
$K \cdot 18 + S \cdot 4 + 2$	0	R/W	Slot Enable	Aktivierung Timeslot Kanal [K] Slot [S]
$K \cdot 18 + S \cdot 4 + 3$	31-0	R/W	Startpoint	Startzeitpunkt Kanal [K] Timeslot [S]
$K \cdot 18 + S \cdot 4 + 4$	31-0	R/W	Clkdiv	Taktteilfaktor für Kanal [K] Timeslot [S]
$K \cdot 18 + S \cdot 4 + 5$	27-0	R/W	Payload	Nutzdaten von Kanal [K] Timeslot [S]

Für N Kanäle gilt K = 0, 1, 2, ..., N-1 Für M Timeslots gilt S = 0, 1, 2, ..., M-1
In diesem Testsystem N = 10, M = 4
*Alle nicht explizit aufgeführten Bits der Register sind unbenutzt

Da das Testsystem gegenwärtig so ausgelegt ist, dass es über zehn Kanäle mit je vier Timeslots verfügt, ergeben sich besagte 18 Register pro Kanal bzw. 180 Register insgesamt. Das erste Register eines jeden Kanals ist das Kontroll- und Statusregister, über welches die Software mit der Statemachine des Patterngenerators kommuniziert. Das enable-Bit dient der Software dabei zum Starten der Statemachine aus ihrem Ruhezustand, wodurch die Erstellung und der Transfer neuer Pattern eingeleitet wird. Die Statemachine kann im Gegenzug über die Done- und Error-Bits der Software mitteilen, ob ein Transfer abgeschlossen ist oder es zu Fehlern gekommen ist. Das Trigger-Mode-Bit stellt ein, welches Verhalten der externe Eingang des Blocks, welcher mit dem Hardware-Timer verbunden ist, aufweisen soll. Ist das Bit nicht gesetzt, so wirkt der externe Eingang wie das Enable-Bit des AXI-Interfaces. Ist das Trigger-Mode-Bit auf eins gesetzt, so ist der externe Eingang des Blocks ein asynchroner Trigger des PSI5-Busses. Dieser kann genutzt werden, um PSI5-Frames unabhängig von einem Sync-Impuls zu übertragen. Dazu erzeugt der Hardware-Timer zu einem vorher definierten Zeitpunkt einen Impuls. Dieser wird an den Patterngenerator weitergeleitet, der daraufhin den Transfer des Frames startet. Soll ein Frame ohne einen Sync-Impuls übertragen werden und der Start der Übertragung nicht von dem Hardware-Timer ausgelöst werden, so kann die Software die Übertragung durch das Setzen des Trigger-Bits anstoßen.
Jeder Timeslot eines Kanals kann separat über das Slot Enable-Bit aktiviert oder deaktiviert werden. Ist ein Slot deaktiviert, so wird dessen Timeslot von der Statemachine übergangen und mit dem nächsten Timeslot fortgefahren. Ist der deaktivierte Timeslot der letzte Timeslot eines Kanals, beendet die Statemachine den Transferzyklus vorzeitig. Dem Timeslot werden über das Payload-Register die Nutzdaten übergeben. Da diese jedoch in ihrer Länge variieren können, müssen diese immer die niederwertigsten Bits des Register belegen und über die fünf Payload Length-Bits muss die Software dem Patterngenerator die Länge der Nutzdaten mitteilen. Ebenfalls muss die Software den Startzeitpunkt des Timeslots in Relation zum Sync-Impuls über das Startpoint-Register und die Bitrate der Übertragung des Timeslots mittels des Clkdiv-Registers definieren.
Da der PSI5-Bus zwei Verfahren zur Bestimmung einer Prüfsumme erlaubt, muss die Software durch das Setzen des Chksm. Mode-Bit spezifizieren, welches Prüfsummenverfahren genutzt werden soll. Sofern eine CRC-Prüfsumme gewünscht ist, muss das Bit gesetzt werden, andernfalls wird ein Paritätsbit als Prüfsumme verwendet. Zusätzlich können bei Bedarf Fehler in den Frame des Timeslots eingespeist werden. Das Chskm. Error-Bit aktiviert die Einspeisung eines Fehlers in die Prüfsumme. Dabei wird ein Bit der Prüfsumme gekippt, sodass die Prüfsumme ungültig wird. Soll die Manchesterkodierung eines Frames gestört werden, so kann die Software das Menc. Error-Bit setzten, woraufhin der Patterngenerator die Bedingungen der Manchesterkodierung verletzt.

4.3.2 PSI5 Frame Erzeugung

Der Block zur Frameerzeugung setzt einen PSI5-Frame aus den einzelnen Eingangsgrößen entsprechend der Definition durch den Data Link Layer zusammen [PSI18a, S.12]. Darin sind die Positionen und Längen einzelner Bits bzw. Gruppen innerhalb eines Frames definiert. Von der in Abschnitt 2.3.1 dargestellten Definition eines Frames werden von diesem Block jedoch nur die Startbits, die Nutzdaten und die Prüfsumme berücksichtigt. Die Nutzdaten werden dabei abweichend von der Data Link Layer Definition als eine gesamte Gruppe betrachtet. Der Anwender ist dafür verantwortlich, die Nutzdaten so auszugestalten, dass dies den geforderten Bedingungen des Data Link Layers z.B. den Data Ranges entsprechen.
Der Block zur Frameerzeugung übernimmt nicht nur die Zusammenstellung eines Frames, sondern führt auch die Manchesterkodierung des Frames und, falls gefordert, eine Fehlereinspeisung durch. Das Ausgangssignal dieses Blocks enthält alle notwendigen Eigenschaften einer Übertragung des PSI5-Busses, sodass es in das Schieberegister des Patterngenerators zum Transfer auf den PSI5-Bus geladen werden kann. Die Frame Erzeugung läuft dabei schrittweise ab. Zunächst werden die Daten in ihrer Reihenfolge angepasst und die Startbits angefügt, anschließend wird die Prüfsumme des Frames bestimmt und angehängt. Die abschließende Operation ist die Manchesterkodierung des Frames.
Die Ablaufsteuerung innerhalb dieses Blocks zur Frameerzeugung wird durch eine in Abbildung 4.7 dargestellte Statemachine mit vier Zuständen bestimmt. Sie enthält nur einen

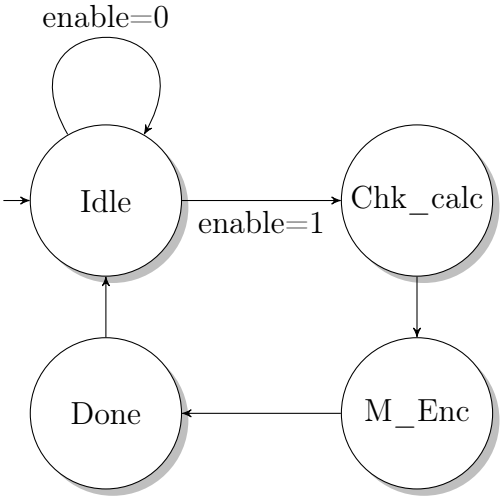


Abb. 4.7: Zustandsübergangsdiagramm des Framegenerators

Zustandsübergang, der durch ein Signal beeinflusst werden kann. Dieser befindet sich zwischen dem Idle-Zustand und dem Chk_calc-Zustand, welcher durch das Setzen des enable-Signals erreicht werden kann. Alle anderen Zustandsübergänge werden automatisch mit dem nächsten Takt passiert.
Im Idle-Zustand werden die Eingangsgrößen des Blocks in interne Register übernommen,

sodass eine nachträgliche Änderung der Eingangssignale während der Erstellung eines Frames nicht zu einem undefinierten oder ungültigen Inhalt des Frames führt. Die Eingangsgrößen sind die späteren Nutzdaten des Frames, die Länge selbiger, sowie drei Steuersignale, welche die Art der Prüfsumme und Fehlereinspeisungen definieren. Die Länge der Nutzdaten kann bis zu 28 Bit betragen (vgl. Abbildung 2.8 in Abschnitt 2.3.1). Im Idle-Zustand wird außerdem das Done-Signal zurückgesetzt, welches, falls gesetzt, der primären Statemachine des PSI5-Patterngenerators anzeigt, dass die Generierung eines Frames abgeschlossen ist.
Im Chk_calc-Zustand wird begonnen, den Frame zusammenzusetzen. Dazu werden in einem internen Register, welches in Abbildung 4.8 dargestellt ist, die Startbits und die Nutzdaten eingefügt. Die beiden Startbits belegen die höchstwertigsten Bits, da der Vek-

33	32	31	30	...	32-N	31-N	30-N	29-N	28-N
Start		Nutzdaten					Prüfsumme		
0	0	N Bits (10...28)					CRC[2]	CRC[1]	CRC[0]

Abb. 4.8: Belegung des internen Vektor des Framegenerators

tor so definiert ist, dass die ersten Bits, welche über den PSI5-Bus übertragen werden sollen, die höchstwertigsten Stellen belegen. Das Schieberegister des Patterngenerators kann den Inhalt dieses Vektors dann über eine Schiebeoperation nach links auf den Bus legen. Damit diese Bitordnung eingehalten wird, muss die Reihenfolge der Nutzdatenbits getauscht werden, sodass das niederwertigste Bits, wie durch den Data Link Layer gefordert, die Position des höchstwertigsten einnimmt und umgekehrt. Das niederwertigste Bit der Nutzdaten belegt damit im internen Vektor des Framegenerators aufgrund der statischen Anzahl an Startbits immer die Bitposition 31.
Im selben Chk_calc-Zustand wird außerdem die Prüfsumme, welche von dem in Abschnitt 4.3.3 beschriebenen Block bestimmt wird, in den Vektor aus Abbildung 4.8 eingesetzt. Die Position der Prüfsumme ist dabei variabel, da die Länge der Nutzdaten zwischen $10 - 28\,Bit$ betragen kann. Innerhalb des Chk_calc-Zustands muss die Länge der gültigen Bits aktualisiert werden, welche anzeigt, bis zu welcher Bitposition die Daten in dem Vektor den gültigen Daten des Frames entsprechen und welche folgenden Bits des Vektors nur als Platzhalter für etwaige größere Frames dienen. Diese Längenangabe entspricht der Länge der gültigen Nutzdatenbits plus den beiden Startbits und einem oder drei Bits für die Prüfsumme. Die Anzahl der Bits der Prüfsumme ist dabei abhängig von dem gewählten Verfahren zur Bestimmung einer Prüfsumme. Siehe dazu Abschnitt 4.3.3. Die Prüfsumme wird im Gegensatz zu den Nutzdaten mit dem höchstwertigsten Bit zuerst in den Vektor des Framegenerators eingebunden, da der Data Link Layer die Übertragung der Prüfsumme mit dem höchsgtwertigsten Bit zuerst vorschreibt. Im Chk_calc-Zustand wird weiterhin, sofern von der primären Statemachine des PSI5-Patterngenerators über

das chksm_error-Signal angefordert, ein Fehler in die Prüfsumme injiziert. Die Fehlereinspeisung funktioniert durch die Invertierung des ersten Bits der Prüfsumme, welches sicherstellt, dass die CRC-Prüfsumme oder das Paritätsbit ungültig werden.
Die Statemachine des Framegeneratorblocks wechselt dann automatisch mit dem nächsten Taktzyklus in den M_Enc-Zustand. In diesem wird der in Abbildung 4.8 dargestellte Vektor des Framegenerators, welcher den nun vollständig zusammengesetzten Frame enthält, manchesterkodiert. Da die Manchesterkodierung die Anzahl der Bits verdoppelt, indem sie jedem Daten-Bit einen Übergang zwischen zwei Bits zuweist, werden die Daten in einem zweiten Vektor gespeichert, welcher mit 66 Bit doppelt so groß ist wie ein PSI5-Frame mit maximaler Nutzdatenlänge. Die Manchesterkodierung wird durch eine XOR-Verknüpfung der Bits des Frames mit einer alternierenden Bitfolge durchgeführt. Dabei wird jedes Bit des Frames zunächst mit einer Null und anschließend einer Eins XOR-verknüpft. Die dabei entstehende Bitfolge entspricht dem Signalverlauf der Stromaufnahme des PSI5-Busses während einer Datenübertragung. Analog zur Bestimmung der Länge des Frames im Chk_calc-Zustand, muss im M_Enc-Zustand die Länge der gültigen Bits im manchesterkodierten Frame bestimmt werden. Da die Länge der manchesterkodierten Daten exakt doppelt so groß ist wie die Länge des Frames, reicht an dieser Stelle eine einfache Shiftoperation der vorherigen Länge um ein Bit nach links. Auch die Manchesterkodierung bietet die Möglichkeit einer Fehlereinspeisung. Dazu werden, wenn das menc_error-Bit gesetzt ist, im M_Enc-Zustand drei Bits des manchesterkodierten Frames durch drei identische Bits ersetzt. Drei aufeinanderfolgende identische Bits sind eine Verletzung der Bedingungen der Manchesterkodierung, da diese spätestens nach zwei identischen Bits den Übergang zu einem invertierten Bit vorsieht.
Mit Abschluss der Manchesterkodierung wechselt die Statemachine in den Done-Zustand, in welchem das Done-Signal für einen Takt gesetzt wird, bevor die Statemachine in den Idle-Zustand zurückkehrt. Jeder Kanal ist für jeden seiner Timeslots mit einer eigenen Instanz des Blocks zur Erzeugung des PSI5 Frames ausgestattet. Existieren neue Daten, welche übertragen werden sollen, können so parallel die Frames für die jeweiligen Timeslots erzeugt werden, ohne dass Timeslots auf die Fertigstellung des Frames eines anderen Timeslots warten müssen.

4.3.3 Prüfsumme

Die Erzeugung der Prüfsumme für den PSI5-Frame wird in einem eigenen Block durchgeführt. Dieser Block kann sowohl das Paritätsbit als auch die 3-Bit-CRC bestimmen und deckt damit beide möglichen Sicherungsmethoden eines PSI5-Frames ab. Der Block zur Generierung des PSI5-Frames teilt diesem Block durch das Setzen des chksmode-Bits mit, welche Art der Prüfsumme erzeugt wird. Ist das Bit gesetzt, bestimmt der Block

die 3-Bit-CRC anhand des durch den PSI5-Standard definierten Polynoms. Siehe dazu Abschnitt 2.3.2. Der Ausgangsvektor, welcher drei Bit breit ist, enthält die drei CRC-Bits beginnend mit dem höchstwertigsten Bit.
Ist als Sicherungsmethode das Paritätsbit gewünscht und dementsprechend das chksmode-Bit nicht gesetzt, besteht das drei Bit breite Ausgangssignal dieses Blocks, welches die Prüfsumme enthält, aus dem Paritätsbit und zwei folgenden dummy Bits zum Auffüllen des Vektors. Der Block zur Generierung des PSI5-Frames muss in diesem Fall sicherstellen, dass er die letzten beiden Bits verwirft und nicht an den Frame anhängt.
Neben dem 28 *Bit* breiten Vektor, welcher den Frame beinhaltet für den die Prüfsumme bestimmt werden soll, besteht die Eingangsgröße des Blocks außerdem aus einem fünf Bit breiten Vektor, welcher die Länge des Frames anzeigt. Durch dieses Signal wird dem Block zur Generierung der Prüfsumme mitgeteilt, welche Bits des Frames gültig sind und zur Berechnung der Prüfsumme herangezogen werden müssen. Die Daten des Frames sind in dem Vektor, beginnend mit dem niederwertigsten Bit, abgelegt. Da der PSI5-Bus über variable Nachrichtenlängen verfügt, werden gegebenenfalls nicht alle Bits bis zum höchstwertigsten Bit des Vektors genutzt. Die Länge zeigt daher das letzte gültige Bit an, welches betrachtet werden muss.
Durch den PSI5-Standard ist definiert, dass das Paritätsbit zu einer geraden Bitsumme führt. Dies bedeutet, dass bei einer ungeraden Anzahl an gesetzten Bits in dem PSI5-Frame eine weitere Eins als Paritätsbit angehängt wird. Die Bestimmung des Paritätsbits erfolgt durch eine XOR-Verknüpfung aller gültigen Bits eines Frames miteinander. Das entstehende Signal der Verknüpfung entspricht dabei direkt dem geraden Paritätsbit.
Für die Berechnung der 3-Bit-CRC wird das Polynom $g(x) = x^3 + x + 1$ genutzt. Dieses lässt sich durch drei Speicher bzw. Verzögerungen und zwei XOR-Operationen darstellen. Die entsprechende Topologie ist in Abbildung 4.9 dargestellt. Die Daten werden dabei

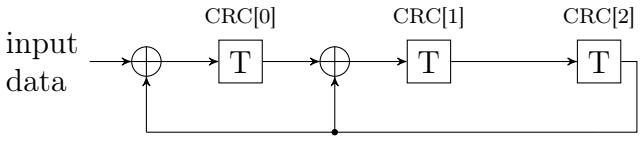


Abb. 4.9: Topologie des CRC-Polynoms, modifziert nach [PSI18a, S. 16]

bitweise beginnend mit dem niederwertigsten Bit in das Generatorpolynom geschoben. Die drei Speicher CRC[0] bis CRC[2] enthalten das Ergebnis des gegenwärtig ausgeführten CRC-Berechnungsschritts. Ist die Berechnung abgeschlossen, befindet sich in diesen Speichern die gesuchte CRC-Prüfsumme für den PSI5-Frame. Die Berechnung ist abgeschlossen, wenn der Berechnungsschritt für alle gültigen Bits und drei abschließende Bits mit dem Wert null durchgeführt wurde (vgl. [PSI18a, S. 15f.]). Im Standard definiert ist der initiale Wert 0b111, welchen die drei Speicher CRC[0] bis CRC[2] vor Beginn der Berechnung einer Prüfsumme enthalten müssen.

Die Umsetzung der Berechnung der 3-Bit-CRC-Prüfsumme erfolgt durch zwei Funktionen, eine Funktion zur Berechnung eines einzelnen CRC-Schritts und einer Verwaltungsfunktion. Die Funktion zur Berechnung eines CRCs führt in Abhängigkeit der drei Speicher CRC[0] bis CRC[2] und dem neuen Eingangsbit die XOR- und Schiebeoperation eines einzelnen CRC-Schritts aus. Diese wird von der Verwaltungsfunktion entsprechend der Anzahl der gültigen Bits mehrfach aufgerufen. Die Verwaltungsfunktion übernimmt ebenfalls die Vorinitialisierung der drei Speicher, sowie das Anhängen der drei abschließenden Bits. Die Bestimmung der Prüfsumme erfolgt in der praktischen Umsetzung für alle Bits in einem Taktzyklus. Dies bedeutet, dass wenn sich die Eingangsignale des Blocks ändern, bereits mit der nächsten steigenden Taktflanke die neue Prüfsumme am Ausgang anliegt. Dazu wird die Berechnung nicht durch die in Abbildung 4.9 dargestellte Architektur und ihre Schiebeoperation durchgeführt, auch wenn diese in der Funktion zur Bestimmung der CRC implementiert ist. Der Aufruf dieser Funktion erfolgt durch eine For-Schleife ohne Verzögerung, welche parallele Instanzen der Funktion erzeugt. Durch die For-Schleife wird während der Synthese aus der in Abbildung 4.9 dargestellten Architektur eine Logikschaltung, welche alle Bits parallel erfasst und verarbeitet.
Dies führt zu einem hohen Logikverbrauch im FPGA. Gegenwärtig ist die Ausnutzung der verfügbaren Logikzellen des FPGAs noch nicht so hoch, dass für Erweiterungen keine freien Zellen mehr zur Verfügung stünden. Werden jedoch in Zukunft weitere Logikzellen benötigt, so kann hier eine große Einsparung erreicht werden, indem eine Pipeline-Architektur implementiert wird, in welcher in jedem Taktzyklus nur ein Schritt der CRC-Berechnung durchgeführt wird.
Ist das Resetsignal des Blocks gesetzt, so gibt dieser unabhängig von den Eingangsdaten einen Vektor zurück, welcher aus drei nicht gesetzten Bits besteht.

4.3.4 PSI5 Schieberegister

Die Daten des Signalverlaufs einer PSI5-Nachricht werden in einem Schieberegister gespeichert und dann bitweise auf den Bus gelegt, sodass der Signalverlauf des entsprechenden PSI5-Frames entsteht. In dem Schieberegister werden die manchesterkodierten Daten des Frames abgelegt, welche durch den in Abschnitt 4.3.2 beschriebenen Framegenerator-Block erzeugt wurden, sodass das Schieberegister eine Länge von 66 Bit aufweist.
Alle benötigten Ausgangs-Schieberegister des PSI5-Patterngenerators sind in diesem Block zusammengefügt, sodass dieser nicht nur ein einzelnes Schieberegister enthält, sondern für jeden Timeslot des Kanals ein separates Register. Die maximale Anzahl der Timeslots pro Kanal des PSI5-Patterngenerators kann zum Synthesezeitpunkt durch einen Parameter variiert werden. Für diesen Block führt dies zu einer Addition oder Subtraktion weiterer Schieberegister, sowie einer Anpassung der Eingangsgröße. Diese ist neben den üblichen

Signalen wie Takt, Reset und Steuersignalen ein Array, welches die Bits der Nachrichten eines jeden Timeslots enthält. Durch das Setzen des preload-Steuersignals kann dieses Array in die einzelnen Schieberegister übernommen werden. Das Array der Eingangsdaten ist dabei in der einen Dimension immer 66 *Bit* breit, da dies der Länge der Schieberegister entspricht. Die Dimension ist abhängig von der Anzahl der Timeslots bzw. Schieberegister auf welche das Array aufgeteilt wird.
Alle Schieberegister teilen sich das selbe preload-Steuersignal, da die Frames aller Timeslots simultan durch die Framegeneratoren erzeugt werden und die Schieberegister ebenfalls parallel mit den Daten beladen werden sollen. Siehe dazu auch Abschnitt 4.3.2. Während das preload-Steuersignal oder das Reset-Signal aktiv sind, ist das Ausgangssignal dieses Blocks auf null gesetzt.
Der Block verfügt nur über ein Ausgangssignal, da die Ausgänge der einzelnen Schieberegister über einen Multiplexer damit verbunden sind (siehe Abbildung 4.10). Dies

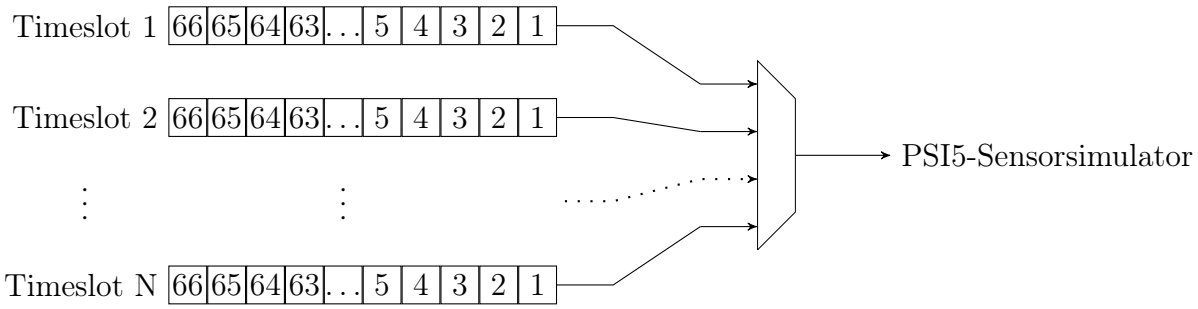


Abb. 4.10: Schieberegister und Multiplexer des Patterngenerators

ermöglicht die Speicherung der Frames der unterschiedlichen Timeslots eines Kanals in verschiedenen Schieberegistern und die sequenzielle Gewährung eines exklusiven Zugriffs der Register auf den Bus. Der enable-Vektor des Multiplexers, von dem jeweils immer nur ein Bit entsprechend dem aktuellen Timeslot aktiv ist, wird dabei ebenfalls genutzt, um auszuwählen, welches Schieberegister aktiv sein soll. So ist sichergestellt, dass immer das Schieberegister aktiv ist, welchem gegenwärtig über den Multiplexer Zugriff auf den Bus gewährt wird. Ist der Frame eines Timeslots vollständig auf den Bus übertragen worden, kann die primäre Statemachine des PSI5-Patterngenerators durch die Auswahl des nächsten enable-Signals das Schieberegister mit dem Frame des nächsten Timeslots auswählen und ihm den Zugriff auf den Bus gewähren.
Das Auslösen der eigentlichen Schiebeoperation des aktuell gewählten Schieberegisters erfolgt jedoch nicht durch die Statemachine, sondern über das shift-Signal, welches von dem PSI5 Taktteiler erzeugt wird (vlg. Abschnitt 4.3.5). Dieser Taktteiler definiert die Frequenz, mit welcher die Schiebeoperation durchgeführt wird und die daraus resultierende Bitrate des PSI5-Busses.

PSI5 Bitzähler

Die PSI5-Nachrichten können in ihrer Länge mit Startbits und Prüfsumme zwischen 13 - 33 *Bit* variieren. Die Schieberegister des Patterngenerators sind daher auf die maximale Länge eines Frames ausgelegt. Da das Schieberegister keine Informationen besitzt, bis zu welcher Position des Registers die Bits gültige Daten eines Frames entsprechen und ab welchem Punkt sich nur noch Platzhalter in dem Schieberegister befinden, existiert dieser zusätzliche Bitzähler. Er zählt jede Schiebeoperation des aktiven Schieberegisters mit. Erreicht der Zählerstand die Anzahl der im aktuell aktiven Timeslot vorhandenen manchesterkodierten Bits, setzt der Bitzähler das shift_done-Signal, um der primären Statemachine des Patterngenerators mitzuteilen, dass alle gültigen Bits des aktuellen Timeslots auf den Bus geschoben wurden. Die Statemachine kann daraufhin die Generierung der Steuerimpulse für das Schieberegister durch den Taktteiler beenden.
Der Bitzähler erhält den Wert der gültigen Bits eines Frames bzw. dessen Länge für jeden Timeslot von dem Framegenerator, welcher diese Längen, wie in Abschnitt 4.3.2 beschrieben, bereits während der Generierung der Frames bestimmt. Der Bitzähler erhält über ein Array die Längen aller Frames der Timeslots eines Kanals simultan und wählt dann über einen Multiplexer die Länge des gegenwärtig aktiven Timeslots aus. Dazu nutzt der Bitzähler den selben enable-Vektor, welcher auch von dem Schieberegisterblock genutzt wird, um das korrekte Schieberegister für den gegenwärtigen Timeslot zu wählen.
Ist das reset-Signal des Bitzählers gesetzt, wird der Zähler auf Null zurückgesetzt und das shift_done-Signal ebenfalls gelöscht, um zu verhindern, dass nach einem, durch einen Reset des Patterngenerators ausgelösten, Abbruch einer Übertragung, die Schieberegister und der Bitzähler fehlerhafte Zustände oder Zählwerte besitzen.

4.3.5 PSI5 Taktteiler

Der Taktteiler des PSI5-Patterngenerators erzeugt zwei Größen für das System. Die erste ist der Bittakt mit einem Tastgrad von 50% und einer durch den Haupttakt und einem Teilfaktor bestimmten Frequenz. Die Frequenz gibt dabei die Datenrate des aktiven Zeitslots des PSI5-Busses vor, wird jedoch von diesem System nur zu Diagnosezwecken verwendet. Sie ist mit den GPIO-Anschlüssen der Hauptplatine verbunden, sodass z.B. bei einer Untersuchung des Busses mit einem Oszilloskop der Bittakt als digitales Hilfssignal bereit steht. Das zweite von dem Taktteiler des PSI5-Patterngenerators erzeugte Signal sind Steuerimpulse, welche jeweils bei einer steigenden und fallenden Flanke des oben genannten Bittaktes erzeugt werden. Diese sind mit dem aktiven Schieberegister des PSI5-Patterngenerators verbunden und erzeugen die eigentliche Datenrate des Busses.

Da der PSI5-Bus manchesterkodiert ist, werden Impulse auf der jeweils steigenden und fallenden Flanke benötigt, da jedem Datenbit ein Zustandsübergang aus zwei Bits im Schieberegister zugeordnet ist.
Intern besteht der Taktteiler aus einem Zähler, dessen Auflösung zum Zeitpunkt der Synthese durch einen Parameter angepasst werden kann. Ist das Enable-Bit des Taktteilers gesetzt, wird der Wert des Zählers mit jedem Haupttakt um eins erhöht. Erreicht der Zähler den Wert, welcher am Eingang des Blocks als Teilfaktor anliegt, so wird das Ausgangssignal des Bittakts und ebenso das für den nächsten Takt andauernde Steuersignal des Schieberegisters gesetzt. Der Teiler wird beim Erreichen dieses Wertes nicht auf null zurückgesetzt, sondern zählt solange weiter, bis er den doppelten Wert des Teilfaktors erreicht hat. An diesem Punkt, wie in Abbildung 4.11 am Übergang zwischen Takt 8 und 9 bzw. 1 gezeigt, wird das Ausgangssignal des gesetzten Bittakts zurückgenommen, sodass dessen fallende Flanke entsteht und ein ein-Haupttakt-andauernder Impuls für das Schieberegister erzeugt wird. Der minimal zulässige Teilfaktor, welcher als Eingangssignal

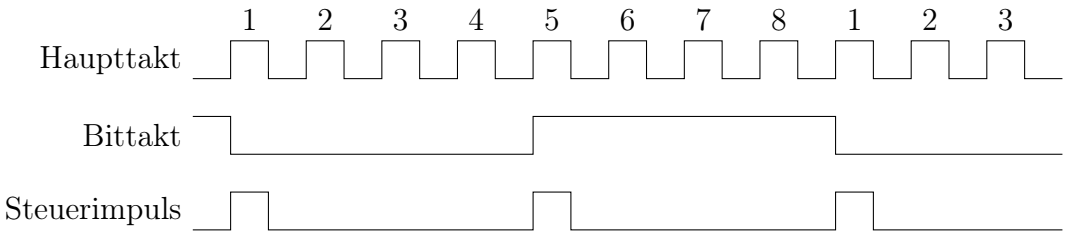


Abb. 4.11: Signale des PSI5-Taktteilers (Teilfaktor 4)

an diesen Block angelegt werden darf, ist eins. Bei einem Wert von null würde andernfalls die Schiebeoperation zur Verdoppelung des Wertes nicht funktionieren. Innerhalb des Taktteilers kommt es neben dem gewünschten Teilfaktor durch die Verdoppelung für die Manchesterkodierung zu einer Halbierung des Taktes. Soll bei einer Eingangsfrequenz von $100\,MHz$ eine PSI5-Bitrate bzw. Frequenz von $125\,kHz$ erreicht werden, so ist ein Teilfaktor von 400 zu wählen, damit die Frequenz der Impulse $250\,kHz$ beträgt, da durch die Manchesterkodierung die doppelte Bandbreite der eigentlichen Bitrate benötigt wird. Ist das Enable-Bit des Taktteilers nicht gesetzt, so werden beide Ausgangssignale zurückgesetzt und der Zähler mit dem Teilfaktor vorinitialisiert. Diese Vorinitialisierung erfolgt, da die Statemachine erwartet, dass, wenn sie den Taktteiler startet, direkt eine Schiebeoperation des Schieberegisters des PSI5-Patterngenerators durchgeführt wird. Da der Zähler bereits im Takt seiner Aktivierung durch die Vorinitialisierung den entsprechenden Vergleichswert erreicht, erzeugt der Taktteiler im selben Takt eine steigende Flanke des Bittaktes und einen Steuerimpuls für das Schieberegister.
Das Eingangssignal des PSI5-Taktteilers, welches die Vorgabe für den Teilfaktor macht, ist über einen Multiplexer mit dem Teilfaktor des aktuellen Timeslots verbunden. Dies ermöglicht es unterschiedliche Teilfaktoren und die daraus resultierenden unterschiedli-

chen Bitraten für jeden der Timeslots zu definieren. Die primäre Statemachine des PSI5-Patterngenerators übernimmt dabei die Auswahl des aktuell zu verwendenden Vorgabewertes bzw. die Ansteuerung des Multiplexers.

4.3.6 PSI5 Slot Timer

Der PSI5 Slot Timer erzeugt die Steuersignale, welche der primären Statemachine anzeigen, wann welcher Timeslot des PSI5-Busses im synchronen Betriebsmodus beginnt oder aktiv ist. Die Timeslots sind durch feste Zeitpunkte in Relation zum Zeitpunkt des Sync-Impulses des PSI5-Busses definiert (vgl. Abschnitt 2.2.2). Um diese Timeslots einzuhalten, benötigt die Statemachine Informationen über den Beginn eines neuen Timeslots. Dazu werden dem PSI5 Slot Timer vom Anwender die Startzeitpunkte der einzelnen Timeslots übergeben. Der Slot Timer besitzt intern einen Zähler, welcher, wenn gestartet, kontinuierlich hochzählt. Erreicht der Zähler des Slot Timers einen Startzeitpunkt eines der Timeslots, wird das entsprechende, einem Timeslot zugeordnete, Signal gesetzt. Der Zähler wird dabei durch den Haupttakt der programmierbaren Logik gespeist, sodass die Angabe der Startzeitpunkte im Bezug auf die Frequenz des Haupttaktes erfolgen muss. Die primäre Statemachine legt den Zeitpunkt fest, ab welchem der Zähler anfängt zu beginnen. Der Zeitpunkt entspricht dem Auftreten des Sync-Impulses des PSI5-Busses, welcher durch die Detektorschaltung und die Statemachine ausgewertet wird. Die Statemachine setzt daraufhin das enable-Signal des PSI5 Slot Timers, sodass dieser beginnt den Zeitraum zwischen dem Sync-Impuls und dem Beginn der Timeslots des PSI5-Busses zu zählen. Das Ausgangssignal des PSI5 Slot Timers ist ein Vektor, welcher für jeden Timeslot ein Bit enthält. Ist ein Bit gesetzt, bedeutet dies, dass der zugehörige Timeslot gerade aktiv ist. Ist kein Timeslot aktiv, z.B. weil sich das System in dem Zeitraum nach einem Sync-Impuls, aber vor dem ersten Timeslot befindet, so ist keins der Bits gesetzt. Eine dezimale Repräsentation des aktuell aktiven Timeslots kann durch die Wandlung der One-Hot-Kodierung des Vektors erreicht werden.
Sollte es zu einer fehlerhaften Vorgabe kommen, bei welcher zwei Timeslots mit dem selben Startpunkt angegeben werden, setzt der Timer das Bit des Timeslots, welcher sich in der Reihenfolge weiter hinten befindet. So wird sichergestellt, dass immer nur ein Timeslot aktiv ist, da von dem Signal, welches den aktiven Timeslot anzeigt, viele interne Funktionen des Patterngenerators, wie die Auswahl des passenden Schieberegisters, abhängen. Das selbe Verhalten zur Sicherstellung nur eines aktiven Slots weißt der Slot Timer auch bei sich überlappenden Timeslots auf. Da der Timer nicht weiß, wann ein Timeslot zu Ende ist, wird mit Erreichen eines neuen Timeslots sofort das Bit des vorherigen Timeslots zurückgesetzt, auch wenn dessen Transfer seines Frames noch nicht abgeschlossen ist. Dies führt dazu, dass, sollten sich Timeslots überlappen, die Statemachine die Schiebeoperatio-

nen des vorherigen Timeslots noch zu Ende durchführen lässt, dann jedoch sofort mit der Übertragung der Daten aus dem neuen Schieberegister, sowie dessen Bittakt beginnt. Der PSI5-Patterngenerator stellt beim Wechsel zwischen den Frames der sich überlappenden Timeslots nicht sicher, dass die Manchesterkodierung eingehalten wird.
Der Anwender hat durch eine geeignete Wahl der Startzeitpunkte der Timeslots dafür zu sorgen, dass der Transfer des vorherigen Timeslots abgeschlossen ist und es zu keinen Kollisionen kommt. Abgesehen von unerwünschten Signalfolgen auf dem Bus, führt eine Kollision der Timeslots aber zu keinen internen Problemen für den Patterngenerator, sodass eine solche Konfiguration gegebenenfalls genutzt werden kann, um zu untersuchen, wie das DUT auf die Kollision von Timeslots verschiedener Sensoren reagiert.

4.3.7 PSI5 primäre Statemachine

Die primäre Statemachine des PSI5-Patterngenerators ist verantwortlich für die komplette Ablaufsteuerung des Patterngenerators von der Erzeugung des Frames bis zur eigentlichen Übertragung der einzelnen Timeslots durch den Sensorsimulator auf den Bus. Die Statemachine besteht aus acht Zuständen, wie in Abbildung 4.12 zu sehen, und zwei zusätzlichen Zählern. Der erste Zähler dient der Umsetzung eines Timeouts, um zu verhindern, dass die Statemachine in einem Zustand hängen bleibt, wenn die erwarteten Signale ausbleiben. Der zweite Zähler wird genutzt, damit die Statemachine Informationen erhält, wie viele der Timeslots bereits abgearbeitet sind und ob das Ende eines Transferzyklusses des PSI5-Busses erreicht ist.
Die Statemachine startet im Idle-Zustand. In diesem Zustand werden sämtliche Steuersignale zurückgesetzt, so dass ein definierter Ausgangszustand hergestellt wird. Verlassen werden kann der Idle-Zustand nur durch das Setzen des enable-Signals des Top-Levels des Patterngenerators. Dieses ist mit dem AXI-Interface des Blocks verbunden, sodass die Software den Start der Statemachine veranlassen kann. Durch das Setzen des enable-Signals wechselt die Statemachine in den Init-Zustand.
Der Init-Zustand dient der Initialisierung des Patterngenerators, in welchem die zu übertragenen Frames des PSI5-Busses erstellt werden. Die Statemachine setzt dazu das enable-Signal des Framegenerators, woraufhin dieser beginnt die Frames der einzelnen Timeslots zu erstellen. Die Vorgaben für die Nutzdaten und anderen Konfigurationen des Frames werden von der Software über das AXI-Interface direkt an den Framegenerator gegeben, die Statemachine ist nur für das Auslösen der Generierung der Frames verantwortlich. Da der Framegenerator die Frames aller Timeslots parallel erzeugt, verfügt er nur über ein Steuersignal, welches von der Statemachine ausgewertet werden muss. Das Steuersignal ist das Done-Signal der Statemachine des Framegenerators, welches die Fertigstellung der Generierung der Frames anzeigt. Ist dieses gesetzt, wechselt die primäre Statemachine des

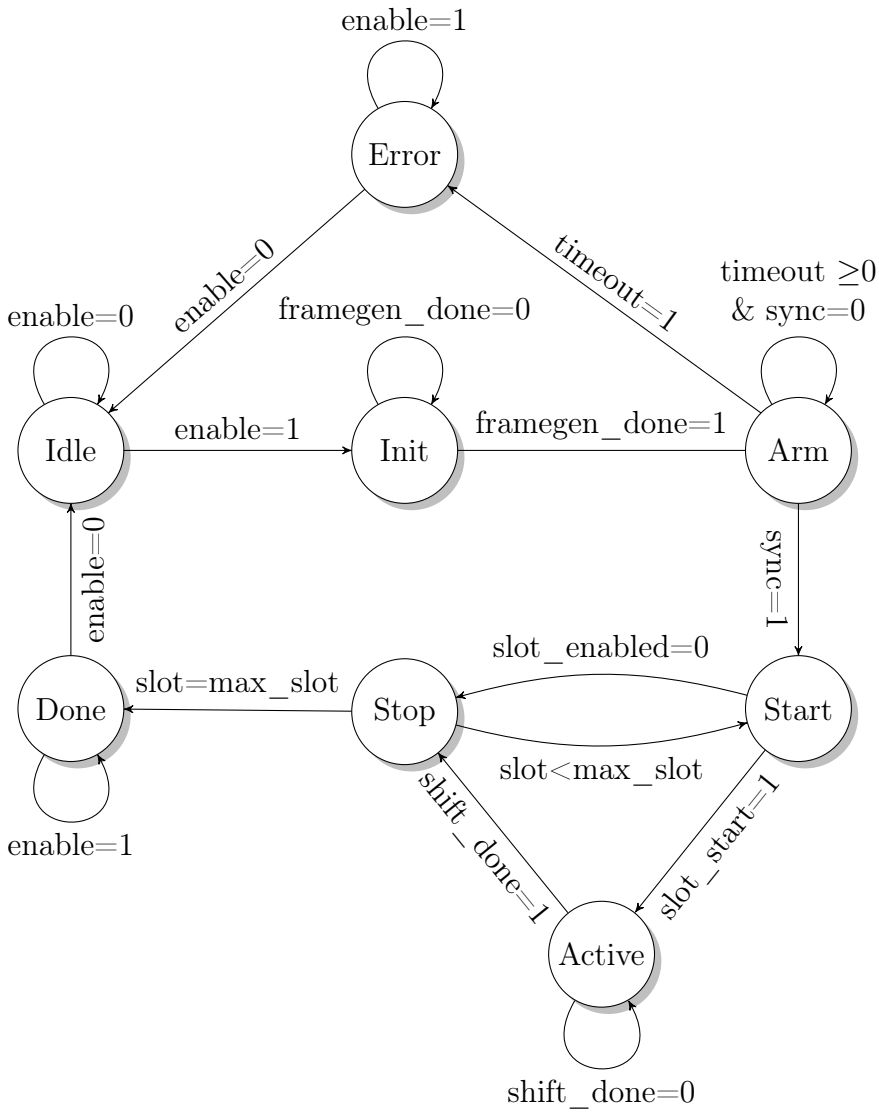


Abb. 4.12

PSI5-Patterngenerators in den Arm-Zustand.

Im ARM-Zustand setzt die Statemachine das preload-Signal des Schieberegister-Blocks, sodass in die Schieberegister die Frames der einzelnen Timeslots geladen werden, welche im vorherigen Schritt erzeugt wurden. Dieser Ladevorgang ist bereits mit einem Takt abgeschlossen, die Statemachine verbleibt jedoch in dem Arm-Zustand solange, bis die Detektorschaltung einen Syncimpuls des PSI5-Busses erkannt hat. Wurde ein Sync-Impuls, welcher den Startpunkt einen Übertragungszyklus markiert, erkannt, wechselt die Statemachine in den Start-Zustand.

Da es jedoch vorkommen kann, dass das DUT keinen Sync-Impuls erzeugt, muss sichergestellt werden, dass es einen Weg gibt, auf welchem die Statemachine den gegenwärtigen Arm-Zustand verlassen kann, ohne einen vollständigen Reset auszuführen. Der Statemachine wird dazu ein Timeout übergeben. Dieser wird mit dem Zählerstand des Timeout-Zählers verglichen, welcher von der Statemachine gestartet wird, sobald sie den Arm-

Zustand erreicht. Entspricht der Zählerstand dem Timeout-Wert, setzt der Zähler das timeout-Signal, welches die Übergangsbedingung der Statemachine in den Error-Zustand ist, in den dann automatisch gewechselt wird. Der Error-Zustand entspricht dem Done-Zustand bis auf den Unterschied, dass im Error-Zustand das error-Signal gesetzt wird, welches der Software mitteilt, dass für den entsprechenden Kanal der Timeout erreicht wurde und der Transfer der Frames nicht stattgefunden hat. Der Error-Zustand kann durch das Zurücksetzen des enable-Signals verlassen werden, woraufhin die Statemachine in den Ausgangszustand Idle zurückkehrt.
Der Start-Zustand, in welchem sich die Statemachine nach einem Sync-Impuls befindet, wird genutzt, um den Slot-Timer zu starten und zu bestimmen, welcher Frame bzw. Timeslot als nächstes übertragen werden soll. Um den Slot-Timer des Patterngenerators zu starten, wird das enable-Signal des in Abschnitt 4.3.6 beschriebenen Timers durch die Statemachine gesetzt. In Abhängigkeit der nächsten zu übertragenden Frames und des Zustands des Slot-Timers wird der nächste Zustand der Statemachine bestimmt, in welchem die eigentliche Übertragung stattfindet. Ist der gegenwärtige Timeslot durch die Einstellungen der Software deaktiviert, wechselt die Statemachine direkt vom Start-Zustand in den Stop-Zustand und der deaktivierte Timeslot wird als beendet deklariert. Ist der gegenwärtige ausgewählte Timeslot durch die Software aktiviert worden, so wartet die Statemachine im Start-Zustand so lange, bis der Slot-Timer signalisiert, dass das Zeitfenster des ausgewählten Timeslots beginnt.
Anschließend wechselt die Statemachine in den Active-Zustand, in welchem der eigentliche Datentransfer stattfindet. In diesem wird der Taktteiler des PSI5-Patterngenerators gestartet, welche das Steuersignal für das in Abschnitt 4.3.4 beschriebene Schieberegister erzeugt, das schlussendlich die Daten des Frames während des Active-Zustands auf den Bus schiebt. Der Slot-Timer bleibt während dieses Zustands aktiv, da er den Startzeitpunkt des nächsten Timeslots in Relation zum Sync-Impuls bestimmt und nicht als Zeitangabe zwischen den beiden Frames. Der Active-Zustand wird verlassen, wenn der Zähler des Blocks mit dem Schieberegister durch das shift_done-Signal anzeigt, dass alle gültigen Bits des gegenwärtigen Timeslots auf den Bus geschoben wurden. Dies kann je nach Nachrichtenlänge der Übertragung verschieden lang dauern, sodass die Statemachine auf das Signal des zusätzlichen Zählers des Schieberegisters angewiesen ist. Der nächste Zustand ist unabhängig von der Dauer des eigentlichen Datentransfers immer der Stop-Zustand.
Erreicht die Statemachine den Stop-Zustand, wird der Zähler der Statemachine, welcher mitzählt, wie viele Timeslots eines Transferzyklusses bereits stattgefunden haben, erhöht, da ein weiterer Timeslot abgeschlossen worden ist. Die Statemachine deaktiviert den Taktteiler, sodass keinen weiteren Steuerimpulse für die Schieberegister erzeugt werden. Andernfalls würde das aktive Schieberegister ungültige Daten auf den Bus schieben, da es selbst keine Informationen darüber besitzt, wie viel Bits ein Frame lang ist. Zeigt

der gegenwärtige Zählerstand des Zählers der Statemachine für die übertragenen Timeslots, dass der letzte, durch die Software als aktiv gekennzeichnete, Timeslot übertragen wurde, wechselt die Statemachine in den Done-Zustand, welcher das Ende eines kompletten Transferzyklusses anzeigt. Andernfalls kehrt die Statemachine in den Start-Zustand zurück und beginnt mit den Vorbereitungen für den Transfer des nächsten Timeslots. Die Kette aus den Start, Active und Stop-Zuständen wird dabei so lange wiederholt, bis alle Timeslots abgearbeitet sind. Der Slot-Timer bleibt daher auch im Stop-Zustand aktiv, da gegebenenfalls noch ein Timeslot folgt.
Erst im Done-Zustand wird der Slot-Timer als letzter Block deaktiviert und das Done-Signal gesetzt, welches der Software signalisiert, dass der Transferzyklus abgeschlossen ist und im Gegensatz zum Error-Zustand kein Timeout als Fehler aufgetreten ist. Die Statemachine verbleibt im Done-Zustand so lange, bis die Software das enable-Signal zurücknimmt. Damit wird sichergestellt, dass die Software das Done-Signal zur Kenntnis nimmt, welches andernfalls nur einen Takt lang gesetzt wäre. Dieses extrem kurze Signal würde nur zufällig von der Software erkannt werden, wenn diese im exakt richtigen Takt das Register des AXI-Interfaces ausliest. Die zweite Problematik, welche durch den Verbleib der Statemachine im Done-Zustand gelöst wird, ist der ungewollte Neustart einer Übertragung durch ein zu lange gesetztes enable-Signal der Statemachine, welches dann, da kein Sync-Impuls vom DUT erzeugt wird, direkt zu einem Timeout bzw. Fehlerzustand führen würde.

4.4 Fehlereinspeisung

Die Ansteuerung der Fehlereinspeisung funktioniert auf Hardwareebene durch einfache GPOs des FPGAs, welche mit den MOSFETs oder Photorelais verbunden sind. Siehe dazu auch Abschnitt 3.4. Der FPGA muss kein Protokoll implementieren, sondern nur bei Bedarf den entsprechenden Ausgangspin einschalten.
Der Block, welcher die Logik zur Fehlereinspeisung beinhaltet, ist daher im Vergleich zu den anderen eingesetzten IP-Blöcken einfach aufgebaut und beinhaltet keine Statemachine. Der Einsatz eines Standard AXI-GPIO-Blocks von Xilinx ist dennoch nicht möglich, da der Block nicht nur die Ansteuerung der GPOs übernimmt, sondern auch verhindern muss, dass es durch die Fehlereinspeisung des Testsystems zu einem Kurzschluss zwischen der positiven Versorgungsleitung und der Masseleitung kommt. Dieser entsteht zum Beispiel, wenn beide MOSFETs der Fehlereinspeisungs-Halbbrücke simultan eingeschaltet sind.
Der Block, welcher die Gültigkeit der Konfiguration prüft, ist in der Anzahl der unterstützten Kanäle über einen Parameter variabel. Das AXI-Interface, welches den Block einbindet, erzeugt dessen Instanz jedoch mit einer statischen Anzahl von zehn Kanälen.

Es verfügt über fünf Register, wobei die letzten vier aus den Vektoren einer Fehlereinspeisung bestehen. Das Register eins enthält zum Beispiel, wie in Tabelle 4.9 dargestellt, den Konfigurationsvektor für Kurzschlüsse gegen Masse. Ist in diesem Konfigurationsvektor

Tabelle 4.9: Register des AXI-Interfaces des Blocks zur Fehlereinspeisung

Register	**Bit(s)**	**R/W**	**Parameter**	**Funktion**
0	31	R	Invalid	Ungültige Kombination ausgewählt
0	3	R/W	Bus komb.	Kurzschlussbus 1 und 2 kombiniert
0	2	R/W	Update	Aktualisierung der Ausgänge
0	1	R/W	Reset	Reset des Kurzschlussblocks
1	9-0	R/W	Short GND	Kurzschluss gegen Masse (Kanal 1-10)
2	9-0	R/W	Short VBAT	Kurzschluss gegen VBAT (Kanal 1-10)
3	9-0	R/W	Short BUS1	Kurzschlussbus 1 (Kanal 1-10)
4	9-0	R/W	Short BUS2	Kurzschlussbus 2 (Kanal 1-10)

*Alle nicht explizit aufgeführten Bits der Register sind unbenutzt

das x-te Bit gesetzt, wird der Kanal x gegen Masse kurzgeschlossen. Da das System sich auf zehn Kanäle beschränkt, sind nur die untersten zehn Bit eines Registers relevant und werden für den Konfigurationsvektor herangezogen.
Durch Setzten des Reset-Bits können sämtliche Ausgänge des Blocks auf null gesetzt und damit alle Kurzschlüsse des Testsystems aufgehoben werden. So lange das Reset-Bit gesetzt ist, wird das Update-Bit ignoriert und die Ausgänge bleiben null.
Setzt die Software das Update-Bit, werden die gegenwärtig in den Registern des AXI-Interfaces vorhandenen relevanten Bits in die Konfigurationsvektoren des Blocks übernommen. Dies kann ebenso durch den externen Timer-Block ausgelöst werden, welcher mit dem Update-Eingang dieses Blocks verbunden ist. Die Vorgaben für die Kurzschlüsse gegen Masse und die Verbindungen zu dem Kurschlussbus 1 und 2 der einzelnen Kanäle werden ohne Modifikationen übernommen und an die Ausgangszellen des FPGAs weitergeleitet. Um sicherzustellen, dass es auch bei fehlerhafter Konfiguration nicht zu einem Kurzschluss zwischen Masse und der positiven Versorgungsspannung kommt, prüft der Block, welche Kanäle er gegen die positive Versorgungsspannung schalten kann. Dies bedeutet, dass der Kurzschluss eines Kanals gegen die Masse den Vorrang vor einem Kurzschluss gegen die Versorgungsspannung VBAT hat.
Die Bestimmung der gültigen Kombinationen erfolgt durch kombinatorische Logik innerhalb eines Taktes. Zunächst wird geprüft, ob der erste Kurzschlussbus eine Verbindung zur Masse besitzt. Dazu wird der Vektor mit den Kurzschlüssen gegen Masse mit dem Vektor der Verbindungen der Kanäle mit dem Kurzschlussbus 1 bitweise Und-verknüpft. Entsteht in dem Vektor eine eins, so ist der entsprechende Kanal sowohl mit der Masse, als auch dem Kurzschlussbus 1 verbunden. Der gesamte Kurzschlussbus 1 besitzt dann Massepotential.
Die gleiche Logik wird ebenfalls mit allen Kanälen und dem Kurzschlussbus 2 durchge-

führt. Der Block der Fehlereinspeisung besitzt nun Informationen darüber, ob die Kurzschlussbusse Massepotential aufweisen. Über das Bit Bus komb. kann der Anwender dem Block mitteilen, dass die beiden Kurzschlussbusse über eine externe Leitung oder Jumper miteinander verbunden sind. Ist dieses Bit gesetzt und besitzt einer der Kurzschlussbusse Massepotential, so geht der Block davon aus, dass auch der zweite Kurzschlussbus mit Masse verbunden ist.
Anhand dieser Informationen bestimmt der Block, welche der Kanäle er gegen die positive Versorgung schalten darf, ohne einen Kurzschluss zwischen den beiden Versorgungsleitungen zu erzeugen. Ausgeschlossen sind dabei Kanäle, die bereits direkt über den N-Kanal-MOSFET der Halbbrücke eine Verbindung zu Masse haben und Kanäle, die mit einem Kurzschlussbus, der Massepotential besitzt, verbunden sind. Kann der Block eine angeforderte Verbindung gegen die positive Versorgungsspannung nicht herstellen, um einen Kurzschluss zu vermeiden, so setzt er das Invalid-Bit. Dieses teilt der Software mit, dass sie eine ungültige Kombination an Kurzschlüssen dem Block übergeben hat und die tatsächlichen Verbindungen bzw. Kurzschlüsse nicht den übergebenen Werten entsprechen.

4.5 32 Bit Timer Block

Die programmierbare Logik verfügt über einen 32 *Bit* Timer, welcher genutzt wird, um Aktionen von anderen Blöcken auszulösen. Der Auslösezeitpunkt kann durch den Hardware-Timer deutlich genauer eingehalten werden, als wenn die Aktionen durch einen Befehl der Software gestartet werden. Die Software, insbesondere die Applikationen, die unter dem nicht echtzeitfähigen Linux laufen, besitzt undefinierte Latenzen und auch der AXI-Bus des Systems kann zeitweilig blockiert sein. Dies macht die Analyse zeitkritischer Parameter durch einen rein softwarebasierten Ablauf unmöglich.
Als Alternative kann daher dieser Timer genutzt werden. Zunächst müssen die anderen Blöcke mit den Daten der gewünschten Aktionen über deren AXI-Interface konfiguriert, jedoch nicht gestartet, werden. Dazu reicht es aus, das entsprechende Update- oder Enable-Bit des entsprechenden Blocks nicht zu setzen. In einem zweiten Schritt wird dieser Timer mit den Zeitpunkten, an welchen die Aktionen ausgeführt werden sollen, konfiguriert und der Timer gestartet. Erreicht der Timer den entsprechenden Zeitpunkt bzw. Vergleichswert, startet er die Aktion eines Blocks indem er dessen externes Eingangssignal setzt. Die Zeiten zwischen den einzelnen Aktionen werden so unabhängig von den Latenzen der Software eingehalten. Die internen Verzögerungen der einzelnen Blöcke bleiben jedoch bestehen. Diese machen im Vergleich zur undefinierten Latenz der Software nur einen sehr geringen Teil aus.
Der Timer kann Aktionen in den folgenden Blöcken auslösen: Dem PSI-Patterngenerator, der Ansteuerung der DACs der Sensorsimulatoren und der Fehlereinspeisung. Im PSI-

Patterngenerator kann der Timer, je nach internen Einstellungen des Blocks, die Erzeugung eines neuen Frames starten oder die asynchrone Übertragung eines bereits erzeugten Frames anstoßen. Die Fehlereinspeisung nutzt das Signal des Timers, um festzulegen, wann die Ausgänge den neuen Wert annehmen sollen bzw. ein Kurzschluss entstehen oder aufgehoben werden soll. Der DAC-Block startet die Aktualisierung der Ströme aller Sensorsimulatoren, wenn er ein Signal vom Timer erhält. Dabei ist zu beachten, dass weiterhin die Laufzeit durch die SPI-Übertragungen zu den DACs entsteht und dementsprechend der Vergleichszeitpunkt des Timers früher gewählt werden muss.
Neben dem Ansprechen dieser internen Blöcke stellt der Timer auch noch zwei Auslösesignale bereit. Das erste dieser beiden ist ein Signal, was auf den externen Trigger-Ausgang gegeben werden kann. So können die Messungen von externen Instrumenten, wie z.B. einem Oszilloskop, durch die Timervorgaben mit dem zu untersuchenden Event synchronisiert werden. Siehe dazu auch den Abschnitt 4.6 über den externen Trigger-Ausgang. Das zweite Signal stellt einen Sync-Ausgang über GPOs des FPGAs für den PSI5-Bus bereit. Dieser kann, sofern das DUT über einen Sync-Eingang verfügt, mit dem DUT verbunden werden, sodass dieses zu genau definierten Zeitpunkten die PSI5-Syncimpulse erzeugt bzw. die Datenkommunikation auf dem PSI5-Bus stattfindet.
Der Timerblock besteht intern aus einem 32 Bit Zähler, dessen Zählstand bei jedem Wert mit den Vergleichswerten, die dem Block übergeben werden, verglichen wird. Entspricht ein Vergleichswert genau dem Zählerwert, so wird der entsprechende Signalausgang des Timer-Blocks gesetzt. Der Timer verfügt über einen internen Taktteiler, um den ihn versorgenden Haupttakt, entsprechend der gewünschten Auflösung, teilen zu können. Die gegenwärtig im Testsystem genutzte Auflösung des Timers ist 1 μs. Der Timer kann mit seinen 32 Bit daher maximal rund 72 Minuten lang zählen.
Das AXI-Interface des Timer-Blocks, welches in Tabelle 4.10 dargestellt ist, besteht aus acht Registern, von welchen das Register 0 das Kontrollregister ist. Mittels des Reset-Bits

Tabelle 4.10: Register des AXI-Interfaces des Timer-Blocks

Register	**Bit(s)**	**R/W**	**Parameter**	**Funktion**
0	31	R	OVFL	Überlauf des Timers
0	2	R/W	Preload	Vorladen des Timers aktivieren
0	1	R/W	Reset	Zurücksetzen des Timers
0	0	R/W	Enable	Timer aktivieren
1	31-0	R	Timer Wert	Aktueller Timer Zählerstand
2	31-0	R/W	Preload Wert	Timer Vorladewert
3	31-0	R/W	Trigger compare	Trigger Ausgang Zeitpunkt
4	31-0	R/W	PSI5 PG compare	PSI5 Start Vergleichszeitpunkt
5	31-0	R/W	PSI5 DAC compare	DAC Vergleichszeitpunkt
6	31-0	R/W	Short compare	Fehlereinspeisung Zeitpunkt
7	31-0	R/W	Sync compare	Sync Ausgang Vergleichszeitpunkt

*Alle nicht explizit aufgeführten Bits der Register sind unbenutzt

kann der Timer zurückgesetzt werden. Ist das Reset-Bit aktiv, sind ebenfalls alle Ausgangssignale des Timer-Blocks deaktiviert, auch wenn der entsprechende Vergleichswert ebenfalls null ist. Während das Reset-Bit aktiv ist, ist der Timer inaktiv, sodass er auch bei gesetztem Enable- oder Preload-Bit nicht hochzählt bzw. vorgeladen wird.
Das Starten oder Stoppen des Timers kann die Software durch das Setzen oder Löschen des Enable-Bits im Register 0 durchführen. Der Timer ist bei gesetztem Enable-Bit aktiv und zählt hoch. Wird das Bit gelöscht, so bleibt der Timer beim aktuellen Zählerstand stehen und zählt von diesem weiter, sollte er ohne zwischenzeitliches Reset wieder aktiviert werden.
Der Zähler im Timer kann mit einem Startwert vorinitialisiert werden. Dazu muss in Register 2 des AXI-Interfaces der gewünschte Startwert abgelegt und das Preload-Bit im Kontrollregister gesetzt werden. So lange das Preload-Bit gesetzt ist, zählt der Timer nicht und auch die Ausgänge des Timer-Blocks sind alle inaktiv. Entspricht ein Vergleichswert genau dem Startwert zur Vorinitialisierung, so wird der zugehörige Ausgang erst mit dem ersten Takt, im dem der Zähler aktiv ist, gesetzt. Erreicht der Zähler seinen Maximalwert von $2^{32} - 1$, so stoppt dieser, um kein durch einen Überlauf entstehendes, undefiniertes Verhalten zu erzeugen. Zusätzlich wird das OVFL-Bit im Kontrollregister gesetzt, sodass die Software erkennen kann, dass der Maximalwert erreicht wurde, ohne dass diese das Register 1 mit dem gegenwärtigen Zählerwert lesen und mit dem Maximalwert vergleichen muss.
Über die Register 3-7 werden jeweils die Vergleichszählerwerte für die Erzeugung der entsprechenden Steuerimpulse bereitgestellt. Erreicht der Timer einen der jeweiligen Vergleichswerte, wird das zugehörige Steuersignal so lange aktiviert, wie der Vergleichswert und der aktuelle Zählerstand übereinstimmen. Die Dauer, die das Signal aktiviert ist, kann, je nach der bei der Synthese gewählten Auflösung des Timers, zu sehr kurzen Impulsen führen. Für die meisten Signale ist dies kein Problem, da diese von anderen internen Blöcken des FPGAs verwaltet werden. Das Signal, welches über die GPOs des FPGAs mit dem DUT verbunden ist, muss jedoch eine ausreichend lange Impulsdauer aufweisen, damit dieses von dem DUT zuverlässig erkannt wird.
Dazu verfügt der Timer-Block über einen zweiten Zähler, der gestartet wird, wenn der Sync-Ausgang des Timer-Blocks aktiviert werden soll. Solange dieser Zähler runterzählt, bleibt der Sync-Ausgang eingeschaltet. Durch eine geeignete Vorinitialisierung dieses zweiten Zählers in Abhängigkeit der Taktfrequenz des Blockes, kann eine ausreichende Länge des Steuerimpulses für das DUT sichergestellt werden. Der Vorinitialisierungswert des Zählers für den Sync-Ausgang ist fest in dem Block konfiguriert und kann nur zum Zeitpunkt der Synthese angepasst werden.

4.6 Triggerausgang

Das Testsystem besitzt einen Ausgang mit BNC-Stecker, welcher für die Synchronisation mit anderem Equipment vorgesehen ist. Damit dieser möglichst nützlich ist und für die unterschiedlichsten Untersuchungen oder Aufbauten genutzt werden kann, ist der Ausgang nicht statisch mit einer Funktion oder Signal des Testsystems verbunden. Stattdessen ist der Triggerausgang mit einem Block verbunden, über welchen sich unter anderem, ähnlich einem Multiplexer, die Signalquelle des Triggerausgangs einstellen lässt.
Die Software kann über das Setzen des Input select-Bits im Register 0 des in Tabelle 4.11 dargestellten AXI-Interfaces des Triggerausgangs zwischen zwei Signalquellen wählen. Das

Tabelle 4.11: Register des AXI-Interfaces des Triggerausgangs

Register	Bit(s)	R/W	Parameter	Funktion
0	31-16	R/W	Pulse duration	Impulslänge des Ausgangssignals
0	4	R/W	Mode	Typ des Ausgangssignals
0	2	R/W	Input select	Auswahl der Signalquelle
0	1	R/W	Reset	Zurücksetzen des Blocks
0	0	R/W	Enable	Ausgangssignal aktivieren
1	9-0	R/W	Ch. enabled	Selektion der Signalquellen Kanäle

*Alle nicht explizit aufgeführten Bits der Register sind unbenutzt

erste ist der Timerblock, welcher zu einem definierten Zeitpunkt ein Signal bereitstellt. Das Signal des Timers kann im Gegensatz zur zweiten Signalquelle unabhängig vom Verhalten des zu untersuchenden PSI5-Busses verwendet werden. Die zweite Quelle sind die Signale, welche von der Detektorhardware des Testsystems für die Syncimpulse des PSI5-Busses erzeugt werden. Diese bestehen aus einem Impuls, welcher so lange andauert, wie der Syncimpuls des PSI5-Busses aktiv ist. Das Bit neben dem Input select-Bit wird gegenwärtig nicht verwendet, um die Auswahl der Signalquelle in Zukunft um ein Bit erweitern zu können, sollten zusätzliche Quellen in das Design integriert werden.
Da das Testsystem nur über einen Triggerausgang verfügt, aber zehn Quellen in Form von Syncimpulsen des PSI5-Busses besitzt, kann über das Register 1 ausgewählt werden, welche Kanäle für die Erzeugung eines Triggersignals herangezogen werden sollen. Die im Register 1 definierte Maske wird dazu mit einem Vektor der Signale der einzelnen Kanäle bitweise Und-verknüpft. Der entstehende Vektor der aktiv auszuwertenden Signale wird durch ein Oder auf ein Signal reduziert. Sind mehrere PSI5-Kanäle als Quelle aktiviert, kann daher nicht bestimmt werden, welcher der Kanäle nun das Ausgangssignal erzeugt hat.
Der Trigger-Output-Block verfügt über zwei Betriebsmodi, welche die Art des Ausgangssignals bestimmen. Im ersten Modus entspricht das Ausgangssignal, welches am BNC-Stecker anliegt, genau dem Signal, das der Block auch erhält. Dieses wird entsprechend der Auswahl der Signalquelle ähnlich einem Multiplexer weitergeleitet. Um diesen Modus

zu aktivieren, muss das Mode-Bit in Register 0 mit null belegt sein. Dieser Modus empfiehlt sich nur, wenn die Signalquelle der Syncimpuls eines oder mehrerer PSI5-Kanäle ist. In diesem Fall entspricht das Ausgangssignal dem Signal, welches von der Oder-Verknüpfung der Signale der Komparatorschaltung für den Syncimpuls PSI5-Bus erzeugt wird. Ist jedoch als Signalquelle der Timerblock gewählt, so entspricht das Signal am Ausgang dem Steuersignal des Timerblocks. Dessen Dauer ist je nach Frequenz des Hauptaktes so kurz, dass es nicht als Ausgangssignal geeignet ist.
Der zweite Betriebsmodi des Timerblocks ist ein Ausgangssignal mit einer einstellbaren Impulslänge. Die Länge des Impulses kann dabei über die obersten 16 *Bit* des Register 0 eingestellt werden. Passiert ein Event, so wird von dem Block ein Zähler gestartet und der Ausgang so lange eingeschaltet, bis der Zähler den im Register für die Pulse Länge definierten Wert erreicht hat. Als Startsignal für den Zähler kann das Steuersignal des Timerblocks direkt verwendet werden, für den Syncimpuls des PSI5-Busses befindet sich noch eine Flankendetektion in der Signalkette. Diese sorgt dafür, dass nur steigende Flanken den Zähler starten und keine permanenten Signale. Andernfalls könnte ein einziges Signal der Komparatorschaltung, welches länger ist als die definierte Impulslänge des Ausgangssignals, mehrere Ausgangsimpulse erzeugen.
Tritt im Modus mit einer festen Impulslänge als Ausgangssignals ein weiteres Event auf, während der Ausgangsimpuls des ersten Events noch nicht vorbei ist, so wird dieses neue Event ignoriert. Erst wenn der Ausgangsimpuls abgeschlossen ist, kann eine neues Event einen erneuten Ausgangsimpuls auslösen. Ist die Ausgangsimpulslänge entsprechend lang gewählt und finden die Events, wie die Sync-Impulse der ausgewählten PSI5-Busse, kurz hintereinander statt, kann es zu einer Maskierung mancher Events kommen.
Das Enable-Bit des Registers 0 ist ein globales Enable und wirkt unabhängig von der Auswahl in Register 1. Ist das Enable-Bit nicht gesetzt, so ist der Ausgang deaktiviert bzw. permanent auf null gesetzt. Erst mit der Aktivierung dieses Bits durch die Software liegt am Ausgang beim Auftreten eines Events ein Signal an.

4.7 Konfigurierbarer Taktteiler

Das Testsystem verfügt über einen konfigurierbaren Taktteiler. Dieser wird genutzt, um das DUT mit einem Takt zu versorgen, wenn dieses über keinen internen Oszillator verfügt oder weitere Taktquellen benötigt. Der Taktteiler kann über das in Tabelle 4.12 dargestellte AXI-Interface in seiner Ausgangsfrequenz angepasst werden und besitzt zwei unabhängige Kanäle. Der Taktteiler teilt den ihn speisenden Haupttakt mittels jeweils einem 32 *Bit* Zähler für jeden der Kanäle. Der erste Zähler wird dabei so lange inkrementiert, bis für den ersten Taktausgang der in Register 0 gespeicherte zugehörige Ma-

Tabelle 4.12: Register des AXI-Interfaces des Taktteilers

Register	Bit(s)	R/W	Parameter	Funktion
0	31-0	R/W	Top Wert	Zähler 1 Maximalwert
1	31-0	R	Timer Wert	Aktueller Zählerstand Zähler 1
2	31-0	R/W	Top Wert	Zähler 2 Maximalwert
3	31-0	R	Timer Wert	Aktueller Zählerstand Zähler 2

*Alle nicht explizit aufgeführten Bits der Register sind unbenutzt

ximalwert erreicht ist. Dann wird das Ausgangssignal invertiert, sodass ein Taktsignal entsteht und der Zähler zurück auf null gesetzt wird. Selbiges gilt analog für den zweiten Zähler und den zweiten Ausgang, welche vollständig unabhängig von den ersten beiden sind. Da nur beim Erreichen des Maximalwertes das Ausgangssignal invertiert wird und intern ein Taktzyklus für das Vergleichen und Rücksetzen benötigt wird, ist der geringste Teilfaktor des Taktteilers zwei. Dies bedeutet, dass wenn der Taktteiler mit einem Takt von $100\,MHz$ über den AXI-Bus gespeist wird und als Maximalwert (x) null angegeben ist, die Ausgangsfrequenz des Teilers gemäß $f_{out} = \frac{f_{in}}{2+x}$ genau $50\,MHz$ beträgt. Die tatsächlich mögliche Ausgangsfrequenz ist dabei jedoch auch von den physikalischen Begrenzungen der Ausgangszellen des FPGAs abhängig.

Erfolgt über das AXI-Interface ein schreibender Zugriff auf das Register 0 oder das Register 2, welche die jeweiligen Maximalwerte der Zähler beinhalten, so wird der entsprechende Zähler auf null zurückgesetzt. Dies verhindert, dass der Ablauf der alten Taktperiode abgewartet werden muss, bis der neue Wert genutzt wird. Dies stellt typischerweise kein Problem dar. Kommt es jedoch durch die Software zu einer fehlerhaften Konfiguration oder ist bewusst eine sehr lange Periodendauer gewählt, so kann es durch die Größe des Zählers von $32\,Bit$ zu einer Verzögerung bis in den Minutenbereich kommen, bis der neue Wert genutzt wird.

Über das Register 1 und Register 3 können die gegenwärtigen Werte der Zähler von der Software ausgelesen werden. Ein Anhalten des Taktteilers bzw. Deaktivieren des Ausgangs über das AXI-Interface ist vorgesehen. Wird jedoch das Resetsignal des AXI-Busses gesetzt, so stellt auch der Taktteiler seinen Betrieb ein, da zu diesem Zeitpunkt die Gültigkeit des den Block speisenden Takts nicht gewährleistet ist. Die beiden Ausgänge sind dann permanent auf Null gesetzt und die beiden Zähler werden zurückgesetzt.

5 Software

Die Software des Testsystems besteht aus zwei unterschiedlichen Betriebssystemen, welche jeweils einen CPU-Kern des ZYNQs belegen und unterschiedliche Softwareapplikationen für das Testsystem ausführen. Auf dem ersten CPU-Kern CPU0 wird durch eine mehrstufige Bootloader-Architektur ein Linux-Kernel geladen, welcher die Basis für die Ablaufsteuerung und die Software zur Interaktion mit dem Anwender bildet. Der zweite CPU-Kern CPU0 wird bei Bedarf durch den Linux-Kernel, nach einer Aufforderung durch die Anwendungsapplikation, gestartet und führt das Echtzeitbetriebssystem FreeRTOS zur Kommunikation und Konfiguration des DUTs aus. Beide Betriebssysteme und deren Applikationen interagieren zum einen miteinander und zum anderen mit den IP-Blöcken der programmierbaren Logik des ZYNQs, welche in Kapitel 4 dargestellt sind. Während die IP-Blöcke vor allem komplexere parallele Hardwarefunktionen abbilden, dient die Software primär der Bedienung, Verwaltung des Testsystems und der Umsetzung der Testdurchläufe durch sequentielle Testschritte.
Die Grundbausteine der Softwarearchitektur dieses Testsystems basiert auf bereits bestehenden Architekturen, wie zum Beispiel dem Linux-Kernel oder den zugehörigen Bootloadern, welche an die Hardware und die Anforderungen des Testsystems angepasst wurden. Die Entwicklung sämtlicher eigener Softwarekomponenten, wie der Ablaufsteuerung oder Applikation, welche unter FreeRTOS läuft, erfolgte in der Programmiersprache C. Die einzige Ausnahme bildet das Webinterface zur Interaktion mit dem Anwender, dessen Design durch die Auszeichnungssprache HTML und Skripte beschrieben ist. In den folgenden Abschnitten sind die einzelnen Software-Komponente des Testsystems im Detail dargestellt.

5.1 FreeRTOS

Das Testsystem benötigt ein Echtzeitbetriebssystem unter anderem für die Ansteuerung der Watchdogs eines DUTs, sofern dieses einen oder mehrere Watchdogs besitzt. Diese Aufgabe könnte auch durch eine, für das zu untersuchende DUT entwickelte, Bare-Metal-Anwendung erfüllt werden. Da das Testsystem jedoch für eine große Breite an DUTs geeignet sein soll, erfolgt die Ansteuerung der Watchdogs auf Basis eines Echtzeitbetriebssystems. Dieses zeichnet sich durch die gesicherte Verarbeitung von Anfragen einer Anwendungsapplikation innerhalb eines definierten Zeitraums unabhängig vom Systemzustand aus. Es ermöglicht, durch die vorhandenen universalen Strukturen und Funktionen, eine einfachere Anpassung der Softwareroutinen an DUTs mit einem anderen Aufbau oder anderen Anforderungen.
Echtzeitbetriebssysteme werden typischerweise für Mikrocontroller mit geringer Rechen-

leistung und begrenztem Speicher eingesetzt. Im Falle des vorliegenden Testsystems ist dies jedoch nicht gegeben, da der CPU-Kern, welcher das Echtzeitbetriebssystem ausführt, ein leistungsstarker ARM Cortex A9 mit einer Taktrate von mehreren hundert Megahertz ist, allerdings sind bei der Ausführung der Software dieses Testsystems strenge Zeitvorgaben einzuhalten. FreeRTOS unterstützt mehrere simultane Tasks, dessen Äquivalent in einem nicht-echtzeitfähigen Betriebssystem ein Thread wäre. Ein weiterer Bestandteil sind Mutexe und Semaphoren zur Synchronisierung und Ressourcenmanagement, sowie Software-Timer. Der Taskhandler kann in zwei Modi arbeiten. Der erste arbeitet ohne Ticks und ist vor allem für batteriebetriebene Systeme gedacht, damit das System nicht ständig aufgeweckt wird. Das Testsystem nutzt jedoch den zweiten Modus des Task-Handlers, welcher durch einen Timer-Interrupt ausgelöste Ticks nutzt, da das Testsystem keine solchen Beschränkungen im Bezug auf die Energieaufnahme besitzt.
FreeRTOS wird als Echtzeitbetriebssystem des Testsystems gewählt, da es über einen Task-Handler verfügt, welcher die Zuweisung von Prioritäten an Tasks unterstützt, sodass bestimmt werden kann, welche Funktionen den Vorrang erhalten. Dies ist für dieses Testsystem von großer Bedeutung, da so eine einfache Priorisierung der Tasks, welche den Watchdog ansteuern, vor anderen Aufgaben vorgenommen werden kann. Über die Mutexe kann einem Task außerdem der exklusive Zugriff auf einen IP-Block zugewiesen werden, sodass es zu keinen Kollisionen kommt. Weitere Kriterien für die Auswahl von FreeRTOS sind die vorhandene Integration in die Toolchain zur Softwareentwicklung von Xilinx, eine gute Dokumentation des Betriebssystems und die MIT-Lizenz, unter welcher FreeRTOS veröffentlicht ist. Diese erlaubt die lizenzfreie Einbindung in Closed-Source Projekte.
Der Tick-basierte Task-Handler funktioniert mit einem Timer, welche eine Tick-ISR-Routine auslöst, in welcher zunächst der Tick-Zähler erhöht wird. Dieser Zähler bildet die Grundlage von FreeRTOS für Funktion wie Warten und Timeouts. Daher hängt die zeitliche Auflösung des Betriebssystems direkt mit dem Intervall zusammen, mit welchem die Tick-Interrupt Service Routine (ISR) aufgerufen wird. Soll zum Beispiel ein Task für $5\,ms$ warten, so muss die angegebene Zeit erst über ein Makro in Ticks umgerechnet werden. Zeiten unterhalb von einem Tick bzw. einem Timer-Intervall sind nicht möglich.[Rea]
Der Task-Handler wechselt nach einem Aufruf durch die Tick-ISR-Routine den Task nach einem gewichteten Round-Robin-Verfahren in Abhängigkeit von der Priorität der Tasks. Das Intervall mit welchem die Tick-ISR-Routine durch den Timer aufgerufen wird, beträgt in typischen Mikrocontroller Anwendungen in etwa $1\,\mathsf{ms}$, da bei einem häufigeren Aufrufen andernfalls zu viel Rechenzeit durch den Task-Handler verbraucht würde. In diesem Testsystem wurde jedoch ein Intervall von $10\,\mu s$ gewählt, um die benötigte Auflösung für die Einhaltung der Zeitfenster der Watchdogs des DUTs zu erreichen. Dies ist Aufgrund der hohen Rechenleistung der eingesetzten CPU problemlos möglich.
Es wird die Version 10 von FreeRTOS in einer von Xilinx modifizierten Variante, welche auf die Hardware der ZYNQ-Architektur abgestimmt wurde, verwendet.

5.2 Linux

Das eingesetzte Linux besteht aus dem von Xilinx für die ZYNQ-SoC-Reihe angepassten Petalinux-Kernel und einem Rootdateisystem der Distribution Debian. Der Kernel wird mit der Toolchain der Petalinux-Tools in der Version 2018.2 erzeugt und besitzt aufgrund der festen Bindung zwischen Petalinux-Tools und den mitgelieferten Kernelquellen die Version 4.14. Das erzeugte Kernel-Image wird in der Bootpartition des eMMC-Speichers, welcher sich auf dem Trenz TE0720 Modul befindet, abgelegt. Die zweite und letzte Partition des Speichers enthält das Rootdateisystem. Eine separate Partition für die Home-Verzeichnisse der User existiert nicht, stattdessen sind diese Bestandteil des Rootdateisystems.
Damit die Ausgaben des Kernels während des Bootvorgangs über den USB-Serial-Adapter der Trägerplatine betrachtet werden können, wird der Kernel so konfiguriert, dass er die uart0-Schnittstelle des ZYNQs für die Ausgaben nutzt. Das Trenz TE0720 Modul ist mit der ISL12022 Echtzeituhr von Renesas ausgerüstet, welche vom Linux-Kernel zur Bestimmung der Systemzeit genutzt wird und über I2C mit dem ZYNQ verbunden ist. Damit der Kernel die externe Echtzeituhr unterstützt, muss bei dessen Kompilierung die folgende Option gesetzt werden: CONFIG_RTC_DRV_ISL12022. Des weiteren werden für die Echtzeituhr und den CPLD-Systemcontroller der TE0703 Trägerplatine die i2c-tools benötigt, sodass diese im Root-Dateisystem vorhanden sein müssen.
Damit Treiber vom Kernel zu einem späteren Zeitpunkt nachgeladen werden können, wird dieser mit der Option loadable module support gebaut. Die als Modul nachgeladenen Treiber sind in diesem Testsystem vor allem die Treiber, welche zur Kommunikation und Verwaltung des zweiten CPU-Kerns und dessen Betriebssystem, sowie der programmierbaren Logik eingesetzt werden. Beispielsweise wird das UIO-Treiber Modul nachgeladen, welches durch die Option Userspace I/O platform driver with generic IRQ handling aktiviert wird und als Treiber für die Kommunikation mit den IP-Blöcken der programmierbaren Logik verwendet wird (vgl. Abschnitt 5.6.2). Um den zweiten CPU-Kern aus dem Linux zu verwalten und ihn zu den passenden Zeitpunkten starten und anhalten zu können, wird das remoteproc framework verwendet, welches eine virtuelle Repräsentation der Hardware und das Virtualisierungs-Framework virtio bereitstellt auf dem der Kommunikations-Bus rpmsg zwischen dem Linux-Kernel und der FreeRTOS-Instanz basiert. Die Option zur Aktivierung dieser Funktionen als Kernel-Modul ist abhängig von der eingesetzten Hardware und der verwendeten Umgebung zur Erzeugung des Linux-Kernels, da die Hersteller der SoCs oftmals Modifikationen oder Erweiterungen vornehmen, welche sich nicht in den normalen Kernelquellen finden. Die Option wird von Xilinx für die ZYNQ-7000 SoC-Reihe mit Support ZYNQ remoteproc bezeichnet und befindet sich unter den Remoteproc drivers. Als letzte Option wird der Userspace firmware loading support aktiviert, welcher benötigt wird, damit die Binärdatei, welche das Image der FreeRTOS-Instanz enthält, im Root-

dateisystem abgelegt wird und Anwendungen aus dem Userspace die auszuführende Datei vor dem Start des CPU-Kerns auswählen können.
Da einige dieser Treiber als Module in den Kernel nachgeladen werden, sind diese nicht im Kernel-Image enthalten, sondern im Modulverzeichnis des Rootdateisystems. Weil jedoch nicht das von der Petalinux-Toolchain erzeugte Rootdateisystem in diesem Testsystem verwendet wird, sondern sich ein headless Debian auf dem eMMC-Speicher des Testsystems befindet, werden diese Module in das verwendete Rootdateisystem kopiert. Für alle weiteren benötigten Anwendungen verfügt Debian über eine Paketverwaltung, welche es ermöglicht Abhängigkeiten und Anwendungen, welche nicht in dem Installationsimages des Rootdateisystems enthalten sind, nachzuinstallieren. Der Bezug von Paketen für die Installation kann dabei direkt als Download aus den Paketquellen erfolgen, wenn das System über einen Internetzugriff verfügt. Ansonsten ist es möglich Pakete offline z.B. über USB auf das System zu übertragen und dort zu installieren. Die zusätzlich auf dem Debian des Testsystems installierten Pakete umfassen die json-c-Bibliothek für das Lesen von Dateien im JavaScript Object Notation (JSON)-Format (vgl. Abschnitt 5.8), einem Apache Webserver und dem PHP Hypertext Prozessor, welche beide für die Erzeugung des Webinterfaces benötigt werden.

5.3 Architektur

Die Abbildung 5.1 zeigt in einem groben Überblick wie die Architektur der Software in Verbindung mit dem Gesamtsystem steht. Die beiden ARM A9-CPU-Kerne des eingesetzten ZYNQs führen getrennte Betriebssystem aus, welche über einen Remote Processor Messaging Framework (rpmsg)-Bus miteinander Kommunizieren. Ebenfalls exemplarisch dargestellt ist das DUT mit seinen beiden SPI-Schnittstellen, von welchen eine zur Konfiguration des DUTs eingesetzte wird und über die zweite die von dem DUT empfangenen PSI5-Nachrichten ausgelesen werden können. Beide CPU-Kerne teilen sich den

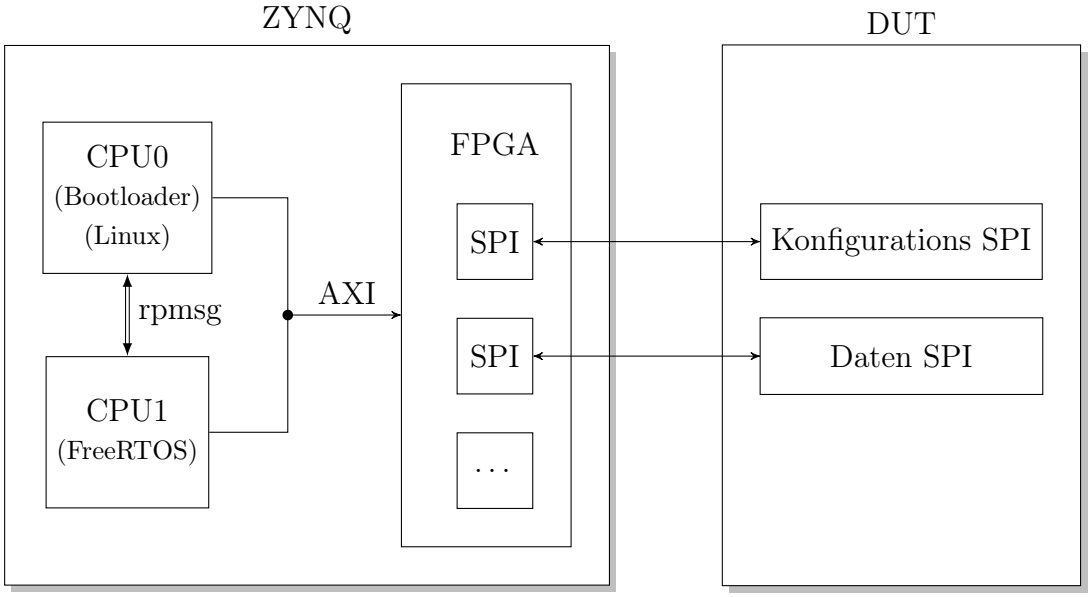


Abb. 5.1: CPU-Architektur und Schnittstellen

selben AXI-Bus und haben jeweils auch einen Zugriff auf die Interfaces aller IP-Blöcke der programmierbaren Logik. Damit es jedoch nicht zu inkonsistenten Konfigurationen der Blöcke kommt, ist es in der Architektur der Software des Testsystems vorgesehen, dass die FreeRTOS-Instanz nur Zugriffe auf die mit dem DUT verbundenen GPIO-Blöcke und die Konfigurations-SPI-Schnittstelle ausführt. Die Ablaufsteuerung unter Linux bedient alle anderen Blöcke, sodass eine strengen Teilung der Zugriffsrechte entsteht. Möchte die Ablaufsteuerung eine Einstellung des DUTs über die Konfigurations-SPI vornehmen, so muss sie, da die Verwaltung der Konfigurations-SPI von der FreeRTOS-Instanz durchgeführt wird, eine rpmsg-Nachricht an die FreeRTOS-Instanz mit den Daten senden. Die FreeRTOS-Instanz leitet die erhaltenen Konfigurationdaten dann zu einem geeigneten Zeitpunkt über die Konfigurations-SPI an das DUT weiter. Ein Zugriff auf die Nutzdaten-SPI kann von der Anwendung der Ablaufsteuerung direkt über einen Treiber erfolgen.
Da die beide CPU-Kerne sich den selben RAM als Hauptspeicher teilen, aber unterschiedliche Betriebssysteme ausführen, welche nicht wissen, wie das jeweils andere Betriebssystem den Speicher verwaltet, besitzen beide getrennte Speicherbereiche. In diesen

Speicherbereichen arbeitete jeweils ein Betriebssystem exklusiv. Die einzige Ausnahme zu dieser Teilung stellt der Bootprozess dar, währenddessen die erste CPU den Speicher der zweiten mit dem von der zweiten CPU auszuführenden Image vorlädt. Die Trennung zwischen den Betriebssystemen betrifft aber nicht nur den exklusiven Zugriff auf die AXI-Interfaces der IP-Blöcke und den Hauptspeicher, sondern auch weitere Ressourcen. Die FreeRTOS-Instanz wird z.B. mit der Option DUSE_AMP=1 gebaut, um zu verhindern, dass die FreeRTOS-Instanz versucht Ressourcen zu initialisieren, welche vom Linux-Kernel verwaltet werden.
Die Binärdateien beziehungsweise Images einzelnen Softwarebestandteile befinden sich in unterschiedlichen Speichern des System. Der FSBL befindet sich zusammen mit dem zugehörigen Bitstream des FPGAs im QSPI-Flash. Ebenfalls im QSPI-Flash abgelegt ist der Second Stage Bootloader, dessen Implementierung in diesem System U-Boot übernimmt. Das Image des Linux-Kernels wird in getrennten Partitionen mit dem Rootdateisystem im eMMC gespeichert, der im Gegensatz zum QSPI-Flash über einen bedeutend größeren Speicher verfügt und einen höheren möglichen Datendurchsatz besitzt, jedoch eine komplexere Ansteuerung erfordert. Alle bisher genannten Softwarekomponenten werden von dem ersten CPU-Kern cpu0 ausgeführt. Das Image der FreeRTOS-Instanz des zweiten CPU-Kerns cpu1 wird nicht in einem separaten Speicher abgelegt auf welchen die cpu1 einen direkten Zugriff hat, sondern als Datei im Rootdateisystem, das sich in der zweiten Partition des eMMCs befindet. Damit die cpu1 Zugriff auf das auszuführende FreeRTOS-Image hat, wird dies vor dem Start der cpu1 durch das Linux bzw. die cpu0 in den der cpu1 zugewiesenen Bereich des Random-Access Memory (RAMs) geladen. Der

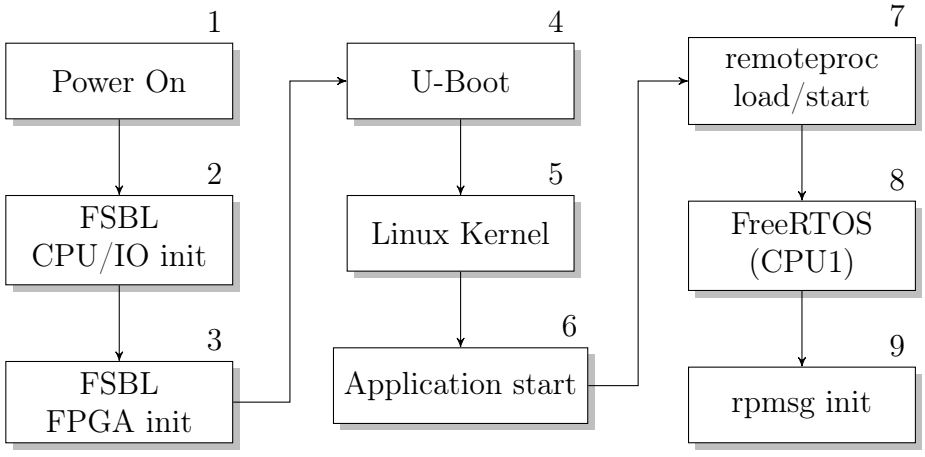


Abb. 5.2: Bootvorgang des Systems.

Bootvorgang des Testsystems aus Abbildung 5.2 beginnt mit dem Einschalten der Versorgungsspannung. Der ZYNQ-SoC ermittelt daraufhin anhand der Schalterstellungen der Trenz TE0703 Trägerplatine von welchem Speicher er das Bootimage (BOOT.BIN) laden soll. Mögliche Varianten sind dabei der QSPI-Flash und die SD-Karte, wobei erstere in der aktuellen Version des Testsystems genutzt wird. Das Bootimage der ZYNQ-Architektur

besteht aus einem First Stage Boot Loader und dem Bitstream für die programmierbare Logik. Wurde das Bootimage von dem internen Bootcode des SoC geladen, so wird von dem FSBL in einem ersten Schritt das Prozessorsystem des ZYNQs konfiguriert. Dazu zählen unter anderem die Taktgeneratoren, die Speicherschnittstellen und benötigten Peripheriecontroller. Ist dieser Schritt abgeschlossen, so beschreibt der FSBL die programmierbare Logik mit dem ebenfalls im Bootimage BOOT.BIN enthaltenen Bitstream. Die Tatsache, dass die programmierbare Logik mit dem Bitstream beschrieben wurde, lässt sich auch an der seitlichen grünen LED des Trenz TE0720 Modul erkennen, welche erlischt, wenn die Konfiguration der programmierbare Logik abgeschlossen ist. Der First Stage Boot Loader hat damit seine Aufgabe erfüllt und führt den als dritte Komponente im Bootimage enthaltenen Secondary Bootloader U-Boot aus.
U-Boot als zweite Bootloaderstufe erlaubt erstmalig den Eingriff von außerhalb in den Bootprozess. Durch die serielle Schnittstelle und das Konfigurationsmenü von U-Boot können die vom selbigen ausgeführten Schritte in einer Script-ähnlichen Beschreibung angepasst werden. U-Boot verfügt gegenüber dem FSBL über die Unterstützung von Dateisystemen, wodurch es möglich wird in U-Boot den Namen eines Kernel-Images in einem Dateisystem anstatt einer Speicheradresse anzugeben. U-Boot lädt das Kernel-Image aus der ersten Partition des eMMCs, welche mit dem FAT-Dateisystem formatiert ist, in das RAM und führt dieses aus. Ab diesem Zeitpunkt übernimmt der Linux-Kernel die Kontrolle über die von dem ersten CPU-Kern cpu0 auszuführenden Operationen.
Der nächste Schritt im Bootprozess des Systems betrifft die cpu1, welche bisher inaktiv war. Die cpu1 wird durch die Ablaufsteuerung, welche als Anwendung unter Linux läuft gestartet. Dazu wird zunächst durch das Remote Processor Framework das auszuführende Image in den Speicherbereich des RAMs geladen, auf welchem die cpu1 anschließend arbeiten soll. Ist das FreeRTOS-Image in den RAM geladen worden, wird durch einen zweiten remoteproc-Befehl der zweite CPU-Kern cpu1 gestartet und die FreeRTOS-Instanz ausgeführt. Damit beide nun aktiven Betriebssysteme miteinander Kommunizieren können, wird ein in Abschnitt 5.3.2 beschriebener rpmsg-Kanal zwischen beiden initialisiert.

5.3.1 Device Tree

Der Device Tree dient dem Kernel als Beschreibung der Hardware. Darunter ist in diesem Fall nicht nur die physikalische Hardware zu verstehen, sondern auch der Aufbau und die Verwendung der Speicher, sowie die in der programmierbaren Logik enthaltenen Funktionen. Anhand des Device Trees bestimmt der Kernel, welche Komponenten vorhanden sind und wie diese als Devices bezeichneten Systembestandteile zu verwenden sind. Dementsprechend werden sie vom Kernel Treiber geladen, das Speichermanagement angepasst oder andere Veränderungen vorgenommen.

Die typische Struktur eines Device Trees ist hierarchisch aufgebaut, sodass diese aus mehreren Dateien bestehen kann, welche sich auf die Beschreibung einzelner Hardwareaspekte aufteilen. Die Dateien eines Device Trees sind in der Lage, die Eigenschaften, welche von der unterliegenden eingebunden Datei definiert wurden, zu überschreiben. Xilinx liefert mit den Petalinux-Tools einen generischen Device Tree für die ZYNQ-Architektur aus, welcher bereits um die Besonderheiten der ZYNQ-Architektur gegenüber einem typischen Device Tree für ein auf einem ARM Cortex A9 basierenden System erweitert wurde. Der Device Tree des Testsystems basiert auf generischen Device Trees der ZYNQ-Architektur, wurde jedoch modifiziert, um die verwendeten IP-Blöcke und den Einsatz der zweiten CPU für eine FreeRTOS-Instanz widerzuspiegeln. Ebenso musste er angepasst werden, damit die Schnittstellen der Trenz Trägerplatine, wie Ethernet und USB verwendet werden können. Eine Liste mit den vollständigen Modifikationen und Erweiterungen des Device Trees befindet sich im Anhang A.2. Die Kernpunkte dieser Erweiterungen bzw. Anpassungen werden im weiteren Verlauf dieses Abschnitts erläutert.
Der Großteil der Modifikationen im Bezug auf die Speicherverwaltung des zweiten CPU-Kerns der Dual-Core-CPU des ZYNQs wurden in Übereinstimmung mit dem User Guide [Xil18b] von Xilinx getroffen. Über den in Auszug 5.1 des Device Trees dargestell-

```
4       reserved-memory {
5           ranges;
6           rproc_0_reserved: rproc@3e000000 {
7               no-map;
8               reg = <0x3e000000 0x01000000>;
9           };
10      };
```

Quellcode 5.1: Speicherreservierung

ten reserved-memory-Eintrag wird ein Bereich des DDR3-Hauptspeichers für die spätere Verwendung durch die FreeRTOS-Instanz reserviert. Der Parameter reg = <0x3e000000 0x01000000> gibt dabei die Adresse bzw. Position und die Größe des zu reservierenden Speicherbereichs an. Dieser Eintrag im Device Tree sagt nichts über die weiteren Verwendungszwecke des Speicherbereichs aus. Er schließt diesen Bereich nur aus der Speicherverwaltung des Linux-Kernels aus, sodass der Speicher nicht von den normalen Kernelroutinen belegt werden kann. Es wird ein Teil davon verwendet, um dort das auszuführende Executable and Linkable Format (ELF)-Image der FreeRTOS-Instanz abzulegen. Dieser separate Speicherbereich wird in dem Auszug 5.2 des Device Trees mit elf_ddr_0 bezeichnet, um deutlich zu machen, dass es sich um einen Speicherbereich im DDR-Hauptspeicher des Systems handelt, auch wenn dessen compatible-Attribut mit sram angeben ist. Dieses Attribut wird nur benötigt, damit dieser Speicherbereich im Hauptspeicher bei der Beschreibung des Remote Processor Framework-Devices als Speicher für das Firmware Image angegeben werden kann. Das Remote Processor Framework-Device wird dabei als Repräsentation des zweiten CPU-Kerns des ZYNQs genutzt. Das compatible-Attribut des

```
14      amba {
15          elf_ddr_0: ddr@0 {
16              compatible = "mmio-sram";
17              reg = <0x3e000000 0x400000>;
18          };
19      };
20
21      remoteproc0: remoteproc@0 {
22          compatible = "xlnx,zynq_remoteproc";
23          firmware = "firmware";
24          vring0 = <15>;
25          vring1 = <14>;
26          srams = <&elf_ddr_0>;
27      };
```

Quellcode 5.2: Remoteproc Definition

remoteproc-Devices spezifiziert die von Xilinx angepasste Version der Kernel-Module, welche zur Verwaltung der zweiten CPU geladen werden sollen. Über den Firmware-Eintrag kann der Pfad zu einem ELF-Image im Rootdateisystem des Linux angegeben werden, welches beim Systemstart in den dafür reservierten Speicherbereich geladen werden soll. Dieser ist im Device Tree des Testsystems jedoch nur mit einem Platzhalter versehen, welcher das automatische Laden eines Images beim Systemstart verhindert. Das eigentliche Image wird später über die Anwendung der Ablaufsteuerung und den Sysfs-Interface des remoteproc-Treibers ausgewählt. Die beiden vring-Einträge bestimmen, welche Interrupts vom Kernel ausgehend und in Richtung des Kernels genutzt werden sollen.
Für jeden IP-Block mit einem AXI-Interface, welches durch das Linux angesprochen werden soll, ist ein Eintrag im Device Tree vorhanden, welcher dem Kernel die Adresse, Typ und den passenden Treiber mitteilt. Da alle IP-Blöcke in diesem Testsystem den selben Interfacetyp besitzen und den selben Treiber nutzen, sind alle Device Tree Einträge mit dem exemplarisch dargestellten Eintrag 5.3 des Patterngenerators vergleichbar aufgebaut. Durch den Eintrag compatible = "generic-uio"; erhält der Kernel die Information, dass das Interface des IP-Blocks mit einem generischen UIO-Treiber kompatibel ist, woraufhin er diesen Treiber für das Device lädt. Die Referenz &psi5_pg_0 wird während der Kom-

```
141     &psi5_pg_0 {
142         compatible = "generic-uio";
143     };
```

Quellcode 5.3: UIO Treiberkompatibilität

,

pilierung des Device Trees durch die eigentliche Basisadresse des AXI-Interfaces ersetzt, welche sich im Adressbereich zwischen 4000_0000 und 7FFF_FFFF befindet, welcher dem AXI-Master zugewiesen ist und mit welchem die IP-Blöcke verbunden sind. Die Adressen werden bei der Kompilierung aus dem Hardware-Description-File des Designs der pro-

grammierbaren Logik extrahiert. Eine Übersicht über die Adressen der IP-Blöcke findet sich in Tabelle 4.1.
Der chosen-Eintrag im Device Tree bezieht sich auf kein echtes Device, sondern dient als Platzhalter für Parameter des Kernels wie die Bootargumente. Der Auszug 5.4 aus dem Device Tree des Testsystem zeigt die Bootargumente (bootargs) des Kernels. Das erste

```
28      chosen {
29          bootargs = "console=ttyPS0,115200 earlyprintk root=/dev/
                mmcblk1p2 rw rootwait uio_pdrv_genirq.of_id=generic-uio";
30          stdout-path = "serial0:115200ns";
31      };
```

Quellcode 5.4: Bootargumente im Device Tree

Argument definiert die Auswahl der seriellen Schnittstelle für die Konsole und deren Baudrate, während das zweite Argument earlyprintk die Ausgabe von Informationen über die serielle Schnittstelle vor der Initialisierung der eigentlichen Konsole aktiviert. Für Ausgaben im Bootprozess, welche bereits kurz nach der Übergabe von U-Boot an den Kernel stattfinden sollen, ist die Angabe in den Bootargumenten (bootargs) nicht ausreichend, sodass über den Parameter stdout-path des chosen-Eintrags ebenfalls die Definition der seriellen Schnittstelle und ihrer Baudrate erfolgt. Auch wenn sich die Syntax der beiden Definitionen unterscheidet, beziehen sie sich dennoch auf die selbe Schnittstelle mit einer identischen Baudrate. Die mittleren drei Bootargumente teilen dem Kernel mit, wo das Rootdateisystem zu finden ist und dass der Kernel dieses so mounten soll, dass Lese- und Schreibzugriffe möglich sind, sowie dass gegebenenfalls gewartet werden soll, bis das Rootdateisystem bereit ist. Es ist unter dem Device-Pfad /dev/mmcblk1p2 erreichbar, welches der zweiten Partition des eMMC des SoC-Moduls entspricht. Der letzte Parameter der Bootargumente (bootargs) ändert die internen Parameter des uio_pdrv_genirq Treibermoduls, sodass dieses auch mit den weiter oben beschriebenen generic-uio Einträgen des Device Trees kompatibel ist.

5.3.2 Remote Processor Messaging Framework

Die beiden unterschiedlichen Betriebssysteme des Testsystems, welche jeweils von einem CPU-Kern ausgeführt werden, benötigen eine Möglichkeit miteinander zu kommunizieren. Als Schnittstelle dieser Kommunikation wird rpmsg eingesetzt. Der rpmsg-Bus ermöglicht eine Kommunikation zwischen dem Linux-Kernel und externen Prozessoren über einzelne Kanäle. Er implementiert dazu einen Transport Layer als weitere Schicht auf das Virtualisierungs-Framework virtio, dessen virtuelle Hardware und Treiber den MAC Layer der Kommunikation zwischen beiden Prozessoren darstellt. Der eigentliche Austausch der Daten, welcher in einem Schichtenmodell dem physical Layer entsprechen wür-

de, erfolgt durch einen Speicher, auf welche beide Prozessoren Zugriff haben. Bei der ZYNQ-Architektur befindet sich der Speicher, welchen beide Prozessoren zum Austausch nutzen, in einem reservierten Bereich des DDR3-Hauptspeichers.[Che17]
Da der rpmsg-Bus nur eine Kommunikation mit dem Linux-Kernel, nicht jedoch mit den eigentlichen Anwendungen des Betriebssystems, ermöglicht, wird ein Kernel-Treiber benötigt, welcher ein Interface bereitstellt mit dem ein Zugriff auf den rpmsg-Bus aus dem Userspace erfolgen kann. Dafür wird ein Character-Treiber verwendet, der für jeden rpmsg-Kanal ein /dev/rpmsgX Interface erzeugt, über welches Anwendungen aus dem Userspace Zugriff auf den entsprechenden rpmsg-Kanal haben. Die rpmsg-Kanäle werden dabei durch den Treiber über ihren Namen identifiziert. Der Kanal, welchen das Testsystem nutzt, um mit der FreeRTOS-Instanz zu kommunizieren, ist mit rpmsg-freertos-tf-channel benannt. Der in diesem Testsystem eingesetzte Treiber rpmsg_user_dev_driver für den Zugriff aus dem Userspace wurde von Xilinx modifiziert, sodass dieser eine festgelegte Shutdown-Nachricht über den rpmsg-Kanal versendet, wenn das Filehandle einer Anwendung zu dem vom Treiber erzeugten /dev/rpmsgX Interface geschlossen wird. Empfängt die Gegenseite des rpmsg-Kanals diese spezifische Nachricht, kann sie dies als Anlass nehmen, sich selbst zu beenden.[Xil17, S. 52]
Die rpmsg-Nachrichten, welche über den Kanal zwischen Linux und der FreeRTOS-Instanz ausgetauscht werden, bestehen aus einem Struct, welches vier 32 *Bit* Variablen enthält. Das Struct ist in Abbildung 5.3 dargestellt und wird in beide Richtungen, von der Anwendung unter Linux oder in Gegenrichtung von der FreeRTOS-Instanz ausgehend, genutzt. Der Autor hat diese Nachrichtenstruktur gewählt, da ein Struct mit seiner festen Grö-

```
1  struct _rpmsg{
2      uint32_t status;
3      uint32_t command;
4      uint32_t address;
5      uint32_t value;
6  };
```

Abb. 5.3: rpmsg Struct

ße die Übermittlung einfacher gestaltet. Durch diese feste Struktur muss kein Feld mit einer Größenangabe übertragen und ausgewertet werden, um prüfen zu können, ob die Nachricht vollständig übertragen wurde. Die Auswertung dieses Structs und die möglichen Befehle, welche durch das Struct übermittelt werden können, sind in Abschnitt 5.5.2 dargestellt.

5.4 SafeSPI

Zur Übertragung der Konfigurations-, Status- und Nutzdaten zwischen dem DUT und dem Testsystem werden, wie bereits beschrieben, SPI-Schnittstellen verwendet. Die SPI-Nachrichten, die über diese Schnittstelle ausgetauscht werden, sind keine DUT-spezifische Implementierung, sondern es wird der SafeSPI-Standard als Grundstruktur verwendet. Dieser spezifiziert einen Data Link Layer bzw. Logical Layer. Er umfasst Vorgaben im Bezug auf die vorhandenen Felder in einem Frame, sowie deren Größe. Ebenfalls Teil des SafeSPI-Standard ist eine 3-Bit CRC-Prüfsumme, welche die ansonsten nicht gegen Bitfehler gesicherte Übertragung absichern soll.
Die Implementierung des SafeSPI-Standards inklusive der Berechnung der CRC-Prüfsumme erfolgt in diesem Testsystem nicht durch den IP-Block der programmierbaren Logik, sondern durch Software. Dies ist möglich, da der SafeSPI-Standard nur Vorgaben für die Verwendung der Bits eines Frames und deren Länge macht. Der einzige Parameter des SafeSPI-Standard, welcher einen direkten Einfluss auf die Hardware hat, ist die Verwendung von SPI-Nachrichten mit einer Länge von $32\,Bit$. Das Testsystem nutzt daher die in Abschnitt 4.1 dargestellte SPI-Schnittstelle, da diese in der Lage ist, Nachrichten mit dieser Länge zu versenden.
Im SafeSPI-Standard wird zwischen vier Frameformaten unterschieden, welche sich auf die Übertragung von Konfigurationsdaten oder Sensordaten aufteilen. Für beide Übertragungszwecke ist jeweils ein Frame im Out-Of-Frame Format oder In-Frame Format definiert. In diesem Testsystem wird zur Kommunikation mit dem DUT ausschließlich das Out-Of-Frame Format verwendet. Darin werden die, der Anfrage des Masters zugehörigen, Daten erst in einem zweiten Frame übertragen. Die vom Master empfangenen MISO-Daten beziehen sich daher immer auf die MOSI-Daten des vorausgegangenen Frames. Soll der Master ein Register des Slaves auslesen, so muss dieser zunächst die gewünschte Adresse in einem Frame übertragen. Die Nutzdaten bleiben bei dieser Übertragung ungenutzt. Anschließend muss er einen zweiten Frame versenden, um die Antwort vom Slave zu erhalten, da MISO und MOSI immer synchron übertragen werden. Für den zweiten Frame muss aus diesem Grund, sofern keine weiteren Daten zu übertragen sind, ein Dummy-Frame genutzt werden, welcher dem Slave anzeigt, dass keine neuen MOSI-Daten vorhanden sind (NOP). Das In-Frame Format des SafeSPI-Standards, bei welchem die Antwort des Slaves bereits in den hinteren Bits des selben Frames übertragen werden, findet keine Verwendung, da die gegenwärtig zu untersuchenden DUTs dies nicht verlangen.

Abb. 5.4: SafeSPI Frame zur Übertragung von Konfigurationsdaten, modifiziert nach [Saf16, S. 14f.]

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOSI	Target Address (10 Bit)										Data (19 Bit)																			CRC		
MISO	D	Source Address (10 Bit)									Data (18 Bit)																			CRC		

Anmerkungen:
Das Feld D ist bei der Übertragung von Konfigurations- oder Statusdaten (keinen Sensordaten) immer mit 0 belegt.
Da das Out-Of-Frame Protokoll genutzt wird, gehören die MISO / MOSI Daten zu unterschiedlichen Frames.
Um die Antwort eines Frames zu erhalten, muss noch ein Dummy-Frame (NOP) übertragen werden.

Abb. 5.5: SafeSPI Frame zur Übertragung von Sensordaten, modifiziert nach [Saf16, S. 14f.]

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOSI	Target Address (10 Bit)										Data (19 Bit)																			CRC		
MISO	D	Source Address (10 Bit)									S1	Data (16 Bit)																	S0	CRC		

Anmerkungen:
Das Feld D ist bei der Übertragung von Sensordaten immer mit 1 belegt.
Da das Out-Of-Frame Protokoll genutzt wird, gehören die MISO / MOSI Daten zu unterschiedlichen Frames.
Um die Antwort eines Frames zu erhalten, muss noch ein Dummy-Frame (NOP) übertragen werden.

Der Frame zur Übertragung von Konfigurationsdaten ist in Abbildung 5.4 dargestellt und besteht für die Datenrichtung vom Master zum Slave aus der Registeradresse des Slaves, welche gelesen oder beschrieben werden soll und den eigentlichen Daten. Die Registeradresse ist 10 *Bit* breit, sodass nach Subtraktion der drei Bits der Prüfsumme noch 19 *Bit* für Nutzdaten bereitstehen. In Gegenrichtung besitzt die Antwort des Slaves noch das zusätzliche Bit D, sodass die Nutzdaten der Antwort maximal 18 *Bit* breit sein dürfen. Das Bit D zeigt bei einer null an, dass es sich bei der Antwort des Slaves um Konfigurations- oder Statusdaten handelt. Hat das Bit D den Wert eins, so enthält die Antwort des Slaves Sensordaten. In diesem Fall ist der Antwortframe geringfügig anders aufgebaut. Es werden, wie in Abbildung 5.5 dargestellt, noch zwei Statusbits S1 und S0 eingefügt, welche den Status des Sensors anzeigen und ob dessen Daten gültig sind. Die Sensordaten eines Frames dürfen maximal 16 *Bit* breit sein. Sollte ein Sensorwert eine größere Länge aufweisen, so muss dieser auf mehrere Frames verteilt übertragen werden.
Der Logical Layer des SafeSPI-Standards macht noch weitere Vorgaben bezüglich der Verwendung der Bits der einzelnen Felder eines Frames, lässt dabei jedoch viele Freiräume für die Implementierung, sodass das Testsystem gegebenenfalls angepasst werden muss, wenn die Implementierung der DUTs sich unterscheiden. Nicht vorgeschrieben ist z.B. wie zwischen schreibendem und lesendem Zugriff unterschieden wird oder ein Dummy Frame bzw. NOP-Befehl auszusehen hat.
Die vom SafeSPI-Standard spezifizierte Prüfsumme ist eine 3-Bit CRC, welche das Polynom $g(x) = x^3 + x + 1$ zur Generation der Prüfsumme nutzt. Das CRC-Polynom einer SafeSPI ist identisch mit dem einer PSI5-Schnittstelle. Allerdings ist der Wert, mit welchem die Speicher des in Abbildung 5.6 dargestellten Polynoms, vor der eigentlichen Generati-

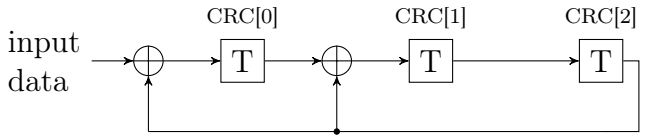


Abb. 5.6: CRC-Polynom der SafeSPI, Polynomdefinition nach [Saf16, S. 17]

on der Prüfsumme initialisiert werden, nicht identisch mit dem einer PSI5-Schnittstelle. Die SafeSPI-Standard schreibt einen Initialwert von 0b101 vor. Alle Bits eines SafeSPI-Frames werden durch die 3-Bit CRC geschützt. Da zum Zeitpunkt der Generierung der Wert der CRC-Bits in einem Frame nicht bekannt ist, werden diese mit null angenommen. Eine alternative Betrachtungsweise ist, dass alle 29 zum Zeitpunkt der Generierung der Prüfsumme gültigen Bits durch das CRC-Polynom geschoben werden und zum Abschluss noch drei Nullen folgen. Der SafeSPI-Standard bezeichnet die drei folgenden Nullen als Target Value (vgl. [Saf16, S. 17]).
Die Generierung der Prüfsumme erfolgt nicht in der programmierbaren Logik, sondern in den Softwareroutinen, welche auch den SafeSPI-Frame zusammensetzen. Dazu wird

zunächst der Frame aus den Feldern Target Adress und Data, welche zusammen 29 Bit des Frames ausmachen, zusammengestellt. Anschließend werden diese 29 Bit bitweise in die Struktur des in Abbildung 5.6 dargestellten Polynoms geschoben. Die softwaretechnische Umsetzung dieser Struktur ist gegenüber der Darstellung auf einen weiteren Speicher angewiesen, da ansonsten nach der Übernahme des Wertes von CRC[1] in den Speicher von CRC[2], der ehemalige Wert von CRC[2] nicht mehr vorhanden wäre, welcher jedoch für die XOR-Verknüpfung am Eingang der Struktur benötigt wird.

5.5 FreeRTOS Funktionsablauf

Die FreeRTOS-Instanz des Testsystems wird von der Ablaufsteuerung, die als Anwendung unter Linux läuft, beim Beginn eines Tests gestartet. Danach erzeugt sie als ersten Schritt einen Mutex, welcher die exklusiven Zugriffsrechte auf die Konfigurations-SPI-Schnittstelle verwaltet. Eine Verwendung dieser Schnittstelle durch den Linux-Kernel ist vorgesehen, sodass es ausreichend ist, wenn die FreeRTOS-Instanz die Zugriffsrechte ausschließlich für ihre eigenen Tasks verwaltet. In einem zweiten Schritt werden zwei Tasks erzeugt, von denen sich einer um die Initialisierung des DUTs und der andere um die Schnittstellen zum anderen CPU-Kern mit der Ablaufsteuerung kümmert.
Die Initialisierung des DUTs besteht im wesentlichen aus der Konfiguration der in Abschnitt 4.7 beschriebenen Taktquelle zur Versorgung des DUTs mit einem Systemtakt und dem Setzen von Steuersignalen. Letzteres kann gegebenenfalls auch komplexer ausfallen und aus sequentiellen Bitmustern für typische Steuereingänge eines DUTs, wie Reset, Enable oder Testpins, bestehen. Die Ansteuerung der eigentlichen Ausgangspins des FPGAs erfolgt über einen GPIO-IP-Block, welcher über ein AXI-Interface angesprochen wird, sodass die FreeRTOS-Instanz nur Speicherzugriffe ausführen muss. Da das Speichermanagement von FreeRTOS deutlich einfacher als das des Linux-Kernels ist, wird hier kein Mapping des Speichers oder Treibers benötigt, sondern nur ein Zeiger auf die entsprechende Adresse des Registers des AXI-Interfaces im C-Quellcode. Ist die Initialisierung des DUTs abgeschlossen, beendet sich der für diese Aufgabe zuständige Task selbst. Die Konfigurationsregister des DUTs werden durch diese Initialisierung nicht beschrieben, sondern nur externe Steuereingänge des DUTs bedient. Die Konfiguration der Register des DUTs wird durch die Ablaufsteuerung durchgeführt.
Die Initialisierung der Kommunikation zwischen den beiden Betriebssystemen ist im Vergleich zur Initialisierung des DUTs deutlich aufwändiger und erfolgt in mehreren Stufen. Zunächst wird durch den zweiten Task der Hardware Interface Layer (HIL) initialisiert, welcher die Ressourcen der Schnittstellen zwischen den beiden CPU-Kernen für die FreeRTOS-Instanz verwaltet. Die nötigen Informationen über die Ressourcen befinden sich in einem Ressourcetable, welches zum Zeitpunkt der Kompilierung von FreeRTOS definiert sein muss. Eine dynamische Anpassung dieser Größen oder Speicherbereiche ist nicht vorgesehen, sondern erfordert eine erneute Generierung der Firmware. In einem zweiten Schritt wird, sobald das virtio-Interface aktiv ist, der rpmsg-Kanal mit dem Identifier rpmsg-freertos-tf-channel zur Kommunikation mit der Ablaufsteuerung initialisiert. Ist die Initialisierung des rpmsg-Kanals erfolgreich verlaufen, rufen dessen Softwareroutinen eine Callback-Funktion auf, welche die Tasks zur Auswertung der rpmsg-Nachrichten erzeugt. Die Funktionsweise der Auswertung der eingehenden rpmsg-Nachrichten ist in Abschnitt 5.5.2 beschrieben. Der Task zur Initialisierung der Kommunikation zwischen beiden CPU-Kernen bleibt während des Tests des DUTs im Ruhezustand, welchen er nur

kurzzeitig unterbricht, um eine Poll-Funktion zur Verwaltung des HILs auszuführen.
Enthält eine rpmsg-Nachricht die Shutdown-Sequenz, wird dies dem Task von der rpmsg-Auswertung über eine globale Variable mitgeteilt. Der Task beginnt daraufhin die Terminierung der FreeRTOS-Instanz. Als erstes werden die in Abschnitt 5.5.2 beschriebenen Tasks zur Auswertung und Ausführung von Befehlen aus rpmsg-Nachrichten beendet und deren Speicher freigegeben. Anschließend werden die Tasks zur Ansteuerung des Watchdogs beendet, sodass dieser Task, welcher ursprünglich zur Initialisierung der Kommunikation zwischen den beiden Betriebssystemen erstellt wurde, der einzige verbleibende Task ist. Der Task schließt dann den rpmsg-Kanal, sodass auf der Gegenseite der Linux-Kernel ebenfalls das zugehörige /dev/rpmsgX-Device freigeben kann. In einem letzten Schritt löscht der Task den HIL, da dieser nicht mehr benötigt wird und beendet sich selbst. Der Task-Sheduler der FreeRTOS-Instanz besitzt daraufhin keinen Task mehr, welcher auszuführen ist, sodass der zweite CPU-Kern des ZYNQs abgeschaltet werden kann, ohne dass es zu inkonsistenten Zuständen oder anderen Problemen kommt.

5.5.1 Watchdog

Einige der mit diesem Testsystem zu untersuchenden DUTs besitzen einen Watchdog, dessen Aufgabe es ist, zu überwachen ob der das DUT ansteuernde Mikrocontroller korrekt funktioniert. Bei der Implementierung eines typischen Watchdogs muss sich der Mikrocontroller mit einem definierten Intervall bei dem DUT melden, sodass dieses erkennen kann, dass der Mikrocontroller weiterhin aktiv ist. Dies kann durch das Übermitteln einer definierten Nachricht über eine der durch den Mikrocontroller bedienten Schnittstellen des DUTs oder durch eine separate Signalleitung zwischen den beiden erfolgen. Kommt es zu einer Zeitüberschreitung, weil die nächste Nachricht verzögert ist oder ausbleibt, geht das DUT davon aus, dass es zu einem Fehler in der Software und einem Ausfall des Mikrocontrollers gekommen ist. Das DUT kann dann dementsprechend sein Verhalten anpassen, indem es zum Beispiel einen Reset ausführt.
Eine einfache Abwandlung eines Watchdogs mit Auswertung einer Zeitüberschreitung ist ein Watchdog mit einem Zeitfenster, in welchem der Mikrocontroller antworten muss. Dieser besitzt zwei Schwellen, die den Beginn und das Ende des Zeitfensters definieren und welche der Mikrocontroller nicht verletzen darf. Solche einfachen Watchdogs sind durch ihre simplen Bedingungen, die ein Mikrocontroller erfüllen muss, damit der Watchdog nicht auslöst, nicht in der Lage komplexe Fehler des Mikrocontrollers oder der Software festzustellen. Insbesondere für DUTs, welche in sicherheitskritischen Anwendungen verbaut werden, kann es daher vorkommen, dass ein einfacher Watchdog nicht ausreicht, um die Sicherheitsanforderungen an das Gesamtsystem zu erfüllen.
Als Alternative bieten sich daher Challenge-Response-Watchdogs an, welche keine vor-

definierten Nachrichten zwischen Mikrocontroller und DUT besitzen. Die Abbildung 5.7 zeigt das Prinzip eines solchen Watchdogs. Das DUT stellt zunächst eine Nachricht bereit,

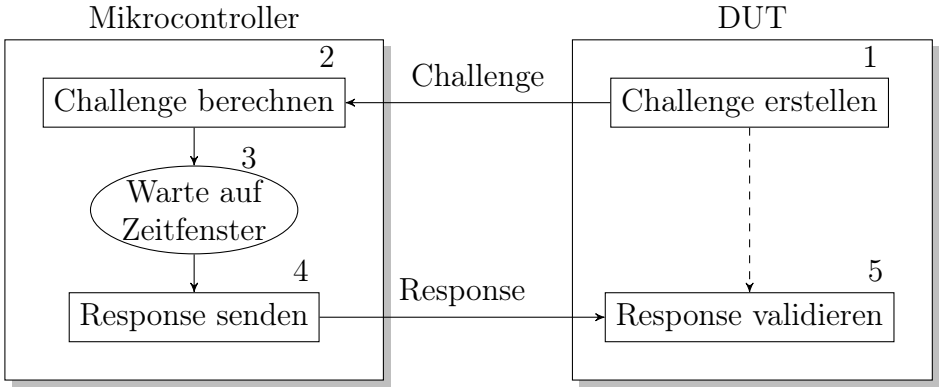


Abb. 5.7: Challenge-Response-Watchdog Verfahren

welche über eine der Schnittstellen zwischen dem DUT und dem Mikrocontroller übertragen wird. Diese Nachricht enthält einen Startwert einer Aufgabe (Challenge), welche der Mikrocontroller zu lösen hat. Die Aufgabe besteht aus einer Reihe von logischen und arithmetischen Operationen, welche der Mikrocontroller auf dem übertragenen Startwert ausführt. Der durch die Operationen entstehende Wert ist die Antwort (Response), welche der Mikrocontroller zurück zum Watchdog des DUTs sendet. Das DUT kann daraufhin untersuchen, ob die erhaltene Antwort der erwarteten Lösung der Aufgabe entspricht. Der Challenge-Response-Watchdog ist dabei ebenfalls mit einem Zeitfenster gekoppelt, sodass der Mikrocontroller nicht nur die korrekte Antwort erzeugen, sondern auch diese zum korrekten Zeitpunkt dem DUT präsentieren muss. Damit wird zum einen die Fähigkeit des Mikrocontrollers überwacht, logische und arithmetische Operationen korrekt auszuführen und zum anderen eine Zeitüberschreitung z.B. durch verzögerte Softwareroutinen erkannt.[SS18]

Das Testsystem muss in der Lage sein einen Watchdog zu bedienen, da ansonsten die zu untersuchende DUTs, welche einen solchen Challenge-Response-Watchdog besitzen, gegebenenfalls nicht oder nur eingeschränkt funktionieren. Die Ansteuerung des Watchdogs wird durch einen Task des Echtzeitbetriebssystems FreeRTOS durchgeführt, da die Antworten für den Watchdog in einem definierten Zeitfenster stattfinden müssen. Die Implementierung dieser Funktionalität unter Linux ist nicht möglich, da es je nach Systemauslastung zu einer Verzögerung kommen kann, welche gegebenenfalls das Zeitfenster für die Antwort verletzt. Die FreeRTOS Instanz dieses Testsystems verfügt über einen eigenen Task für einen Watchdog. Dabei kann über die Priorität der Watchdog Tasks definiert werden, welcher Watchdog zuerst bearbeitet werden soll, für den Fall, dass mehrere auf den selben Ausführungszeitpunkt fallen. Die Tasks, welche den Watchdog ansteuern, besitzen dabei eine höhere Priorität als die restlichen Tasks, sodass sichergestellt wird, dass das DUT nicht in einen Fehlerzustand geht, wenn das System viele rpmsg-Anfragen

der Ablaufsteuerung des Testsystems abarbeiten muss.
Der Watchdog des DUTs kommuniziert mit der FreeRTOS Instanz über die SPI-Schnittstelle, welche auch zur Konfiguration des DUTs genutzt wird. Die einzelnen Funktionen, welche für den SPI-Transfer aufgerufen werden, stellen über Mutexes sicher, dass auf den IP-Block mit dem SPI-Interface nur von einem Task aus exklusiv zugegriffen werden kann. Der Zugriff der anderen Tasks ist dabei so lange blockiert, bis der SPI-Transfer abgeschlossen ist und die Daten des Blocks ausgelesen wurden. Erst dann darf der Taks mit der nächsthöheren Priorität das SPI-Interface nutzen. So erhält die Ansteuerung des Watchdogs immer den Vorrang vor anderen SPI-Nachrichten. Diese Ansteuerung erfolgt in zwei Phasen. In der ersten Phase werden dem Watchdog einmalig Konfigurationsdaten über die SPI-Schnittstelle zugesandt. Diese konfigurieren gegebenenfalls vorhandene Parameter des Watchdogs und starten diesen. In der nun folgenden zweiten Phase wird, wie in Abbildung 5.8 zu erkennen ist, zunächst die Challenge (Aufgabe) über die SPI-Schnittstelle

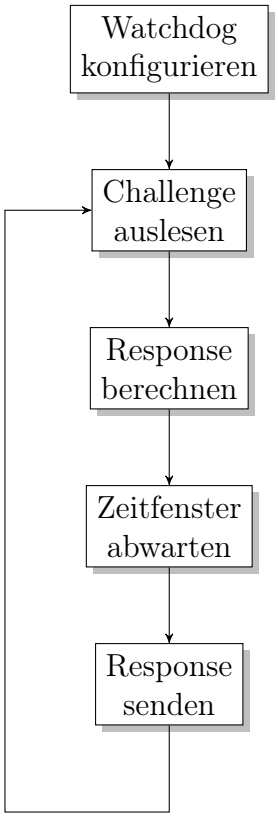


Abb. 5.8: Ansteuerung des Challenge-Response-Watchdogs

aus dem DUT ausgelesen. Die Challenge ist ein Wert auf dem dann die logischen und arithmetischen Operationen ausgeführt werden, um den Antwortwert zu erhalten. Dieser wird nicht direkt zum Watchdog zurückgesendet, sondern der Task wartet ab, bis sich das Zeitfenster für die Antwort öffnet. In der Zwischenzeit können andere Tasks ihre Aufgaben ausführen. Der Task wartet dabei für einen definierten Zeitraum, da der Watchdog das Zeitfenster relativ zum Abruf der Challenge startet. Sind die Zeitfenster des Watch-

dogs in Relation zu einer absoluten Zeit, ist es sinnvoll den Task der Ansteuerung des Watchdogs ebenfalls bis zu einem absoluten Zeitpunkt warten zu lassen, bis dieser die Response sendet. Nach deren Versand ist der gegenwärtige Zyklus abgeschlossen und ein neuer beginnt mit dem erneuten Abfragen der Challenge des DUTs. Dieser Zyklus wird endlos wiederholt, bis die FreeRTOS Instanz den Task der Ansteuerung des Watchdogs beendet, was z.B. passiert, wenn ein Testdurchlauf des Testsystems abgeschlossen ist.

5.5.2 Auswertung der rpmsg

Die als rpmsg-Nachricht versandten Befehle, die von der Ablaufsteuerung, welche unter Linux läuft, ausgehen, werden von der FreeRTOS Instanz ausgewertet und die entsprechenden Befehle umgesetzt. Die Auswertung erfolgt dabei in mehreren Schritten und durch mehrere Tasks. Dies ist notwendig, da die Ausführung der Befehle gegebenenfalls länger dauern und es ansonsten zu einer Blockade der rpmsg-Verbindung zwischen beiden Betriebssystemen bzw. CPU-Kernen kommen kann. Gegenwärtig sind in dem Testsystem vier Befehle spezifiziert, welche die FreeRTOS Instanz ausführen kann. Die Struktur einer rpmsg-Nachricht, mittels welcher die Befehle und Antworten ausgetauscht werden, ist in Abschnitt 5.3.2 aufgeführt. Der folgende Abschnitt beschreibt die Befehle und ihre Parameter, sowie deren Auswertung und Umsetzung.
Erhält die FreeRTOS Instanz eine rpmsg-Nachricht über ihren zum Linux des ersten CPU-Kerns verbundenen Kanal, wird eine Callback-Funktion aufgerufen, welche einen Pointer auf die Speicherstelle der Nachricht und deren Länge enthält. Die Callback-Funktion prüft in einem ersten Schritt, ob die rpmsg-Nachricht der Shutdown-Nachricht entspricht (vgl. Abschnitt 5.3.2). Ist das der Fall, wird die globale Variable zum Beenden der FreeRTOS Instanz gesetzt und die Funktion beendet. Andernfalls wird geprüft, ob die Länge der erhaltenen Nachricht der Größe des Struct aus Abbildung 5.3 von $4 \times 32\,Bit$ entspricht. Anschließend wird das Struct in einer Queue platziert.
Die Queue, welche sich in Abbildung 5.9 im oberen Bereich der Darstellung befindet, dient zur Übergabe der rpmsg-Nachrichten bzw. Structs von der Callback-Funktion zum Task, welcher die Auswertung durchführt. Sollte die Auswertung und Ausführung der Befehle einer rpmsg-Nachricht länger dauern als das Intervall zwischen den eingehenden Nachrichten, werden diese so lange in der Queue gelagert, bis der Task der Auswertung sie nacheinander abarbeitet. Diese Zwischenspeicherung ermöglicht die Annahme von kurzfristigen Bursts an rpmsg-Nachrichten. Ist die Rate der eingehenden Daten jedoch dauerhaft höher als die Abarbeitung der Anfragen, so läuft die Queue voll und es kommt zu einer Blockade der Callback-Funktion, da diese die neuen Daten erst in der Queue ablegen kann, wenn ein Element abgearbeitet und aus der Queue entfernt worden ist. Die gegenwärtige Größe der Queue beträgt zehn Speicherplätze für je ein Struct.

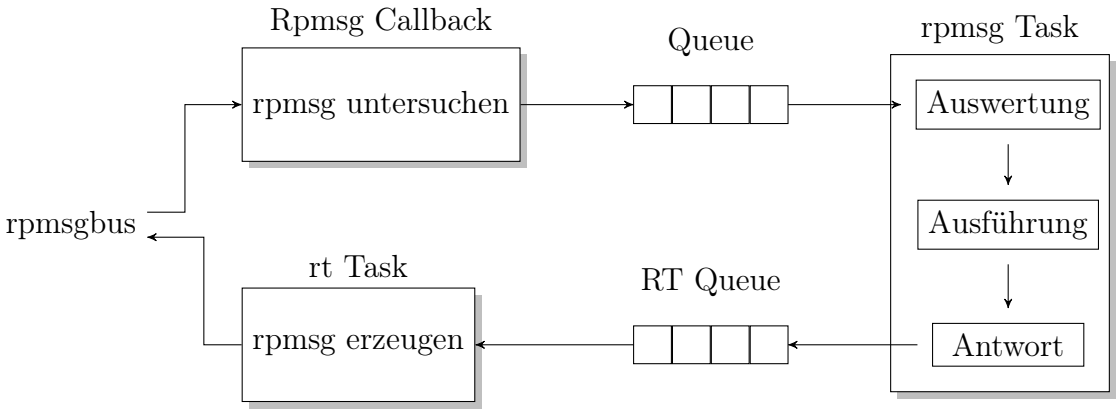


Abb. 5.9: Auswertung der rpmsg Nachrichten

Der zur Auswertung eingesetzte Task besitzt eine geringere Priorität, als die Tasks der Watchdogs, da die Aufgaben der Watchdog-Tasks zeitkritischer sind, als alle möglichen rpmsg-Befehle. Wird die SPI-Schnittstelle zum Transfer von Daten zum DUT gegenwärtig von der Watchdog Ansteuerung genutzt, muss der Task warten, bis das Interface wieder freigegeben ist.
Die Auswertung, welcher Befehl von der FreeRTOS Instanz ausgeführt werden soll, erfolgt anhand des Command-Feldes der rpmsg-Nachricht. Die Tabelle 5.1 zeigt dabei alle möglichen Befehle und deren Command-Werte. So können beispielsweise Register des DUTs

Tabelle 5.1: rpmsg Befehlsübersicht

Befehl	**Status**	**Command**	**Address**	**Value**
Read SPI	x	0xF1	Register Address (10 Bit)	x
Write SPI	x	0xF2	Register Address (10 Bit)	Reg. Data (16 Bit)
Start WD	x	0xF3	x	WD Config (16 Bit)
Stop WD	x	0xF4	x	x

über die Konfigurations-SPI ausgelesen oder beschrieben werden. Soll ein Register ausgelesen und der Registerinhalt an die Ablaufsteuerung zurückgesendet werden, muss als zusätzlicher Parameter die zu lesende Register-Adresse des DUTs im Address-Feld hinterlegt werden. Bei einem schreibenden Zugriff auf ein Register muss neben dessen 10 *Bit* Adresse auch noch der gewünschte Registerinhalt im Value-Feld angegeben sein. Da die zu untersuchenden DUTs 16-Bit Register besitzen, werden aktuell nur die untersten 16 *Bit* des Value-Felds an das DUT übertragen. Das Struct einer rpmsg-Nachricht ist mit je 32 *Bit* pro Feld ausreichend groß für eine zukünftige Anpassung an andere Registergrößen (vgl. Abschnitt 5.3.2). Wird ein Feld einer rpmsg-Nachricht nicht vollständig genutzt, befinden sich die gültigen Bits immer an den niederwertigsten Bitpositionen und die restlichen Bits werden mit Null aufgefüllt. Über die Befehle mit den Command-Werten 0xF3 und F4 kann die Ansteuerung des Watchdogs konfiguriert und gestartet bzw. beendet werden. Allen Befehlen ist gemein, dass sie das Status-Feld nicht nutzen. Dieses wird nur für die

rpmsg-Antwort verwendet.
Wurde der Befehl ausgeführt und hat der Tasks bei einem lesenden SPI-Zugriff die Antwort vom DUT erhalten, wird die rpmsg-Antwort für die Ablaufsteuerung zusammengesetzt. Diese besteht aus dem selben $4 \times 32\,Bit$ rpmsg-Struct, wie der Befehl. Wie aus Tabelle 5.2 zu entnehmen ist, bleibt das Command und Address-Feld identisch, sodass die Ablaufsteuerung die Antwort einem vorausgegangenen Befehl zuordnen kann. Bei einem schreibenden

Tabelle 5.2: rpmsg Befehlsübersicht (Antworten)

Befehl	**Status**	**Command**	**Address**	**Value**
Read SPI	SPI Status	0xF1	Reg. Address (10 Bit)	Reg. Data (16 Bit)
Write SPI	0x0	0xF2	Reg. Address (10 Bit)	Reg. Data (16 Bit)
Start WD	0x0	0xF3	0x0	WD Config (16 Bit)
Stop WD	0x0	0xF4	0x0	0x0

Zugriff wie dem Start des Watchdogs und dessen Konfiguration oder einem schreibenden Zugriff auf ein Register des DUTs über die SPI enthält das Value-Feld der Antwort die unveränderten Daten des Befehls. Bei einem lesenden Zugriff wird das Value-Feld mit den ausgelesenen Daten gefüllt. Zusätzlich enthält bei einem lesenden SPI-Zugriff das Status-Feld einen Statuswert, anhand dessen die Ablaufsteuerung erkennen kann, ober der SPI-Zugriff erfolgreich war oder Fehler, wie eine ungültige Prüfsumme, aufgetreten sind. Bei einem schreibenden Zugriff wir stattdessen das Status-Feld nach Abschluss des Transfers mit 0x0 gefüllt. Ob der Zugriff auf das Register des DUTs erfolgreich war, lässt sich letztendlich damit jedoch nicht feststellen. Soll ein schreibender Zugriff verifiziert werden, ist es notwendig, dass die Ablaufsteuerung einen neuen rpmsg-Befehl sendet, welcher das zuvor beschriebene Register ausliest und den tatsächlichen Registerwert mit dem Erwartungswert vergleicht.
Die erzeugte Antwort wird von dem Auswerte-Task in die Return-Queue geschoben (vgl. Abbildung 5.2). Diese Return-Queue dient der Kommunikation zwischen dem Auswerte-Task und dem Task, welcher die Antwort über den rpmsg-Bus zurück an das Linux der ersten CPU zurücksendet. Der zusätzliche Task für das Zurücksenden einer rpmsg-Nachricht wird benötigt, damit, falls der rpmsg-Bus gerade in Verwendung ist, nicht der Task zur Abarbeitung der Befehle blockiert wird. Der Task zum Zurücksenden der rpmsg-Nachricht rt_task besitzt gegenüber den anderen Tasks einen sehr begrenzten Funktionsumfang. Er wartet endlos auf das Vorhandensein von Antworten in der Return-Queue, entnimmt diese aus der Queue und versendet sie über den rpmsg-Bus. Anschließend wartet der Task erneut auf Nachrichten in der Return-Queue.

5.6 Treiber

Die Ablaufsteuerung, welche als Anwendung im Userspace des Linux-Kernels läuft, benötigt Treiber, um aus dem Userspace mit anderen Komponenten des Testsystems zu kommunizieren. Die Kommunikation mit der FreeRTOS-Instanz des zweiten CPU-Kerns erfolgt über rpmsg. Die Anwendung muss außerdem in der Lage sein, die IP-Blöcke, welche sich in der programmierbaren Logik befinden, zu steuern. Das Beschreiben der entsprechenden Register der AXI-Interfaces der IP-Blöcke erfolgt über eine Speicherabbildung. Damit diese auch für Anwendungen aus dem Userspace möglich ist, wird ein UIO-Treiber im Linux-Kernel eingesetzt, welcher die Brücke in den Userspace bildet. Beide Treiber werden im Folgenden näher beschrieben.

5.6.1 rpmsg

Für die Kommunikation über rpmsg aus dem Userspace wird der in Abschnitt 5.3.2 beschriebene Treiber von Xilinx verwendet. Dieser Character-Treiber erzeugt einen Eintrag unter /dev/rpmsgX, welchen Anwendungen über einen Filehandler ansprechen können. Das X in dem Pfad entspricht dabei einer Zahl, mit welcher die Instanzen eines jeden rpmsg-Kanals differenziert werden. Die Nummerierung beginnt dabei mit Null und wird vom Kerneltreiber für jeden neuen aktiven rpmsg-Kanal um eins inkrementiert. Da eine eindeutige Identifizierung eines Kanals über die Nummer nicht möglich ist, muss gegebenenfalls über das virtuelle Dateisystem Sysfs, welches Informationen über die Kernelsubsysteme bereitstellt, geprüft werden, welchen eindeutigen Namen der rpmsg-Kanal hat. Die Anwendung muss dafür wie im Beispiel-Quellcode 5.5 dargestellt, einen Filehandler

```
1     int rpmsg_fd;
2
3     rpmsg_fd = open(rpmsg_dev, O_RDWR);
4     if (rpmsg_fd < 0) {
5         //Error
6     }
```

Quellcode 5.5: rpmsg Filehandler Initialisierung

,

auf das entsprechende Interface öffnen, welcher einen lesenden und schreibenden Zugriff gewährt. Die Variable rpmsg_dev entspricht dabei dem Pfad zum Interface des Character-Treibers für den jeweiligen rpmsg-Kanal. Da das Testsystem gegenwärtig nur über einen rpmsg-Kanal verfügt, ist dieser Pfad im Testsystem immer /dev/rpmsg0 und eine Verifizierung über die Einträge im Sysfs-Dateisystem kann entfallen.
Ist der Filehandler geöffnet, können Nachrichten an die FreeRTOS-Instanz auf dem rpmsg-Bus über den write-Befehl verschickt werden. Dazu wird wie im Beispiel-Quellcode 5.6 zu

erkennen ist, neben dem Filehandler die zu sendenden Daten und die Länge dieser Daten benötigt. In diesem Testsystem bestehen sämtliche versandte Nachrichten aus dem

```
1     struct _rpmsg x;
2
3     if(write(rpmsg_fd, &x, sizeof(x)) != sizeof(x)){
4         //Error
5     }
```

Quellcode 5.6: rpmsg Nachricht versenden

,

im Abschnitt 5.3.2 dargestellten Struct, sodass die Länge einer jeden Nachricht konstant ist. Anhand des Rückgabewertes der write-Funktion kann bestimmt werden, ob ein Fehler während der Übertragung aufgetreten ist. Eine einwandfreie Übertragung wird durch einen Rückgabewert von 0 signalisiert. Das Lesen eines Structs aus dem rpmsg-Kanal wird durch die read-Funktion durchgeführt. Da die im Beispiel-Quellcode 5.7 dargestellte Verwendung der read-Funktion den ausführenden Thread blockiert, wenn keine Nachricht vorhanden ist, müssen nachfolgende Maßnahmen ergriffen werden, um eine Blockade der Anwendung zu verhindern. Das Testsystem prüft über die select-Methode vor dem

```
1     struct _rpmsg y;
2     int length;
3
4     do{
5         length = read(rpmsg_fd, &y, sizeof(y));
6     }while(length < sizeof(y));
```

Quellcode 5.7: rpmsg Nachricht auslesen

,

eigentlichen Lesen einer Nachricht, ob solch eine vorhanden ist. Ist keine Nachricht vorhanden, kann mittels eines Timeouts der select-Methode verhindert werden, dass es zu einer vollständigen Blockade der Anwendung kommt. Die Anwendung kann dann mit einer Fehlermeldung beendet werden, wenn sie nicht mehr in der Lage ist, mit der FreeRTOS-Instanz zu kommunizieren oder Nachrichten von ihr zu erhalten. Bei einer Beendigung der Anwendung, oder wenn die FreeRTOS-Instanz nicht mehr benötigt wird, ruft die Anwendung die close-Methode auf, welche den Filehandler schließt. Der rpmsg-Treiber versendet daraufhin unabhängig von der Anwendung die Shutdown-Nachricht zur Beendigung der FreeRTOS-Instanz an selbige (vgl. Abschnitt 5.5 und 5.3.2).

5.6.2 UIO Treiber

Die Ablaufsteuerung unter Linux benötigt einen Treiber, um die IP-Blöcke der programmierbaren Logik ansteuern zu können. Darunter fällt zum Beispiel neben dem PSI5-

Patterngenerator auch eine der beiden SPI-Schnittstellen zur Kommunikation mit dem DUT. Während die Konfigurations-SPI-Schnittstelle durch die FreeRTOS-Instanz verwaltet wird und die Ablaufsteuerung daher die Konfigurationsdaten über den rpmsg-Bus zuerst an die FreeRTOS-Instanz sendet, erfolgt die Ansteuerung des Nutzdaten-SPI-Interfaces direkt durch die Ablaufsteuerung aus Linux.
Der Treiber für die IP-Blöcke umfasst in diesem Testsystem nur einen geringen Funktionsumfang, sodass die Implementierung eines vollständigen individuellen Treibers für jedes Modul einen unverhältnismäßig hohen Entwicklungsaufwand bedeuten würde. Die Alternative dazu besteht in der Verwendung des UIO-Treibers, einem universellen Treiber, der die Verlagerung der spezifischen Logik eines Treibers in den Userspace erlaubt. Der eigentliche UIO-Treiber besteht nur aus einem universellen Modul, welches den Zugriff auf die Adressen des externen Devices aus dem Userspace erlaubt. Die eigentliche Logik des Treibers wird dann durch die Anwendung implementiert. Der Kernel erkennt durch die Einträge im Device Tree, für welche Devices er den UIO-Treiber laden soll (vgl. Abschnitt 5.3.1). Für jedes dieser Devices wird dann eine Interfacedatei unter /dev/uioX angelegt, über welches das Device angesprochen werden kann. [Han06] Da analog zum rpmsg-Treiber eine Identifizierung nicht alleine über die Ziffer am Ende des Pfads der Interfacedatei erfolgen kann, müssen gegebenenfalls Informationen über das virtuelle Dateisystem Sysfs eingeholt werden.
Die Implementierung des Treibers für die AXI-Interfaces besteht im wesentlichen aus der Abbildung der Register in einen Speicherbereich, auf welchen innerhalb von Anwendungen zugegriffen werden kann, sodass die Anwendung die entsprechenden Bits lesen oder setzen kann. Dazu wird zunächst ein Filehandler auf die Interfacedatei /dev/uioX des UIO-Treibers erzeugt, welcher der Anwendung ermöglicht, auf den Treiber und durch dessen Weiterleitung der Anfragen durch die Struktur des Kernels, auf die AXI-Interfaces der IP-Blöcke zuzugreifen. Das erste Byte der Interfacedatei /dev/uioX des UIO-Treibers entspricht dabei ebenfalls dem ersten Byte der Basisadresse des AXI-Interfaces. Die eigentliche Abbildung der Register des AXI-Interfaces in einen, der Anwendung zugewiesenen, Speicherbereich erfolgt, wie in dem Quellcodebeispiel 5.8 gezeigt, durch die mmap-Funktion. Dieser wird als erster Parameter NULL übergeben, was dem Kernel signalisiert, dass er sich die Speicherstelle, an welche das AXI-Interface abgebildet werden soll, aussuchen kann. Der Rückgabewert der Funktion mmap ist die vom Kernel gewählte Speicherstelle, über welche die Anwendung nun aus dem Userspace auf das abgebildete Register eines IP-Blocks zugreifen kann. Über den zweiten Parameter wird die Größe des abzubildenden Speicherbereichs definiert. Dieser ist in diesem Fall für sämtliche AXI-Interfaces aller IP-Blöcke eine Speicherseite (page), da der Kernel standardmäßig $4\,kB$ große Speicherseiten (PAGE_SIZE) verwendet, die ausreichend groß sind, um alle Register des größten AXI-Interfaces abzubilden. Das größte AXI-Interface besitzt der PSI5-Patterngenerator mit $720\,Byte$. Die Attribute PROT_READ und PROT_WRITE teilen

```
1   void *ptr_mmap;
2
3   uio_fd = open(uio_dev, O_RDWR);
4   if (uio_fd < 0) {
5       //Error
6   }
7
8   if ( !(ptr_mmap = (void*) mmap(NULL, PAGE_SIZE, PROT_READ|
        PROT_WRITE, MAP_SHARED, uio_fd, 0x0)) ){
9       //Error
10  }
```

Quellcode 5.8: UIO Initialisierung / Memory Mapping

,

dem Kernel mit, dass schreibende und lesende Zugriffe auf den abgebildeten Speicher erfolgen dürfen. Der letzte Parameter ermöglicht die Angabe eines Offsets. Da jedoch sämtliche Register eines Interfaces beginnend mit der Basisadresse abgebildet werden sollen, ist der Offset mit 0x0 angegeben.
Die Zugriffe auf die Register des AXI-Interfaces der IP-Blöcke erfolgt über den Pointer, welcher auf den Speicher zeigt, an dessen Position die Register eines AXI-Interfaces abgebildet werden. Da der Pointer nur auf die abgebildete Basisadresse des AXI-Interfaces und damit das erste Register zeigt, muss für jedes weitere Register ein Offset auf die Adresse addiert werden. Um die Zeigerarithmetik mit dem Offset vorzunehmen und auf den Inhalt des Registers zugreifen zu können, wird der Pointer auf einen 32 *Bit* unsigned Integer gecastet, da die Register der AXI-Interfaces bei allen in diesem Testsystem verwendeten IP-Blöcken ebenfalls 32 *Bit* breit sind. Ein lesender und schreibender Zugriff erfolgt dabei, wie im C-Quellcode 5.9 dargestellt, analog zu einem Zugriff auf eine gewöhnliche Speicherstelle durch eine Dereferenzierung des Pointers. Wird der abgebildete

```
1   //Schreibender Zugriff
2   *((uint32_t*)(ptr_mmap + offset)) = write_data;
3
4   //Lesender Zugriff
5   read_data = *((uint32_t*)(ptr_mmap + offset));
```

Quellcode 5.9: UIO AXI Registerzugriff

,

Speicher nicht mehr benötigt, wird die Abbildung über den Befehl munmap freigeben. Anschließend kann der zugehörige Filehandler geschlossen werden. Da die Ablaufsteuerung eine Verbindung zu fast allen IP-Blöcken der programmierbaren Logik benötigt, werden beim Start der Anwendung sämtliche Register der Interfaces in den Speicherbereich der Anwendung gemapped und die Pointer auf das jeweils erste Register eines Interfaces bzw. die Basisadressen der Interfaces in einem Array gespeichert. Während der Laufzeit der Anwendung wird für einen Registerzugriff dann nur noch der Index des Arrays benötigt,

über welchen der entsprechende IP-Block und ein etwaiger Offset für die Auswahl des Registers in einem Interface ausgewählt wird.

5.7 Ablaufsteuerung

Die Anwendung zur Ablaufsteuerung ist das zentrale Element der Software des Testsystems. Sie übernimmt die Ansteuerung der IP-Blöcke in der programmierbaren Logik, die Verwaltung der FreeRTOS-Instanz, und die Auswertung der von dem Testsystem erhaltenen Daten des DUTs. Die Anweisungen welche testspezifischen Schritte die Ablaufsteuerung durchzuführen hat, werden in einer Testdatei im JSON-Format hinterlegt. Der Aufbau und Inhalt dieser Datei ist im Abschnitt 5.8 dargestellt.
Wird die Anwendung zur Ablaufsteuerung gestartet, reserviert sie zunächst in einem ersten Schritt Speicher, welcher für die Strings der Log- und Fehlermeldungen verwendet wird. Über einen Flag beim Aufruf der Anwendung kann der Anwender entscheiden, ob die Logdatei auch Informationen zu erfolgreichen Testschritten enthalten soll (Debuginformationen) oder nur aus Fehlermeldungen bestehen soll. Soll die Logdatei sowohl Fehlermeldungen als auch Debuginformationen enthalten, wird der Pointer des C-Strings, welcher die späteren Debuginformationen enthalten soll auf den Pointer des Strings mit den Fehlermeldungen gesetzt. Die beiden Meldungstypen teilen sich dann den selben Stringbuffer und die selbe Logdatei. Andernfalls werden die Debuginformationen verworfen. Konnte dieser Speicher erfolgreich reserviert werden, beginnt die Ablaufsteuerung mit der Initialisierung des UIO-Treibers, wie in Abschnitt 5.6.2 beschrieben ist. Für den Zugriff auf das Interface des Treibers, ebenso wie für die Befehle an das remoteproc-Framework zum Starten und Anhalten der zweiten CPU, benötigt die Anwendung Rootrechte, da diese Aktionen einen großen Eingriff in das System darstellen, welche der Linux-Kernel nicht allen Benutzern gewährt. Nach der Initialisierung des UIO-Treibers wird der zweite CPU-Kern des ZYNQs mit der FreeRTOS-Instanz zur Ansteuerung der Konfigurationsschnittstelle des DUTs gestartet. Sobald die FreeRTOS-Instanz aktiv ist, wird von der Ablaufsteuerung über den rpmsg-Bus und den rpmsg-Treiber eine Kommunikationsverbindung zwischen der Ablaufsteuerung und der FreeRTOS-Instanz erzeugt (vgl. Abschnitt 5.5).
Ist diese Verbindung aktiv, ist das Testsystem bereit für die Ausführung eines Tests. Dazu wird zunächst der aktuelle Zeitstempel vom Timer des Linuxkernels gesichert, welcher als spätere Referenz für Warteanweisungen in Relation zum Testbeginn genutzt wird. Anschließend wird die äußere Ebene der JSON-Testdatei ausgewertet und anhand der Informationen der Einträge dieser Ebene gegebenenfalls der Watchdog des DUTs gestartet. Bei Bedarf versendet die Ablaufsteuerung einen Befehl an die FreeRTOS-Instanz, welche die eigentliche Ansteuerung des Watchdogs übernimmt. Desweiteren wird sowohl in der Logdatei als auch der zusätzlichen Statusdatei der Name des Testablaufs hinter-

legt. Die zusätzliche Statusdatei enthält Informationen über den Namen des Testablaufs, die Gesamtanzahl der Testschritte des Testablaufs, sowie die Nummer des gegenwärtig bearbeiteten Testschritts. Die Statusdatei wird mit jedem Testschritt aktualisiert und dient als Datenquelle z.b. für das in Abschnitt 5.9 beschriebene Webinterface. Mit dem Beschreiben der Statusdatei und Logdatei hat die Ablaufsteuerung alle einmaligen Initialisierungsaktionen vor dem Start der eigentlichen Abarbeitung der Testschritte ausgeführt. Die Ablaufsteuerung beginnt nun mit der Hauptroutine, welche die Testschritte umsetzt. Die Hauptroutine der Ablaufsteuerung, welche in Abbildung 5.10 zu sehen ist, wird in einer Schleife ausgeführt, bis alle Testschritte der Testdatei abgearbeitet sind oder die Ausführung aufgrund eines Fehlers abgebrochen wird. In ihr wird zunächst der Inhalt des

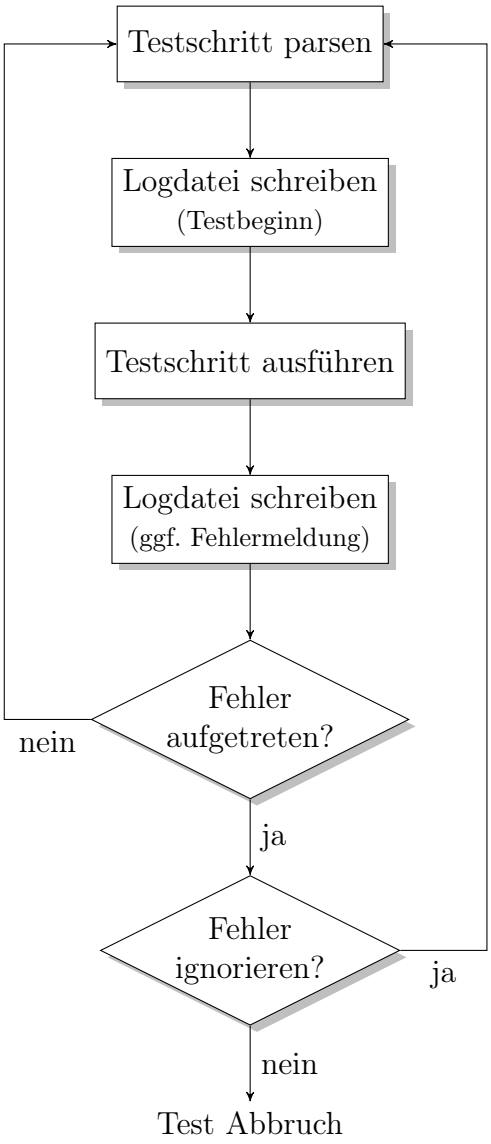


Abb. 5.10: Auswertung der rpmsg Nachrichten

ausführenden Testschritts aus dem JSON-Format in die internen Datenstrukturen der Ablaufsteuerung überführt. Die internen Datenstrukturen der Ablaufsteuerung für Test-

anweisungen bestehen aus Structs, welche einen einfachen Zugriff auf die Daten erlauben, wobei diese jedoch weiterhin nach Anweisung gruppiert sind. Tritt bei dieser Überführung der Daten ein Fehler während des Parsens auf, wird die weitere Ausführung des Testablaufs abgebrochen, da das Testsystem keine gültigen Anweisungen hat. Die Einstellungen in dem Testschritt bezüglich des Ignorieren des Fehlers greifen in diesem Fall nicht, da dieses sich nur auf das Ausführen der Testanweisung beziehen, nicht aber auf das Parsen selbiger.
Als nächster Schritt der Hauptroutine wird sowohl in der Logdatei als auch der Statusdatei der Beginn der Ausführung eines neuen Testschritts vermerkt, sofern die Debugoption aktiviert ist. Dieser Eintrag dient dem Anwender bei einer potentiellen Fehlersuche, um festzustellen, welchen Testschritt das Testsystem angefangen hat, bevor es zu einem Abbruch der Ausführung gekommen ist. Anschließend wird der eigentliche Testschritt entsprechend seiner Anweisungen von der Ablaufsteuerung ausgeführt. Dabei ist es egal, ob es sich z.B. um eine Warteanweisung oder einen Konfigurationsschritt handelt. Alle Testschritte werden in identischer Form ausgeführt. Für Details der einzelnen Testanweisungen siehe deren Definition, welche in Abschnitt 5.8 beschrieben sind. Kommt es bei der Ausführung eines Testschritts zu einer Abweichung von den erwarteten Werten oder einem Fehler, entscheidet die Ablaufsteuerung anhand der Definition für das Verhalten in einem Fehlerfall, die Bestandteil der Anweisungen eines Testschritts ist, ob dieser Fehler ignoriert werden kann und die Ausführung des Test fortgesetzt wird oder der Test abgebrochen wird. Bevor die Ablaufsteuerung die Ausführung weiterer Testschritte abbricht, wird für den gegenwärtigen Testschritt mit den Abweichungen eine Fehlerbeschreibung erzeugt, welche in der Logdatei abgelegt wird. Eine Sonderstellung nehmen hierbei Zugriffe auf das DUT über die beiden SPI-Schnittstellen ein. Treten hier Abweichungen von den erwarteten Register- oder Statusinhalten auf, wird für die Logdatei nicht nur eine generische Fehlermeldung angefertigt, sondern ein String, welcher die einzelnen abweichenden Bits ihrem Erwartungswert gegenüberstellt.
Ist es zu keinen Abweichungen gekommen oder können diese ignoriert werden, beginnt die Ablaufsteuerung mit der Bearbeitung des nächsten Testschritts von vorne. Sind alle Testschritte abgearbeitet oder soll der Test abgebrochen werden, beginnt die Anwendung mit der Beendigung ihrer selbst. Dazu werden zunächst die Filehandler für die Log- und Statusdatei geschlossen. Anschließend werden die von der Anwendung verwendeten Treiber deinitialisiert bzw. die Filehandler zu den Interfaces der UIO- und rpmsg-Treiber geschlossen. Das Schließen des Filehandlers des rpmsg-Treibers führt dazu, dass die FreeRTOS-Instanz des zweiten CPU-Kerns eine shutdown-Nachricht erhält und als Folge mit der Beendigung beginnt. Siehe dazu Abschnitt 5.3.2. Ist der Filehandler des rpmsg-Treibers geschlossen, wartet die Ablaufsteuerung noch eine gewisse Zeit, damit die FreeRTOS-Instanz ihre Ausführung eingestellt hat, bevor sie über das remoteproc-Framework den zweiten CPU-Kern des ZYNQs anhält. Dies ist die letzte Aktion, welche die Ablaufsteue-

rung ausführt, sodass sie mit Abschluss dieser Aktion beendet wird.
Die Ablaufsteuerung ist eine reine Konsolenanwendung ohne grafische Oberfläche. Die grafische Oberfläche wird durch das in Abschnitt 5.9 beschriebene Webinterface umgesetzt. Der dafür benötigte Webserver ist eine eigenständige Anwendung und kein Bestandteil der Ablaufsteuerung. Die Kommunikation zwischen dem Webinterface und der Ablaufsteuerung findet durch die Statusdatei sowie durch die Logdatei statt, welche nach jedem Testschritt geschrieben werden, sodass eine Aktualisierung der Darstellung des Webinterfaces während eines laufenden Tests möglich ist. Der Aufruf der Anwendung sieht wie folgt aus: linux_test_app [Flags] Testdatei Logdatei. In dem Platzhalter 'Testdatei' wird der Pfad und die JSON-Datei spezifiziert, welche die Testschritte des durchzuführenden Tests enthält. Über den Platzhalter 'Logdatei' kann der Anwender den Pfad sowie den Namen zu dem Speicherort angeben, an welchem die Anwendung zur Ablaufsteuerung die Logdatei erzeugt. Neben dieser erzeugt die Anwendung zur Ablaufsteuerung auch die Statusdatei, mit den Informationen über den gegenwärtigen Testschritt. Diese Datei wird dabei immer im selben Dateipfad wie die ausgeführte Binärdatei der Anwendung erzeugt. Eine Einstellung des Speicherorts dieser Datei über einen Parameter beim Aufruf der Anwendung ist aktuell nicht möglich. Die gültigen Flags bei einem Aufruf der Anwendung sind -v, -d und -c. In der gegenwärtigen Version der Anwendung sind die beiden Optionen -v und -d identisch. Beide Optionen aktivieren die zusätzlichen Einträge in der Logdatei, welche alle Testschritte beschreiben und nicht nur die aufgetretenen Fehler. Die -c Option aktiviert die zusätzliche Ausgabe der Logeinträge auf die Konsole. Durch die -c Option wird die Erzeugung der Logdateien nicht deaktiviert, sondern die Ausgabe auf der Konsole findet simultan zu dem Abspeichern in der Logdatei statt.

5.8 JSON Wrapper

Die Definition der Testabläufe erfolgt über eine Datei im JSON-Format. Jeder Eintrag dieser Datei beschreibt einen sequenziell auszuführenden Test- oder Konfigurationsschritt für das Testsystem. Das JSON-Format wurde gewählt, da ursprünglich eine grafische Oberfläche zur Definition des Testablaufs im Webinterface des Testsystems vorgesehen war und das JSON-Format hervorragende Unterstützung durch die gängigen Webtechnologien genießt. Es ist ein schlankes Datenformat, das die Darstellung von Informationen in Objekten, Arrays und Variablen erlaubt, welche lesbar sind. Gegenüber reinen maschinenlesbaren Dateiformaten ermöglicht das JSON-Format eine direkte Erstellung und Bearbeitung durch den Anwender. Es besteht aus Schlüssel-Wert-Paaren, welche die Informationen speichern. Einer der Kernaspekte dieses Formates ist, dass es die Verschachtelung der Informationen erlaubt. Ein Wert zu einem Schlüssel kann aus einem Objekt bestehen, welches wiederum mehrere Schlüssel-Wert-Paare enthält. Diese Verschachtelung wird in dem Testsystem ausgenutzt, um die Informationen eines einzelnen Testschrittes zu gruppieren und damit die Lesbarkeit zu erhöhen. Die erlaubten Datentypen des JSON-Formats sind Strings, numerische Zahlen, boolsche Werte, Nullwerte, Objekte und Arrays.[Ecm17] Arrays können dabei eine geordnete Liste an Werten enthalten oder mit Objekten gefüllt sein. Abweichend von der normalen Konvention, werden in der Software des Testsystems boolsche Werte durch die numerischen Werte `1` (true) und `0` (false) dargestellt.
Die Software zur Ablaufsteuerung des Testsystems nutzt die Bibliothek `json-c` ([Eri18]) als Parser, um den Inhalt einer Datei im JSON-Format in eine Repräsentation durch C-Datenstrukturen zu wandeln. Der eigentliche Inhalt der JSON-Objekte wird dabei nicht durch die `json-c`-Bibliothek geparsed, sondern durch die Ablaufsteuerung, welche den Inhalt eines Testschrittes anschließend in C-Structs zwischenspeichert. Diese werden von der Ablaufsteuerung genutzt, um den Testschritt auszuführen.
In Quellcode 5.10 ist ein Beispiel-Ausschnitt einer Testdatei im JSON-Format zu sehen, die einen Testablauf für das DUT spezifiziert. Das erste Schlüssel-Wert-Paar des Root-JSON-Objekts ist der Schlüssel `name`, hinter dem der Name des Test hinterlegt wird. Dieser Name wird auch bei der Ausführung des Test in der Logdatei vermerkt, sowie im Webinterface angezeigt. Das zweite Feld mit dem Schlüssel `dut_mode` legt fest, ob der Watchdog des DUTs angesteuert werden soll. Ist der Wert dieses Felds `1`, wird der Watchdog durch die FreeRTOS-Instanz aktiviert. Eine `0` als Wert dieses Feldes lässt den Watchdog inaktiv. Eine gesetzte `1` bedeutet dabei nicht, dass das Testsystem mit den ersten Testschritten wartet, bis der Watchdog erfolgreich gestartet wurde. Es empfiehlt sich daher als ersten Testschritt eine Warte-Anweisung einzufügen, bis die Statusregister des DUTs anzeigen, dass der Watchdog aktiv ist. Das letzte Feld des Root-JSON-Objekts besteht aus einem Array, welches die JSON-Objekte sämtlicher Testschritte in chronologischer Ordnung enthält. Alle drei genannten Felder des Root-JSON-Objekts sind

```
{
    "name": "Test 1",
    "dut_mode": 1,
    "test_steps":[
        {
            "name": "Test Step S1",
            "enabled": 1,
            "type": 6,
            "step": {
                "error_mode": 0,
                "update_mode": 1,
                ...
            }
        },
        {
            ...
        },
        ...
    ]
}
```

Quellcode 5.10: Ausschnitt eines Beispiels einer JSON-Testdatei

Pflichtfelder, welche vorhanden sein müssen. Andernfalls bricht die Anwendung der Ablaufsteuerung den Test sofort ab, da sie die Objekte nicht korrekt parsen kann bzw. sie nicht alle benötigten Informationen besitzt. Die Ablaufsteuerung bricht den Test auch ab, wenn in den Objekten der späteren Testschritte ein Syntaxfehler vorhanden ist oder ein Pflichtfeld fehlt.
Die Tabelle 5.3 zeigt den Aufbau und die Funktion eines JSON-Objekts eines Testschritts (vgl. Quellcode 5.10). Dieses Objekt ist für alle Arten von Testschritten identisch, da es nur einen Rahmen bildet und die eigentlichen Anweisungen sich in dem verschachtelten step-Objekt befinden. Das name-Feld gibt den Namen des Testschritts an, welcher für eine

Tabelle 5.3: Felder des JSON-Objekts eines Testschritts

Feldname	**Datentyp**	**Werteber.**	**Plicht**	**Funktion**
name	String		Ja	Bezeichnung des Testschritts
enabled	Zahl	0,1 (Bool)	Ja	Testschritt Aktivierung
type	Zahl	0,1,2,..,7	Ja	Befehlstyp des Testschritts
step	Objekt		Ja	Befehlsanweisungen des Testschritts

bessere Zuordnung der Ausgaben in den Logdateien und dem Webinterface vorgesehen ist. Über das enabled-Feld kann ein Testschritt aktiviert oder deaktiviert werden, sodass auf einfache Weise Variationen von bereits bestehenden Testabläufen unter der Auslassung einzelner Schritte erzeugt werden können. Ist dieses Feld auf 0 gesetzt, überspringt die Ablaufsteuerung den Testschritt und fährt ohne den Inhalt zu parsen mit dem nächsten Testschritt fort. Der Wert des type-Feldes gibt an, um welchen Art von Befehlen es bei der eigentlichen Testschrittanweisung im step-Objekt handelt. Eine Liste mit allen

gegenwärtig verfügbaren Testschrittanweisungen ist in Tabelle 5.4 enthalten. Der Wert

Tabelle 5.4: Numerische Werte der Testschritttypen

type-Wert	**Testschrittanweisung**
0	Warte Anweisung
1	Daten-SPI Zugriff
2	Konfigurations-SPI Zugriff
3	PSI5-Patterngenerator Konfiguration
4	Sensorsimulator DAC Konfiguration
5	Konfiguration des Triggerausgangs
6	Kurzschlusseinspeisung
7	Timer Konfiguration

des type-Feldes muss unbedingt mit dem Inhalt des step-Objekts übereinstimmen, da die Ablaufsteuerung keine andere Möglichkeit besitzt den Typ des Befehls festzustellen. Eine fehlerhafte Zuweisung der Nummer für die Testschrittanweisung führt zwangsweise zu einem Abbruch des Tests, da das Testsystem beim Parsen erwartete Felder dann nicht finden bzw. zuordnen kann. Die maximale Anzahl von Testschritten in einem Test ist nur durch den Wertebereich eines Integers begrenzt, welcher als Index für das Array genutzt wird. Eine zusätzliche künstliche Beschränkung der Schrittanzahl existiert nicht.
Einige Typen von Testanweisungen teilen sich die Definition für den Zeitpunkt, zu welchem Daten übernommen werden sollen und die Einstellungen für das Verhalten der Ablaufsteuerung im Fehlerfall, sodass diese hier einmalig für alle Typen aufgelistet ist. Sollte eine Testanweisung eine Einstellung nicht unterstützen, so ist diese in dessen Tabelle aus dem Wertebereich des entsprechenden Parameters genommen.
Die Definition, wann die internen Funktionen eines IP-Blocks die Daten aus dem AXI-Interface des Blocks übernehmen soll, erfolgt über das update_mode-Feld einer Testschrittanweisung. Eine vollständige Liste aller möglichen Werte ist in Tabelle 5.5 dargestellt. Bei einem Wert von 0 findet keine Übernahme der Werte statt. Diese Option ist dafür

Tabelle 5.5: Numerische Werte des Updatezeitpunkts

update_mode	**Art des Datenübernahme**
0	Kein Update (nur neue Werte)
1	Sofortige Update der neuen Daten
2	Update über Hardware Timer
3	Nur Update (keine neuen Werte)

gedacht, dass die eigentliche Datenübernahme durch eine zweite Testschrittanweisung mit dem Wert 3 zu einem späteren Zeitpunkt ausgelöst wird. Mittels des Werts 1 werden die neuen in dem Testschritt vorhandenen Daten direkt nach dem Empfang durch den IP-Block von selbigem in seine internen Funktionen übernommen. Der Wert 2 startet, ebenso

wie der Wert 0 keine Übernahme der Daten in die Funktionen des IP-Blocks, da bei diesem Wert die Ablaufsteuerung die Übernahme der Daten durch den Hardware Timer konfiguriert. Ist der Wert 2 für das update_mode-Feld einer Testschrittanweisung definiert, so muss diese dementsprechend auch einen Timerwert enthalten, bei dessen Erreichen die Daten übernommen werden.
Der zweite Wert, welcher von mehreren Arten an Testschrittanweisungen geteilt wird, ist der error_mode-Wert, der das Verhalten der Ablaufsteuerung in einem Fehlerfall definiert. Wie aus Tabelle 5.6 ersichtlich ist, liegt der Wertebereich für die Definition des Fehlerverhaltens zwischen 0 und 3. Die Fehlertoleranz der Ablaufsteuerung steigt mit jedem Wert an, sodass beim höchsten Wert sämtliche Fehler und Abweichungen toleriert werden. Im Gegensatz dazu beendet die Ablaufsteuerung beim Wert 0 die Ausführung eines

Tabelle 5.6: Numerische Werte des Fehlerfalldefinition

error_mode	**Art des Datenübernahme**
0	Keinen Fehler ignorieren
1	Abweichende Dateninhalte ignorieren
2	Fehler des DUTs ignorieren
3	Sämtliche Fehler ignorieren

jeden Tests, sofern nicht alle Werte des DUTs und des Testsystems den Erwartungswerten entsprechen. Bei einem Wert von 1 werden abweichende Inhalte von den Daten, die das Testsystem aus den Registern des DUTs erhält, ignoriert, nicht jedoch Fehler in der SPI-Verbindung zum DUT und dem Testsystem selbst. Mittels des Werts 2 können sämtliche Fehler des DUTs ignoriert werden, sodass nur Fehler in der Konfiguration oder dem Ablauf des Testsystems einen Abbruch der Testdurchführung herbeiführen. Im Folgenden werden verschiedene Arten von Testschrittanweisungen (step-Objekten) aufgeführt und erklärt.

5.8.1 Warte-Testschrittanweisung

Die Warte-Testschrittanweisung ermöglicht es Verzögerungen in den Testablauf einzubauen. Diese können dabei je nach Einstellung bis zu einem definierten Zeitpunkt andauern, oder bis ein bestimmter Zustand des DUTs eintritt. Die zugehörigen Felder sind in Tabelle 5.7 zu sehen. Der Wert des Feldes error_mode zur Definition des Verhaltens im Fehlerfall funktioniert wie in Abschnitt 5.8 beschrieben, allerdings sind nur die Optionen 0 und 3 für diesen Befehl gültig. Als nächstes Pflichtfeld der Warte-Testschrittanweisung existiert das wait_mode-Feld, welches die Größe definiert, anhand derer die Software entscheidet wie lange die Verzögerung andauert. Alle gültigen Optionen sind in Tabelle 5.8 dargestellt. In Abhängigkeit dieser Auswahl des Warte-Typs muss das zugehörige Feld für die Größe, welches einer der letzten vier Einträge der Tabelle 5.7 entspricht, in dem JSON-Objekt

Tabelle 5.7: Felder des JSON-Objekts eines Warte-Befehls

Feldname	Datentyp	Werteber.	Plicht	Funktion
error_mode	Zahl	0,3	Ja	Verhalten im Fehlerfall
wait_mode	Zahl	0,1,2,3	Ja	Typ des Warte-Befehls
timeout	Zahl	>0 (int32)	Nein	SPI-Timeout
sys_timer	Zahl	>0 (int64)	Nein	Wartedauer in Systemzeit
hw_timer	Zahl	>0 (uint32)	Nein	Wartedauer in Hardware Timer
sspi_cfg	Objekt		Nein	Daten-SPI Zugriff
mspi_cfg	Objekt		Nein	Konfiguration-SPI Zugriff

Tabelle 5.8: Numerische Werte der Art des Warte-Befehls

wait_mode	Art des Datenübernahme
0	Warten basierend auf Hardware Timer
1	Warten basierend auf System Timer
2	Warten auf Datenregisterinhalt
3	Warten auf Konfigurationsregisterinhalt

der Testschrittanweisung vorhanden sein. Keines dieser Felder ist ein explizites Pflichtfeld, dennoch muss das passende Feld Bestandteil des JSON-Objekts sein. Andernfalls bricht die Ablaufsteuerung den Test ab, da sie das Objekt nicht parsen kann. Ist der Wert von wait_mode 0, wird in diesem Testschritt eine Verzögerung eingebaut, welche so lange andauert bis der Hardware Timer den über das Feld hw_timer definierten Wert erreicht hat. Da dieser in der gegenwärtigen Auslegung eine Auflösung von einer Mikrosekunde besitzt, kann der Wert von hw_timer als Wartezeit in Mikrosekunden seit dem Startzeitpunkt des Timers betrachtet werden. Die Angabe ist immer als relativer Wert zum Startzeitpunkt des Timers zu sehen. Soll mit dem Hardware Timer eine absolute Zeit seit dem Beginn des Testdurchlaufs bestimmt werden, so muss dieser dementsprechend als erster Testschritt gestartet werden. Bei der Ausführung dieses Befehls wird von der Ablaufsteuerung geprüft, ob der Hardware Timer läuft, da die Software andernfalls endlos auf das Erreichen des Timerwerts warten würde. Der Testablauf wird gegebenenfalls bei einem inaktiven Timer abgebrochen.

Mittels eines Wertes des wait_mode-Feldes von 1 findet die Definition der Länge der Verzögerung nicht anhand des Hardware Timers, sondern basierend auf der Systemzeit des Linux-Kernels statt. Die Angabe erfolgt immer im Bezug auf den Start des Test. Die Anwendung der Ablaufsteuerung speichert die Systemzeit zum Start des Tests und berechnet alle weiteren Warte-Testschrittanweisungen auf Systemzeitbasis in Relation zu diesem Zeitpunkt. Die Auflösung des Systemtimers beträgt eine Nanosekunde, sodass auch die Angabe des sys_timer-Felds in Nanosekunden erfolgen muss. Trotz dieser Angabe ist das Erreichen einer solchen Auflösung in der Praxis nicht realistisch. Der Anwender sollte von einer realen Auflösung im Bereich von Mikrosekunden ausgehen.

Eine alternative Verzögerung auf der Basis von festen Zeiten ist eine Warte-Testschrittanweisung, welche auf das Eintreten eines bestimmten Registerwerts der Daten- oder Konfigurations-SPI des DUTs wartet. Dazu wird der wait_mode 3 oder 4 gewählt. Die Einstellungen für den entsprechenden SPI-Zugriff erfolgen dann über das JSON-Objekt des SPI-Zugriffs, welches in dem Feld sspi_cfg oder dem Feld mspi_cfg hinterlegt wird. Dieses Objekt entspricht einer eigenständigen Testschrittanweisung, dessen Einstellungen für einen Zugriff über das Konfigurations-SPI in Abschnitt 5.8.3 und für einen Zugriff über einen Daten-SPI in Abschnitt 5.8.2 auf die vom DUT empfangene PSI5-Nachrichten, beschrieben ist. Beiden ist gemein, dass sie über eine Maske verfügen, sodass die Ablaufsteuerung nicht nur auf ein spezifischen kompletten Registerinhalt, sondern auch nur auf bestimmte Bits davon warten kann.

5.8.2 Daten-SPI Zugriff

Eine Testschrittanweisung mit einem type-Wert von 1 definiert einen Zugriff auf das DUT über die Daten-SPI. Diese in Tabelle 5.9 dargestellte Anweisung wird genutzt, um die zuvor per simulierten PSI5-Bus an das DUT gesendeten Daten auszulesen und mit einem Erwartungswert zu vergleichen und ist daher eine der zentralen Funktionen des Testsystems. Das update_mode-Feld des Befehls für einen Daten-SPI Zugriff erlaubt die Verwen-

Tabelle 5.9: Felder des JSON-Objekts der Daten-SPI

Feldname	Datentyp	Werteber.	Plicht	Funktion
update_mode	Zahl	0,1,3	Ja	Art der Werteübernahme
error_mode	Zahl	0,1,2,3	Ja	Verhalten im Fehlerfall
channel	Zahl	0,1,...,9	Ja	Auswahl PSI5 Kanal
timeslot	Zahl	0,1,...,3	Ja	Auswahl PSI5 Timeslot
reg_data	Zahl	≥0 (16 Bit)	Ja	Erwartete Daten
mask	Zahl	≥0 (16 Bit)	Ja	Maske für Daten
status	Zahl	0,1,...,3 (2 Bit)	Ja	SafeSPI Sensor Status

dung des Werts 2 (Hardware Timer) nicht, da eine Steuerung des SPI-Zugriffszeitpunkts durch den Hardware Timer des Testsystems nicht möglich ist. Als error_mode werden alle in Tabelle 5.6 definierten Werte unterstützt. Ein Daten-SPI Zugriff ist die einzige Anweisung, welche die Verwendung des Werts 2 als error_mode erlaubt. In diesem Fall werden Abweichungen in den Bits des SafeSPI Sensor Status sowie den eigentlichen Daten ignoriert, während bei einem error_mode mit dem Wert 1 nur Abweichungen in den eigentlichen Daten ignoriert werden. Über die Parameter channel und timeslot wird der Datensatz eines simulierten Sensors ausgewählt, welcher ausgelesen werden soll. Das Testsystem setzt dementsprechend automatisch einen SafeSPI-Frame zusammen, um die gewünschten Daten aus dem DUT auszulesen. Die Felder reg_data und status enthalten die

Erwartungswerte für den Datensatz eines simulierten Sensors sowie die Sensor Statusbits des SafeSPI-Frames, in welchem das Testsystem die Daten von der DUT erhält. Über das mask-Feld kann eine Maske definiert werden, über welche die Bits des erhaltenen Datensatzes mit dem Erwartungswert des reg_data-Felds verglichen werden sollen. Die Maske wirkt nur auf die Bits des reg_data-Felds und nicht auf die beiden Statusbits. Entsprechen die Daten nicht dem Erwartungswert, erzeugt die Ablaufsteuerung einen Eintrag in der Logdatei, welche die abweichenden Bits aufzeigt. Über den weiteren Testverlauf bei einer Abweichung entscheidet die Ablaufsteuerung anhand des Wertes des error_mode-Felds. Ein schreibender Zugriff über die Daten-SPI-Schnittstelle ist nicht vorgesehen.

5.8.3 Konfigurations-SPI Zugriff

Der Konfigurations-SPI Zugriff dient dem Schreiben oder Lesen von Konfigurations- und Statusregistern des DUTs. Die Anweisung eines solchen Registerzugriffs trägt im ihrem Befehlsrahmen den type-Wert 2. Im Gegensatz zu einem in Abschnitt 5.9 beschriebenen Daten-SPI Zugriff gibt es in einem SafeSPI-Frame für den Zugriff auf ein Konfigurationsregister keine Statusbits, sodass der Eintrag für den Erwartungswert der Statusbits entfällt. Dementsprechend entfällt auch die Option 2 des error_mode-Felds, da das gegebenenfalls von der Ablaufsteuerung zu ignorierende Statusbit nicht vorhanden ist. Anhand des rw-Felds der in Tabelle 5.10 dargestellten Anweisung entscheidet die Ablaufsteuerung, ob ein schreibender oder lesender Zugriff auf das Register erfolgen soll, wobei ein schreibender Zugriff durch einen Wert von 1 definiert ist und ein lesender durch eine 0. Andere Angaben in diesem Feld führen zu einem Abbruch des Testablaufs durch die Ablaufsteuerung, da sie nicht dem gültigen Wertebereich entsprechen. Die Adresse des zu lesenden oder zu

Tabelle 5.10: Felder des JSON-Objekts der Konfigurations-SPI

Feldname	Datentyp	Werteber.	Plicht	Funktion
update_mode	Zahl	0,1,3	Ja	Art der Werteübernahme
error_mode	Zahl	0,1,3	Ja	Verhalten im Fehlerfall
rw	Zahl	0,1 (Bool)	Ja	Art des Registerzugriffs
addr	Zahl	>=0 (16 Bit)	Ja	Register Adresse
reg_data	Zahl	>=0 (16 Bit)	Ja	Inhalt des Registers
mask	Zahl	>=0 (16 Bit)	Ja	Maske für Registerdaten

schreibenden DUT-Registers wird über das addr-Feld angegeben, welches Werte mit einer maximalen Breite von 16 Bit aufnimmt. Da es sich bei den Registeradressen um nicht vorzeichenbehaftete Werte handelt, muss der Wert größer als null sein. Der zu schreibende Wert des Registers, oder Erwartungswert des Registerinhalts bei einem lesenden Zugriff, wird in dem reg_data-Feld angegeben. Dieses Feld darf, wie auch die zugehörige Bitmaske im mask-Feld, maximal 16 Bit breite Werte enthalten. Die Maske dient bei ei-

nem lesenden Zugriff der Definition, welche Bits des Erwartungswertes reg_data durch die Ablaufsteuerung mit den Bits des ausgelesenen Registers verglichen werden bzw. welche Bits des Registerinhalts ignoriert werden. Bei einem schreibenden Zugriff auf das Register wird zunächst der aktuelle Registerinhalt ausgelesen und von dem ausgelesenen Wert nur die in der Maske angegebenen Bits modifiziert, bevor das Register mit dem modifizierten Wert beschrieben wird. So stellt das Testsystem sicher, dass nur die Bits eines Registers bei einem schreibenden Zugriff modifiziert werden, welche auch in der Maske angegeben wurden.

5.8.4 PSI5-Patterngenerator Konfiguration

Die Konfiguration des PSI5-Patterngenerators erfolgt über unterschiedliche JSON-Objekte bzw. Testschrittanweisungen. Das erste Objekt nimmt die globalen Einstellungen eines Kanals vor und bindet ein Array von JSON-Objekten ein, welche die Konfiguration der einzelnen Timeslots enthalten. Die Anzahl der Objekte in diesem Array ist dabei abhängig von der Anzahl der vorhandenen Timeslots eines Kanals und kann in diesem Testsystem bis zu vier betragen. Über das update_mode-Feld des äußeren JSON-Objekts, welches in Tabelle 5.11 dargestellt ist, wird definiert, wann die Statemachine des PSI5-Patterngenerators aktiv wird. Bei einem Wert von 1 wird die Statemachine direkt aktiv,

Tabelle 5.11: Felder des JSON-Objekts der Patterngenerator Konfiguration

Feldname	**Datentyp**	**Werteber.**	**Plicht**	**Funktion**
update_mode	Zahl	0,1,2,3	Ja	Art der Werteübernahme
error_mode	Zahl	0,3	Ja	Verhalten im Fehlerfall
update_time	Zahl	>0 (32 Bit)	Nein	Zeitpunkt Werteübernahme
channel	Zahl	0,1,...,9	Nein	Auswahl PSI5 Kanal
timeout	Zahl	>0 (32 Bit)	Nein	Sync-Impuls Timeout
trigger_mode	Zahl	0,1 (Bool)	Nein	Verhalten des Timer Eingangs
asyc_trigger	Zahl	0,1 (Bool)	Nein	Asynchronen Trigger auslösen
no_slots	Zahl	0,1,..,4	Nein	Anzahl PSI5 Slots
sl_config	Array		Nein	PSI5 Timeslot Definition

nachdem die Daten dieser Testschrittanweisung an den Patterngenerator übermittelt wurden. Es ist zu beachten, dass ab diesem Zeitpunkt auch der Timeout des Patterngenerators für den Erhalt eines Sync-Impulses beginnt. Der über das gleichnamige Feld timeout zu spezifizierende Timeout verhindert einen Deadlock der Statemachine des Patterngenerators, wenn kein Sync-Impuls auftritt. Soll die Statemachine möglichst lange auf einen Sync-Impuls warten, ist dieser Wert auf den Maximalwert 4294967295 zu setzen, was einer Zeitspanne von ca. 43 Sekunden entspricht.
Ist das update_mode-Feld auf 2 gesetzt, reagiert die Statemachine auf den Hardware Timer. Über den trigger_mode-Parameter kann bestimmt werden, welche Aktion dieser in ei-

nem solchen Fall auslöst. Ist dieses Feld auf 0 gesetzt, wird die Statemachine aktiv, erzeugt einen PSI5-Frame und wartet auf einen Sync-Impuls. Ist der trigger_mode-Parameter 1, wertet die Statemachine das Signal des Hardware Timers als einen asynchronen Triggerimpuls und versendet daraufhin einen PSI5-Frame. Dieser muss zuvor über eine weitere Testschrittanweisung erzeugt worden sein. Siehe dazu auch Abschnitt 4.3.1. Der Zeitpunkt zu welchem der Timer die Aktion auslöst, wird in beiden Fällen über das update_time-Feld eingestellt. Dieses Feld akzeptiert einen 32 *Bit* nicht vorzeichenbehafteten Wert, welcher die Zeit zwischen der Auslösung der Aktion und dem Start des Hardware Timers in Mikrosekunden definiert. Der Kanal, für welchen die aktuellen Einstellungen gelten, wird über den Parameter channel bestimmt. Dabei ist zu beachten, dass die Einstellungen bezüglich des Hardware Timers immer für alle zehn Kanäle des Testsystems gelten und bei mehreren abweichenden Einstellungen immer die letzten Werte gültig sind. Die Einstellungen der Timeslots werden in einem Array gespeichert, welches unter dem Schlüssel sl_config in dem äußeren JSON-Objekt hinterlegt wird. Diese Verschachtelung verbessert die Übersichtlichkeit der Darstellung sowohl der globalen Einstellungen als auch der Timeslots und bietet die Möglichkeit zukünftig die Anzahl der Timeslots, welche vom Testsystem unterstützt werden, zu erhöhen. Die Anzahl der in der Konfiguration eines Kanals vorhandenen Timeslots wird über den Parameter no_slots angegebenen und muss mit der Anzahl der Einträge in dem Array übereinstimmen. Eine fehlerhafte Angabe führt zu einem Abbruch des Testdurchlaufs, da der Parser der Ablaufsteuerung die Einträge des Arrays nicht zuordnen kann.
Die Definition eines Timeslots besteht aus acht Parametern, welche allesamt Pflichtangaben des JSON-Objektes sind (vgl. Tabelle 5.12). Das enabled-Feld bietet die Möglichkeit

Tabelle 5.12: Felder des JSON-Objekts der Timeslot Konfiguration

Feldname	**Datentyp**	**Werteber.**	**Plicht**	**Funktion**
enabled	Zahl	0,1 (Bool)	Ja	Timeslot aktiv
crc_mode	Zahl	0,1 (Bool)	Ja	Auswahl Prüfsummenart
crc_error	Zahl	0,1 (Bool)	Ja	CRC Fehlereinspeisung
manchester_error	Zahl	0,1 (Bool)	Ja	Fehlereinsp. M.Kodierung
clkdiv	Zahl	>0 (32 Bit)	Ja	Teilfaktor PSI5 Bittakt
startpoint	Zahl	>0 (32 Bit)	Ja	Timeslot Startzeitpunkt
payload	Zahl	>0 (28 Bit)	Ja	PSI5 Nutzdaten
payload_length	Zahl	0,1,...,28	Ja	PSI5 Nutzdatenlänge

einzelne Timeslots zu aktivieren oder zu deaktivieren. Ist ein Timeslot deaktiviert, wird er von der Statemachine während eines Übertragungszyklusses übergangen. Mittels des crc_mode-Parameters wird das verwendete Verfahren zur Generierung einer Prüfsumme für den in diesem JSON-Objekt definierten Timeslot ausgewählt. Ist der crc_mode-Parameter 1 so wird die 3-Bit CRC-Prüfsumme verwendet. Andernfalls kommt ein einfaches Paritätsbit zum Einsatz. Die Felder crc_error und manchester_error können gesetzt

werden, um einen Prüfsummenfehler oder einen Fehler in der Manchesterkodierung des Frames einzuspeisen. Mittels des Parameters clkdiv kann der Bittakt der Datenübertragung dieses Timeslots variiert werden. Dazu wird in dem clkdiv-Feld ein Teilfaktor angegeben, welcher aus dem Haupttakt des Testsystems den Bittakt der PSI5-Übertragung erzeugt. Sieh dazu Abschnitt 4.3.5. Die Einstellung des Startzeitpunkts des Timslots erfolgt über den Wert startpoint. Bei der Festlegung dieses Wertes muss der Anwender sicherstellen, dass, sofern mehrere Timeslots vorhanden sind, diese sich nicht überlappen. Die eigentliche Payload, die während eines Timeslots von dem Sensorsimulator zum DUT übertragen werden soll, wird in dem gleichnamigen Feld payload untergebracht. Die Payload muss immer die niederwertigsten Bits des Felds belegen, sodass auch die numerische Repräsentation der Payload im JSON-Format den kleinstmöglichen Wert annimmt. Damit das Testsystem Informationen über die Anzahl der gültigen Bits in diesem Feld, welche zur Payload gehören, besitzt, existiert ein zusätzliches Feld payload_length in welchem die Länge der Payload angegeben wird.

5.8.5 DAC Konfiguration

Die Konfiguration der Digital-To-Analog Converter (DAC) des Sensorsimulators erfolgt getrennt von der Konfiguration des Patterngenerators in einer eigenen Anweisung, dessen type-Wert 4 ist. Wie in Tabelle 5.13 ersichtlich ist werden sowohl der Ruhestrom als auch der Pulsstrom des PSI5-Sensorsimulators eines Kanals gemeinsam in einem Testschritt konfiguriert. Dazu muss über den Parameter channel der Kanal des PSI5-Sensorsimulators

Tabelle 5.13: Felder des JSON-Objekts der DAC Konfiguration

Feldname	**Datentyp**	**Werteber.**	**Plicht**	**Funktion**
update_mode	Zahl	0,1,2,3	Ja	Art der Werteübernahme
error_mode	Zahl	0,1,3	Ja	Verhalten im Fehlerfall
update_time	Zahl	>0 (uint32)	Nein	Zeitpunkt Werteübernahme
channel	Zahl	0,1,...,9	Ja	Auswahl Kanal Sensorsimulator
quiescent	Zahl	0,1,...,50	Ja	PSI5 Ruhestrom in mA
pulse	Zahl	0,1,...,50	Ja	PSI5 Pulsstrom in mA

spezifiziert werden, für welchen diese neuen Werte gelten sollen. Die Angabe der Werte für den Ruhestrom und den Pulsstrom erfolgt in mA in den Feldern quiescent für den Ruhestrom und pulse für den Impulsstrom einer PSI5-Datenübertragung. Der gültige Wertebereich für die Ströme ist $0-50\,mA$, wobei zu beachten ist, dass der Impulsstrom auf den Ruhestrom addiert wird und es bei dem Wert des Impulsstroms nicht um eine Gesamtstromangabe während eines Impulses handelt. Die Angaben der Ströme werden bereits während des Parsens dieser Anweisungen in die passenden Bitwerte der DACs umgerechnet, sodass die Ablaufsteuerung nur die Bitwerte der DACs intern speichern muss.

Sollen mehrere Kanäle des Testsystems ihre Werte simultan ändern, müssen zunächst Testschrittanweisungen definiert werden, welche neue Daten für den Ruhestrom und den Pulsstrom des PSI5-Sensorsimulators enthalten aber die Übernahme dieser Daten in die eigentlichen DACs nicht auslösen. Die Testschrittanweisung für den letzten Kanal, welcher seinen Wert simultan mit den anderen ändern soll, enthält dann einen Wert des update_mode-Felds, welcher die Wandlung durch die DACs auslöst. Alternativ kann auch mit dem letzten Testschritt das update_mode-Feld auf 2 gesetzt werden und im update_time -Feld eine relative Zeit zum Startzeitpunkt des Hardware Timers in Mikrosekunden angegeben werden, zu welcher der Hardware Timer die Wandlung auslöst.

5.8.6 Konfiguration des Triggerausgangs

Der Triggerausgang wird über die Testschrittanweisung konfiguriert, welche in ihrem JSON-Objekt den type-Wert mit 5 belegt hat. Der Triggerausgang kann zur Synchronisation von externen Instrumenten über den BNC-Anschluss des Testsystems genutzt werden. Eine Liste aller Felder des JSON-Objekts zur Konfiguration des Triggerausgangs befindet sich in Tabelle 5.14. Eine Veränderung dieser Konfiguration des Triggerausgangs zu einem

Tabelle 5.14: Felder des JSON-Objekts der Ausgangssignalkonfiguration

Feldname	Datentyp	Werteber.	Plicht	Funktion
update_mode	Zahl	0,1,3	Ja	Art der Werteübernahme
error_mode	Zahl	0,3	Ja	Verhalten im Fehlerfall
enabled	Zahl	0,1 (Bool)	Ja	Triggerausgang aktiv
output_mode	Zahl	0,1 (Bool)	Ja	Ausgangsignaltyp
pulse_duration	Zahl	>0 (uint32)	Ja	Impulslänge
source	Zahl	0,1 (Bool)	Ja	Signalquelle
sync_source	Zahl	0,1,...,1023	Ja	PSI5-Kanal Signalquelle

durch den Hardware Timer vorgegebenen Zeitpunkt ist nicht möglich. Das update_mode-Feld unterstützt daher nicht den Wert 2. Über das source-Feld kann die interne Signalquelle ausgewählt werden, welche an den BNC-Ausgangs des Testsystems weitergeleitet wird. Ist das source-Feld auf 1 gesetzt, ist die interne Signalquelle ein Vergleichsausgang des Hardware Timers. Ist der source-Wert 0, wird der Sync-Impuls der Detektorschaltung einer oder mehrerer PSI5-Kanäle an den BNC-Ausgang angelegt. Für weitere Details siehe Abschnitt 4.6. Die Parameter output_mode und pulse_duration bestimmen, wie das Ausgangssignal aussieht. Ist der Parameter output_mode mit 0 belegt, entspricht das Ausgangssignal am BNC-Stecker des Testsystems genau der gewählten internen Signalquelle. Da dieses Impulssignal in seiner Dauer zu kurz sein kann, wenn der Hardware Timer als interne Signalquelle verwendet wird, existiert ein Ausgangsmodus mit einer festen Signallänge. Diese lässt sich über eine 1 im Feld des Parameters output_mode auswählen und

erzeugt einen Impuls, welcher die in dem Feld pulse_duration angegebene Anzahl an Takten der Haupttaktquelle des Testsystems ($100\,MHz$) entspricht. Ist der Modus mit der einfachen Weiterleitung der internen Signalquelle gewählt, kann dieses Feld durch einen beliebigen Wert als Platzhalter gefüllt werden.
Über das Feld sync_source wird definiert, welche der PSI5-Kanäle des Sensorsimulators als Quelle für ein Sync-Impuls-Signal herangezogen werden. Jedes Bit dieses Wertes steht für einen Kanal, der aktiviert wird. Der Wert ist dementsprechend $10\,Bit$ breit und kann maximal eine numerische Repräsentation im Bereich zwischen 0 und 1023 erreichen. Sind mehrere dieser Bits gesetzt, werden die Signale der Sync-Impuls-Detektion in einer Oder-Verknüpfung zusammengefügt, sodass jeder aktivierte Kanal das Ausgangstriggersignal erzeugen kann. Sind die Komparatoren der Sensorsimulatoren nicht als Quelle des Ausgangstriggersignals ausgewählt, kann dieser Wert mit einem beliebigen Platzhalter gefüllt werden, da dessen Einstellungen von dem IP-Block in diesem Fall ignoriert werden.

5.8.7 Kurzschlusseinspeisung

Die Konfiguration sämtlicher möglicher Kurzschlussoptionen durch das Testsystem erfolgt, im Gegensatz zur Konfiguration des PSI5-Patterngenerators oder der DACs, über eine einzelne Testanweisung. Diese Testanweisung ist in Tabelle 5.15 dargestellt und verfügt über vier Felder für die vier verschiedenen Kurzschlusseinspeisungen, welche jedem Kanal des Testsystems zur Verfügung stehen. Soll die Änderung der aktiven Kurzschlüs-

Tabelle 5.15: Felder des JSON-Objekts der Fehlereinspeisung

Feldname	**Datentyp**	**Werteber.**	**Plicht**	**Funktion**
update_mode	Zahl	0,1,2,3	Ja	Art der Werteübernahme
error_mode	Zahl	0,3	Ja	Verhalten im Fehlerfall
update_time	Zahl	>0 (uint32)	Nein	Zeitpunkt Werteübernahme
short_gnd	Zahl	0,1,...,1023	Ja	Auswahl Kurzschlüsse Masse
short_vbat	Zahl	0,1,...,1023	Ja	Auswahl Kurzschlüsse VBAT
short_bus1	Zahl	0,1,...,1023	Ja	Konfiguration Kurzschlussbus 1
short_bus2	Zahl	0,1,...,1023	Ja	Konfiguration Kurzschlussbus 2

se zu einem durch den Hardware Timer bestimmten Zeitpunkt erfolgen, so muss neben dem Wert des error_mode-Felds von 2 im update_time-Feld der Zeitpunkt angegeben werden. Die Angabe erfolgt dabei relativ in Mikrosekunden, welche seit dem Startzeitpunkt des Hardware Timers vergangen sind. Die Definition, welche der unterschiedlichen Kurzschlusseinspeisungen des Testsystems aktiv sind, wird über die vier Parameter short_gnd, short_vcc,short_bus1 und short_bus2 gesteuert. Das Feld short_gnd aktiviert Kurzschlüsse eines PSI5-Kanals gegen Masse. Über das short_vcc-Feld können Kurzschlüsse gegen die positive Versorgungsspannung VBAT eingeschaltet werden und die beiden Parameter

short_bus1 und short_bus2 konfigurieren die Verbindung der PSI5-Kanäle mit den beiden Kurzschlussbussen. Für weitere Informationen siehe Abschnitt 4.4. Jedes dieser vier Felder nimmt einen numerischen Wert auf, welcher einer nicht vorzeichenbehafteten Zahl mit 10 *Bit* entspricht. Jedes dieser Bits entspricht einem Kanal, wobei das niederwertigste Bit 0 dem Kanal 1 und das höchstwertigste Bit 9 dem Kanal 10 entspricht. Ist ein Bit eines dieser Feld gesetzt, ist der entsprechende Kurzschluss des gewählten Kanals aktiv. Fehlerhafte Mehrfachbesetzungen, welche zu einem direkten Kurzschluss zwischen der positiven Versorgungsspannung VBAT und der Masse führen, werden durch den IP-Block selbst ausgeschlossen (vgl. Abschnitt 4.4).

5.8.8 Timer Konfiguration

Der Hardware Timer erzeugt Steuersignale, welche von anderen IP-Blöcken zur Auslösung von Aktionen herangezogen werden. Damit der Hardware Timer diese Aufgabe erfüllen kann, muss er jedoch zunächst durch eine eigene Testschrittanweisung konfiguriert werden. Sämtliche Felder des JSON-Objekts zur Konfiguration des Hardware Timers sind, wie in Tabelle 5.16 dargestellt, Pflichtfelder. Da der Hardware Timer selbst eine Quelle

Tabelle 5.16: Felder des JSON-Objekts der Hardware Timerkonfiguration

Feldname	**Datentyp**	**Werteber.**	**Plicht**	**Funktion**
error_mode	Zahl	0,3	Ja	Verhalten im Fehlerfall
enabled	Zahl	0,1 (Bool)	Ja	Aktivierung des Timers
reset	Zahl	0,1 (Bool)	Ja	Rücksetzen des Timers
preload	Zahl	0,1 (Bool)	Ja	Timer Vorladen
preload_value	Zahl	>0 (32 Bit)	Ja	Vorladewert

zur Definition von Zeitpunkten zur Auslösung von Aktionen anderer IP-Blöcke ist, verfügt die Konfigurationsstruktur des Timers nicht über einen update_mode-Eintrag. Über den enabled-Parameter wird der Timer aktiviert und deaktiviert. Entspricht der Wert des enabled-Parameters 1, beginnt der Timer seinen Zählstand periodisch zu inkrementieren. Analog dazu wird bei einem Wert von 0 der Timer deaktiviert. Eine automatische Rücksetzung des aktuellen Zählstands bei der Deaktivierung des Timers findet nicht statt, sodass der Timer über das enabled-Feld angehalten und zu einem späteren Zeitpunkt fortgesetzt werden kann. Ist das reset-Feld gesetzt, wird der Timer zurückgesetzt bevor irgendeine von den weiteren potentiellen Einstellungen des Timers vorgenommen wird. Der Timer bietet die Möglichkeit, dass dieser vor dem Start auf einen vordefinierten Wert initialisiert wird. Um diese Funktion zu aktivieren, muss der preload-Parameter auf 1 gesetzt werden und der gewünschte Initialwert des Hardware Timers in dem Feld preload_value angegeben werden. Der interne Zähler des Timers ist 32 *Bit* breit. Der Initialwert des Timers besteht daher auch aus einer nicht vorzeichenbehafteten Zahl mit maximal 32 *Bit* Breite. Wird

die Funktion zum Vorladen des Timers nicht verwendet, kann das preload_value-Feld mit einem Platzhalter belegt werden, welcher aus einem beliebigen Wert des zulässigen Wertebereichs besteht. Die Vergleichswerte, zu welchen Zeitpunkten der Timer Aktionen in den anderen IP-Blöcken ausführen soll, werden über die Testschrittanweisungen besagter IP-Blöcke durchgeführt.

5.9 Webserver

Die Bedienung des Testsystems erfolgt über ein Webinterface, sodass das Testsystem einerseits keine weitere Software auf dem PC des Anwenders zur Bedienung des Testsystems benötigt und andererseits, aufgrund der Netzwerkverbindung, sich nicht direkt am Arbeitsplatz des Anwenders befinden muss. Das Webinterface des Testsystems wird von einem eigenständigen Apache Webserver ausgeliefert und ist nicht Bestandteil der Anwendung zur Ablaufsteuerung des Testsystems. Die Kommunikation zwischen der Ablaufsteuerung und dem Webserver erfolgt durch Argumente beim Aufruf der Anwendung und die Log- und Statusdateien. Der Aufbau des Webinterfaces passt sich dynamisch an die Auflösung des Browsers an und auch der Inhalt der einzelnen Webseiten wird dynamisch gefüllt. Für die dynamischen Aspekte der Webseite werden serverseitig PHP und browserseitig Javascript eingesetzt. Als Grundlage für das Design des Webinterfaces dient das freie Bootstrap Framework, welches HTML und CSS Vorlagen bereitstellt. Diese werden vor allem für die dynamische Generierung des Layouts, der Tabellen und die Menüführung benötigt. Das Bootstrap Framework wird zusammen mit allen weiteren benötigten Javascript-Dateien vom Apache Webserver selbst ausgeliefert. Viele moderne Webseiten sind dazu übergegangen solche Bestandteile von anderen externen Domains nachzuladen. Damit das Testsystem jedoch ohne einen Internetzugriff funktioniert, liefert der Webserver alle benötigten Dateien selbst aus.
Die Hauptseite des Webinterfaces (index.html) verfügt über keine Funktionen, sondern dient nur als Einstiegsseite beim erstmaligen Aufruf durch den Webbrowser. Über die Menüleiste im oberen Bereich des Webinterfaces können die Unterseiten, welche die Funktionen des Testsystems enthalten, aufgerufen werden. Die drei Unterseiten erlauben das Laden von JSON-Testfiles in das Testsystem, das Ausführen eines Test und das Betrachten der erzeugten Logdateien.
Das in Screenshot 5.11 abgebildete Webinterface zum Hochladen von JSON-Testfiles vom PC des Anwenders in das Testsystem stellt eine einfach zu bedienende Alternative zu einem Datentransfer über SSH / SFTP dar, welcher einen zusätzlichen Client auf dem PC des Anwenders benötigen würde. Auf der rechten Seite der Webseite zum Hochladen von Testdateien befindet sich eine Tabelle, welche alle bereits im Testsystem vorhandenen Dateien inklusive deren Bearbeitungsdatum auflistet. Diese Liste ist dafür gedacht,

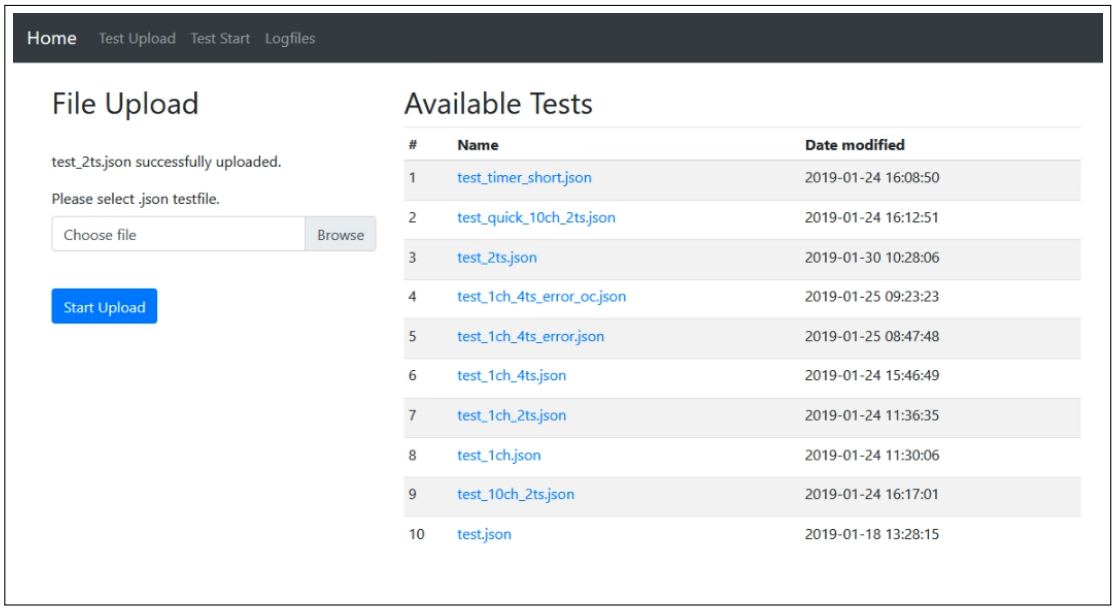


Abb. 5.11: Webinterface zum Datenupload

die Übersichtlichkeit beim Hochladen von neuen Tests zu verbessern und zu verhindern, dass vom Anwender versehentlich Tests mit dem selben Dateinamen hochgeladen werden. Wird eine Datei mit dem selben Dateinamen hochgeladen, so wird die bestehende Datei überschrieben um Aktualisierungen dieser zu erlauben. Die Tabelle mit den vorhandenen Tests wird durch PHP erzeugt. Zunächst wird über die PHP scandir-Funktion eine Variable mit den Informationen aller Dateien im Verzeichnis /var/www/html/test_files/, in welchem sich die Testfiles befinden, gefüllt. Diese Informationen werden dann durch ein weiteres PHP-Skript an der passenden Stelle im HTML-Quellcode der Tabelle eingefügt, sodass die Tabelle bei jedem Seitenaufruf dynamisch generiert wird. Jeder Eintrag in der Übersichtstabelle ist mit einem Link zu der JSON-Datei im Verzeichnis /var/www/html/test_files/ versehen, über welchen selbige aufgerufen werden kann. Webbrowser wie Firefox verfügen über einen JSON-Parser, welcher den Inhalt der JSON-Datei grafisch aufbereitet darstellt.

Über ein Auswahlfeld kann der Anwender die Datei, welche auf das Testsystem geladen werden soll, von seinem PC wählen. Der Browser nutzt zum Upload der ausgewählten Datei die HTTP-Post-Methode, welche von dem Webserver empfangen wird und an den PHP-Prozessor weitergeleitet wird. Anhand eines kleinen Skripts bestimmt dieser was mit der empfangenen Datei geschehen soll. Überschreitet die Datei eine gewisse Größe nicht und hat sie .json als Endung, wird die Datei aus dem temporären Speicherort in das Verzeichnis /var/www/html/test_files/ zu den anderen Testfiles verschoben. Ist der Datenupload abgeschlossen oder fehlgeschlagen, wird dies dem Anwender durch eine entsprechende Meldung oberhalb des Auswahlfeldes für das Testfile mitgeteilt.

Über den Text Start-Menüpunkt kommt der Anwender auf die Oberfläche zur Auswahl und zum Start eines Tests. Diese Webseite hat, wie auch das Webinterface zum Hochladen

der Tests, eine durch die PHP scandir-Funktion generierte Liste aller verfügbaren Tests. Zusätzlich befindet sich, wie in Screenshot 5.12 zu erkennen, neben jedem Eintrag nun ein Radio Button, mittels welchem der auszuführende Test vom Anwender gewählt werden kann. Neben dem Test muss dieser noch spezifizieren, ob er die Debug-Ausgaben in der Log-Datei erhalten möchte und wie die Log-Datei benannt werden soll. Das Auswahlfeld

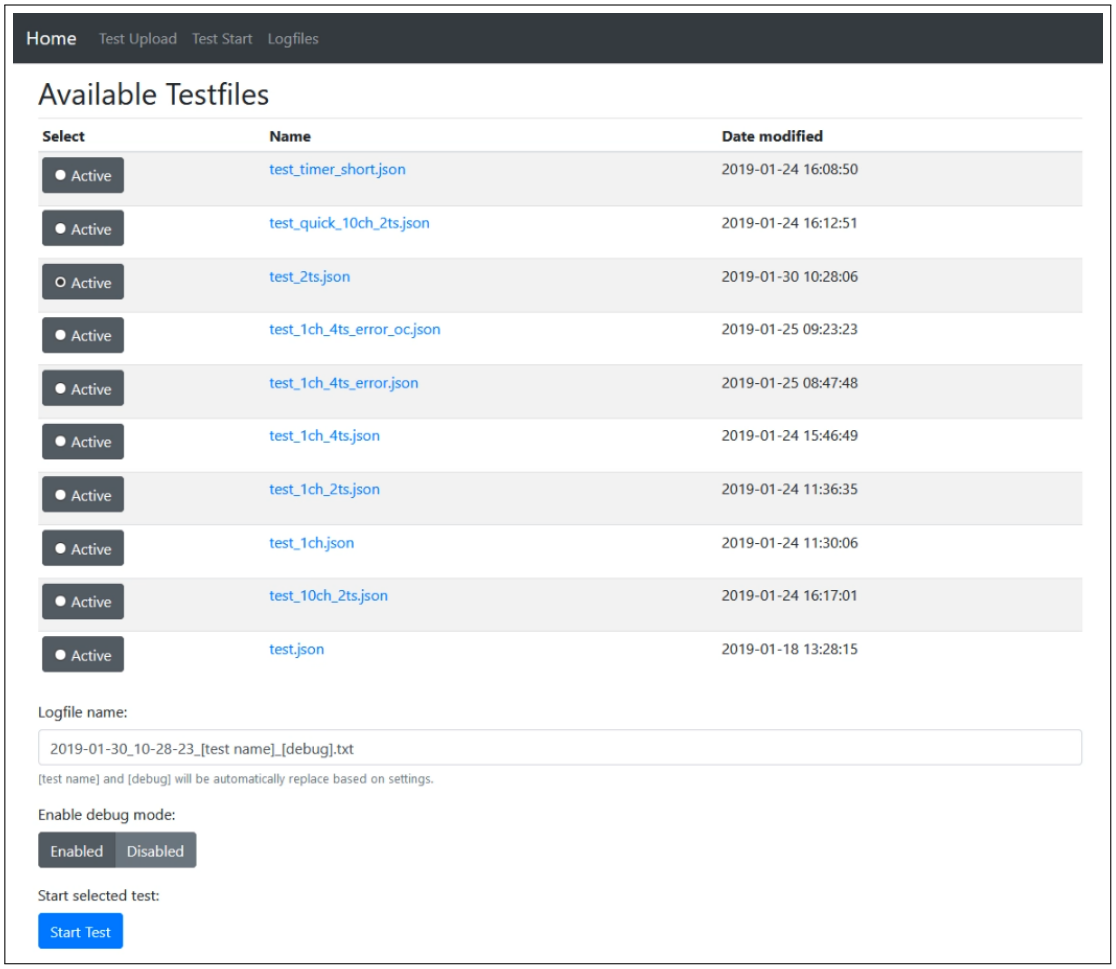


Abb. 5.12: Webinterface zum Starten eines Tests

wird beim Aufrufen der Webseite bereits dynamisch durch ein PHP-Skript mit einem Vorschlag für den Dateinamen gefüllt, welcher das aktuelle Datum und die Uhrzeit, sowie zwei Platzhalter enthält. Die beiden Platzhalter [test name] und [debug] werden beim Start des Tests durch ein weiteres PHP-Skript mit dem Dateinamen des ausgewählten Tests und gegebenenfalls dem Wort Debug, wenn die Debugausgaben aktiviert wurde, gefüllt. Die beiden Platzhalter können auch in jeden anderen frei gewählten Dateinamen für die Log-Datei eingesetzt werden, da das PHP-Skript nur eine einfache 'Suche und Ersetze'-Funktion zum Füllen der Platzhalter verwendet.
Hat der Anwender einen Test gestartet, werden die gewählten Parameter, wie die Dateinamen der Log- und Testdatei von einem PHP-Skript ausgewertet und dementsprechend die Argumente für den Aufruf der Anwendung zur Ablaufsteuerung erzeugt. Diese,

in Abschnitt 5.7 beschriebenen Argumente beim Aufruf der Anwendung, ist die einzige Kommunikation des Webinterfaces in Richtung der Ablaufsteuerung. Ein anderer Eingriff in die Ablaufsteuerung während deren Betriebs ist durch das Webinterface nicht vorgesehen. Der eigentliche Aufruf der Anwendung zur Ablaufsteuerung kann nicht direkt durch die PHP exec-Funktion erfolgen, da die Anwendung mit Root-Rechten ausgeführt werden muss, der PHP-Parser und Webserver jedoch aus Sicherheitsgründen nicht als Root ausgeführt werden. Die exec-Funktion des Skripts ruft daher den Befehl sudo auf, welcher erlaubt Anwendungen mit den Rechten eines anderen Benutzers, in diesem Fall des Root-Benutzers, auszuführen. Die Konfigurationsdatei des sudo-Befehls wurde daher so erweitert, dass nur für die Anwendung zur Ablaufsteuerung eine Ausnahme eingefügt wurde, welche es erlaubt, dass die Anwendung mit Rootrechten von Benutzern ausgeführt wird, welche sonst keine Rootrechte besitzen. Die Rechtestruktur zur Ausführung aller anderen Anwendungen des Linux bleibt davon unberührt. Zur weiteren Verbesserung der Sicherheit kann der Zugriff auf das Webinterface eingeschränkt oder mit einem Passwort versehen werden. Dazu müssen die .htaccess-Konfigurationsdateien des Apache Webservers angepasst werden.
Wurde die Anwendung zur Ablaufsteuerung gestartet und der Test begonnen, wird über ein Javascript der Browser zu einer Statusseite weitergeleitet, mit welcher der Ablauf des Tests betrachtet werden kann. Die Statusseite liest die Statusdatei und die Logdatei, welche von der Ablaufsteuerung erzeugt werden, aus, und stellt diese grafisch aufgearbeitet dar. Aus der Statusdatei, welche von der Ablaufsteuerung nach jedem Testschritt aktualisiert wird, wird der Name des Tests, die Nummer des gegenwärtigen Testschritts, dessen Name und die Gesamtzahl an Testschritten in dem Test ausgelesen. Der Name des Tests dient dem Interface dabei, wie in Screenshot 5.13 zu erkennen ist, als Überschrift und verändert sich während eines Tests nicht. Die Unterüberschrift der Statusseite des Webinterfaces wird über ein Javascript alle $500\,ms$ aktualisiert, sodass sie dynamisch die Bezeichnung des gegenwärtigen Testschritts anzeigt und wie viele Testschritte des Testablaufs bereits abgeschlossen sind. Über diese Angaben kann der Anwender leicht erkennen, wie weit fortgeschritten ein Test ist, oder ob dieser in einem Testschritt steckengeblieben ist. Ebenfalls alle $500\,ms$ wird die Tabelle im unteren Bereich der Statusseite des Webinterfaces aktualisiert, welche den Inhalt der Logdatei anzeigt. Die Tabelle wird bei einer Aktualisierung dynamisch erweitert, sollte ein weiterer Testschritt mit Ausgaben für die Logdatei fertigstellt worden sein. Jede Zeile der Tabelle enthält die Informationen über einen Testschritt und kann aus mehreren Textzeilen bestehen. Enthält der Eintrag der Logdatei für einen Testschritt das Wort Error, wird die entsprechende Zeile der Tabelle durch das Javascript rot eingefärbt, um die Erkennung der Fehler zu verbessern.
Die grafische Aufbereitung der Logdateien kann der Anwender sich auch zu einem späteren Zeitpunkt erneut angucken. Das Webinterface verfügt dazu über die in Screenshot 5.14 dargestellte Übersichtsseite aller bisherigen Logdateien, welche über den Menüpunkt Logfi-

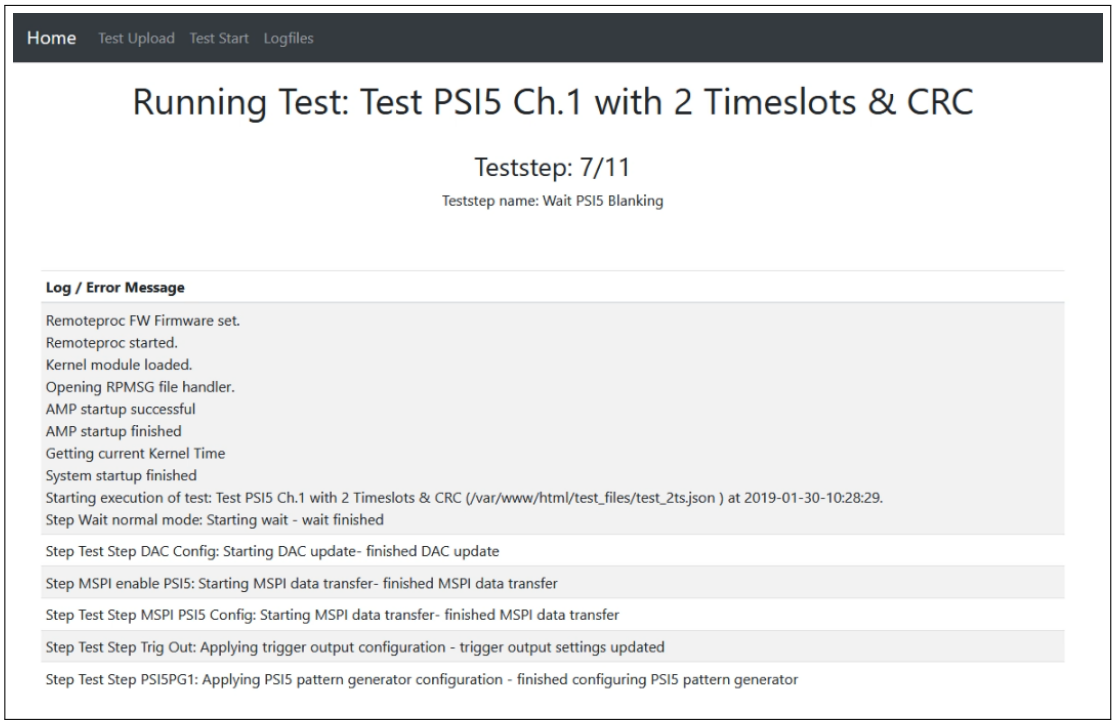


Abb. 5.13: Statusinformationen während eines Testdurchlaufs

les zu erreichen ist. Zur Erzeugung der Tabelle mit Einträgen zu allen Logdateien wird das bereits bekannte PHP-Skript mit der scandir-Funktion eingesetzt. Jeder Eintrag einer Logdatei in dieser Tabelle kann auf zwei Arten betrachtet werden. Durch einen Klick auf den linken Link eines Eintrags wird der User auf eine Seite des Webinterfaces weitergeleitet, welche die grafische Aufbereitung der Logdatei, die bereits während der Testausführung zu sehen war, erneut zeigt. Alternativ kann die Logdatei auch als unveränderte .txt-Datei heruntergeladen und mit einem externen Texteditor betrachtet werden. Neben dem Betrachten und Herunterladen der Logdateien ist es möglich diese über das Webinterface zu löschen. Die Übersichtsseite besitzt dazu neben jeder Logdatei eine Checkbox, mittels welcher eine oder mehrere Dateien für die Löschung markiert werden können. Ein PHP-Skript wertet diese Angaben serverseitig aus und löscht dementsprechend die Datei aus dem Verzeichnis /var/www/html/log_files/. Eine Wiederherstellung gelöschter Dateien ist nicht vorgesehen.

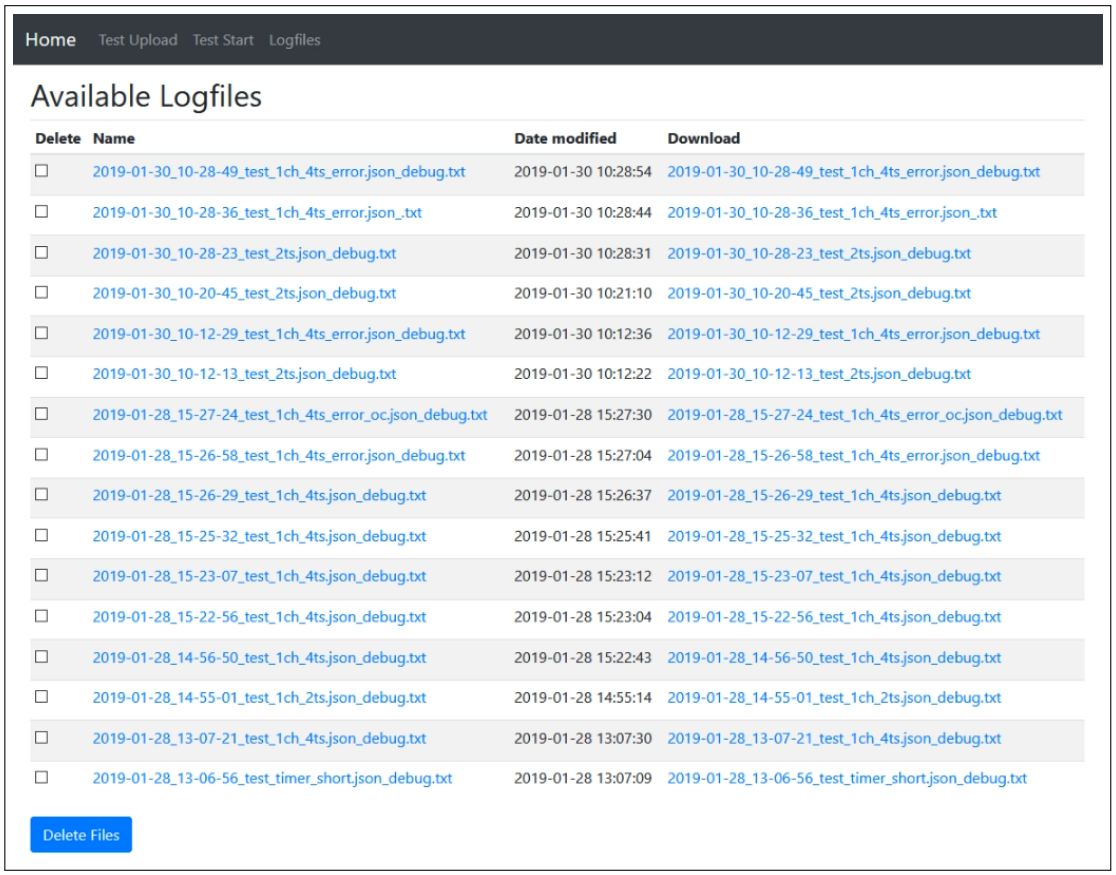


Abb. 5.14: Webinterface zur Übersicht über die Logdateien

6 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Testsystem für den Physical- und Data-Link-Layer des PSI5-Busses entwickelt. Dazu wurde zunächst das PSI5-Protokoll und sein Einsatz in den Insassenrückhaltesystemen sowie weiteren Systemen von Kraftfahrzeugen beleuchtet. Aus dem Physical- und Data-Link-Layer des PSI5-Standards wurden die Parameter des Protokolls abgeleitet, welches das Testsystem zur Simulation eines Sensors in der Lage sein muss zu reproduzieren. Außerdem wurden die möglichen Störgrößen, welche in einem Kraftfahrzeug auftreten können, wie z.B. ein Kurzschluss durch eine Beschädigung des Kabelbaums definiert. Diese bilden weitere Anforderungen für den Entwicklungsprozess des Testsystems, da dieses eine Simulation besagter Störungen durchführen können muss. Die Fehlereinspeisung zur Simulation von Störungen dient dabei nicht nur der Untersuchung des Verhaltens der PSI5-Schnittstelle eines ASICs, sondern auch der Aufdeckung von möglichen Wechselwirkungen innerhalb dessen. Das entwickelte Testsystem schließt dabei die Lücke zwischen den bestehenden Systemen, welche zur Prüfung der PSI5-Schnittstellen eingesetzt werden. Bisher wurden nur automatisierte Testmaschinen genutzt, welche einen großen Funktionsumfang besitzen, jedoch nicht portabel sind, sowie einzelne Testplatinen, welche bei einer häufigen Testwiederholung aufwändig zu bedienen sind und oftmals nur einen Aspekt der PSI5-Schnittstelle untersuchen können. Das neue Testsystem ist aufgrund seiner geringen Abmessungen und geringen Anforderungen an die Spannungsversorgung portabel und verfügt über eine Ablaufsteuerung um Regressionstest wiederholt ohne Eingriff des Anwenders ausführen zu können.
Aufgrund der Vielzahl an verschiedenen Anforderungen an das Testsystem, hat sich eine Struktur ergeben, welche aus einer programmierbaren Logik und zwei CPU-Kernen besteht, die unterschiedliche Software ausführen. Damit diese einzelnen Komponenten optimal miteinander integriert sind, wurde ein Xilinx ZYNQ SoC als Kernstück dieses Testsystems verwendet, da dieser eine programmierbare Logik und zwei ARM Cortex A9 CPU-Kerne mit ausreichend Rechenleistung für das Testsystem in einem SoC vereint. Die Funktionen des Testsystems, welche einen hohen Grad an Parallelisierung benötigen oder sogar simultan ausgeführt werden müssen, wie die Patterngeneratoren der einzelnen simulierten PSI5-Kanäle, wurden als IP-Blöcke in der programmierbaren Logik des Testsystems umgesetzt. Der Großteil der Funktionsblöcke der programmierbaren Logik wurde dabei in SystemVerilog geschrieben, da diese Hardwarebeschreibungssprache ein erweitertes parametrisiertes Design zulässt, wodurch die Blöcke mit einer Änderung eines einzelnen Parameters in der Anzahl der unterstützten PSI5-Kanäle angepasst werden können.
Die sequenziell auszuführenden Komponenten des Testsystem sind in Software umgesetzt worden. Die Softwareaufgaben wurden dabei anhand ihrer Randbedingungen in timing-

kritische und timing-unkritische Aufgaben gruppiert. Die timing-kritischen werden von dem Echtzeitbetriebssystem FreeRTOS ausgeführt, da dieses maximale Ausführungsdauern von Aufgaben garantieren kann. Alle anderen Aufgaben werden von einem Linux ausgeführt, welches aus einem Petalinux-Kernel von Xilinx und einem Debian Rootdateisystem besteht. Die beiden CPU-Kerne mit ihren unterschiedlichen Betriebssystemen bilden ein asymmetrisches Multiprozessorsystem, welches über einen rpmsg-Bus miteinander kommuniziert. Die Kommunikation zwischen den Softwarekomponenten und den IP-Blöcken der programmierbaren Logik, wurde durch die Verwendung von AXI-Interfaces in den IP-Blöcken und dem Einsatz des zugehörigen AXI-Busses implementiert.
Der Ablauf eines Tests der PSI5-Schnittstelle des zu untersuchenden DUTs wird durch den Anwender in einer Datei definiert, welche über ein Webinterface in das Testsystem geladen wird. Die Struktur dieser Testdefinitionen ist dabei so aufgebaut, dass das Testsystem in einfacher Weise an neue Aufgaben angepasst werden kann. Die Testdatei nutzt das JSON-Format, welches menschenlesbar ist und durch die Verschachtelung der einzelnen Strukturen und Testanweisungen eine gute Übersicht über die Testschritte bietet. Die Bedienung des Testsystems erfolgt über ein Webinterface, sodass der Anwender keine spezielle Software auf seinem PC zur Inbetriebnahme des Testsystems benötigt. Das Testsystem erfüllt damit eines der wichtigen Kriterien, um eine möglichst portable und unabhängige Testlösung darzustellen zu können.

6.1 Ausblick

Das Testsystem bietet durch seinen modularen Aufbau der Software und der programmierbaren Logik die Möglichkeit eine Vielzahl von Eigenschaften des Testsystems zu modifizieren. Die programmierbare Logik kann durch die parametrisierte Definition der Anzahl der Kanäle vergleichsweise einfach um weitere PSI5-Kanäle erweitert werden, sodass das Testsystem an DUTs mit mehr als zehn Kanälen angepasst oder eine kleinere Version mit weniger PSI5-Kanälen erstellt werden kann. Welche Änderungen oder Erweiterungen des Testsystems eine sinnvolle Verbesserung bedeuten, ergibt sich erst im Laufe des breiteren Einsatz des Testsystems in der Entwicklung und Verifikation von verschieden ASICs.
Eine der möglichen zukünftigen Erweiterungen betrifft nicht die Kernfunktion des Testsystems, sondern dessen Bedienung. In der zum Zeitpunkt dieser Arbeit aktuellen Version der Software des Testsystems müssen die Testabläufe zunächst manuell in einem Texteditor definiert werden. Die mögliche Erweiterung sieht den Ausbau des Webinterfaces um einen grafischen Editor vor, mittels welchem die Definition der Testabläufe vorgenommen werden kann. Insbesondere im Bezug auf die Timingparameter der einzelnen Timeslots einer PSI5-Datenübertragung bietet ein grafischer Editor einen deutlich besseren Überblick auf die entstehende Testdefinition. Die benötigten Strukturen für die Implementierung einer solchen Erweiterung des Webinterfaces sind bereits in dem Testsystem enthalten. Die Definition der Testabläufe erfolgt bereits in einem JSON-Format, welches sich gut durch Webtechnologien erzeugen und bearbeiten lässt und der Webserver liefert bereits Javascript-Bibliotheken an den Browser aus, welche genutzt werden können, um ein aufwändigeres Webinterface zu gestalten.
Neben der Erweiterung des Testsystems wird in Zukunft vor allem die Anpassung des Testsystems an neue ASICs und ASSPs ein zentrales Thema sein. Auch wenn das Testsystem in seiner Struktur so designed ist, dass es einen möglichst flexiblen Testablauf zulässt, werden für die zukünftig zu testenden DUTs Anpassungen notwendig sein, um Besonderheiten dieser DUTs zu berücksichtigen. Die bestehenden Interfaces zwischen den Bestandteilen des Testsystems sorgen dafür, dass einzelne Komponenten, wie z.B. die Ansteuerung des Watchdogs des DUTs, angepasst werden können, ohne das Veränderungen im gesamten Testsystem vorgenommen werden müssen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Che17] Chehab, Mauro C.: *Remote Processor Messaging (rpmsg) Framework*. https://www.kernel.org/doc/Documentation/rpmsg.txt. Version: 2017. – [online; Stand 17.02.2019]

[DEN19] DENX Software Engineering GmbH: *Das U-Boot - the Universal Boot Loader*. https://www.denx.de/wiki/U-Boot/. Version: 2019. – [Online; Abgerufen am 12.01.2019]

[Ecm17] Ecma International: *ECMA-404 - The JSON Data Interchange Standard*. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. Version: Edition 2, 2017. – [Online; Abgerufen am 19.02.2019]

[Eri18] Eric Haszlakiewicz: *JSON-C - A JSON implementation in C*. http://json-c.github.io/json-c/json-c-0.13.1/doc/html/index.html. Version: v0.13.1, 2018. – [Online; Abgerufen am 19.02.2019]

[Han06] Hans-Jürgen Koch, Linutronix: *The Linux driver implementer's API guide - The Userspace I/O Howto*. https://www.kernel.org/doc/html/v4.14/driver-api/uio-howto.html. Version: v4.14, 2006. – [Online; Abgerufen am 19.02.2019]

[HJH89] Hill, Susan C. ; Jelemensky, Joseph ; Heene, Mark R.: Patent US4816996A - Queued serial peripheral interface for use in a data processing system. (1989), Motorola Solutions Inc.

[Hü16] Hüning, F.: *Sensoren und Sensorschnittstellen*. De Gruyter, 2016 (De Gruyter Studium). – ISBN 9783110438550

[KN15] Klar, H. ; Noll, T.: *Integrierte Digitale Schaltungen: Vom Transistor zur optimierten Logikschaltung*. Springer Berlin Heidelberg, 2015. – ISBN 9783540690177

[Lin16] Linear Technology Corporation : *LTC1661 Micropower Dual 10-Bit DAC in MSOP*. http://www.analog.com/media/en/technical-documentation/data-sheets/1661fb.pdf. Version: 2016. – [online; Stand 20.07.2018]

[PSI16] PSI5 Steering Committee: *PSI5 - Overview*. https://psi5.org/overview/. Version: 2016. – [online; Stand 01.08.2018]

[PSI18a] PSI5 Steering Committee : *PSI5 Peripheral Sensor Interface – Base Standard V2.3*. https://psi5.org/fileadmin/user_upload/01_psi5.org/04_Specification/Specifications_PDFs/v2.3/PSI5_spec_v2.3_Base.pdf. Version: 2018. – [online; Stand 01.06.2018]

[PSI18b] PSI5 Steering Committee : *PSI5 Peripheral Sensor Interface – Substandard Airbag V2.3*. https://psi5.org/fileadmin/user_upload/01_psi5.org/04_Specification/Specifications_PDFs/v2.3/PSI5_spec_v2.3_Airbag.pdf. Version: 2018. – [online; Stand 01.06.2018]

[Rea] Real Time Engineers Ltd.: *The RTOS Tick - RTOS Implementation Building Blocks*. https://www.freertos.org/implementation/a00011.html. – [Online; Abgerufen am 19.02.2019]

[Rob18] Robert Bosch GmbH: *PSI5 Specifications Overview*. https://psi5.org/specification/. Version: 2018. – [Online; Abgerufen am 17.12.2018]

[Saf16] SafeSPI Steering Committee: *SafeSPI - Serial Peripheral Interface for Automotive Safety V1.0*. https://safespi.org/fileadmin/user_upload/02_SafeSPI/04_Specification/SafeSPI_specification_v1.0.pdf. Version: 2016. – [online; Stand 17.02.2019]

[SS18] Sudhaus, Andre ; Subijanto, Tan: Patent WO2018050908A1 - Watchdog for monitoring a processor. (2018), Elmos Semiconductor AG

[Tre15] Trenz Electronic GmbH: *TE0720 Technical Reference Manual*. https://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/Modules_and_Module_Carriers/4x5/TE0720/REV03/Documents/TRM-TE0720-03.pdf. Version: 2015. – [online; Stand 02.08.2018]

[Tre18] Trenz Electronic GmbH: *TE0703 Technical Reference Manual*. https://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/Modules_and_Module_Carriers/4x5/TE0720/REV03/Documents/TRM-TE0720-03.pdf. Version: 2018. – [online; Stand 02.08.2018]

[TS02] Tietze, U. ; Schenk, C.: *Halbleiter-Schaltungstechnik*. Springer, 2002. – ISBN 9783540428497

[WGO10] Weichenberger, Lothar ; Gesell, Dr. A. ; Ohl, Christian: 5 Jahre PSI5-Protokoll - Etablierung eines Standards. In: *Hanser automotive* (2010), Nr. 05/06. http://files.hanser-tagungen.de/docs/20100601173248_24-27%20AM%205-6%2010%20FA%20PSI5.pdf

[Xil11] Xilinx, Inc.: *UG761 - AXI Reference Guide*. https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf. Version: v13.1, 2011. – [Online; Abgerufen am 10.02.2019]

[Xil14] Xilinx, Inc.: *Zynq Block Diagram*. http://www.wiki.xilinx.com/file/view/SystemDiagram.png/533382026/800x507/SystemDiagram.png. Version: 2014. – [Online; Abgerufen am 21.07.2018]

[Xil16] Xilinx, Inc.: *Zynq-7000 All Programmable SoC Overview*. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. Version: v1.10, 2016. – [Online; Abgerufen am 21.07.2018]

[Xil17] Xilinx, Inc.: *UG1186 - OpenAMP Framework for Zynq Devices*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug1186-zynq-openamp-gsg.pdf. Version: v2017.2, 2017. – [Online; Abgerufen am 14.09.2018]

[Xil18a] Xilinx, Inc.: *UG 901 - Vivado Design Suite User Guide*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug901-vivado-synthesis.pdf#nameddest=xSystemVerilogConstructs. Version: v2018.3, 2018. – [Online; Abgerufen am 17.12.2018]

[Xil18b] Xilinx, Inc.: *UG1186 - Libmetal and OpenAMP User Guide*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1186-zynq-openamp-gsg.pdf. Version: v2018.2, 2018. – [Online; Abgerufen am 10.02.2019]

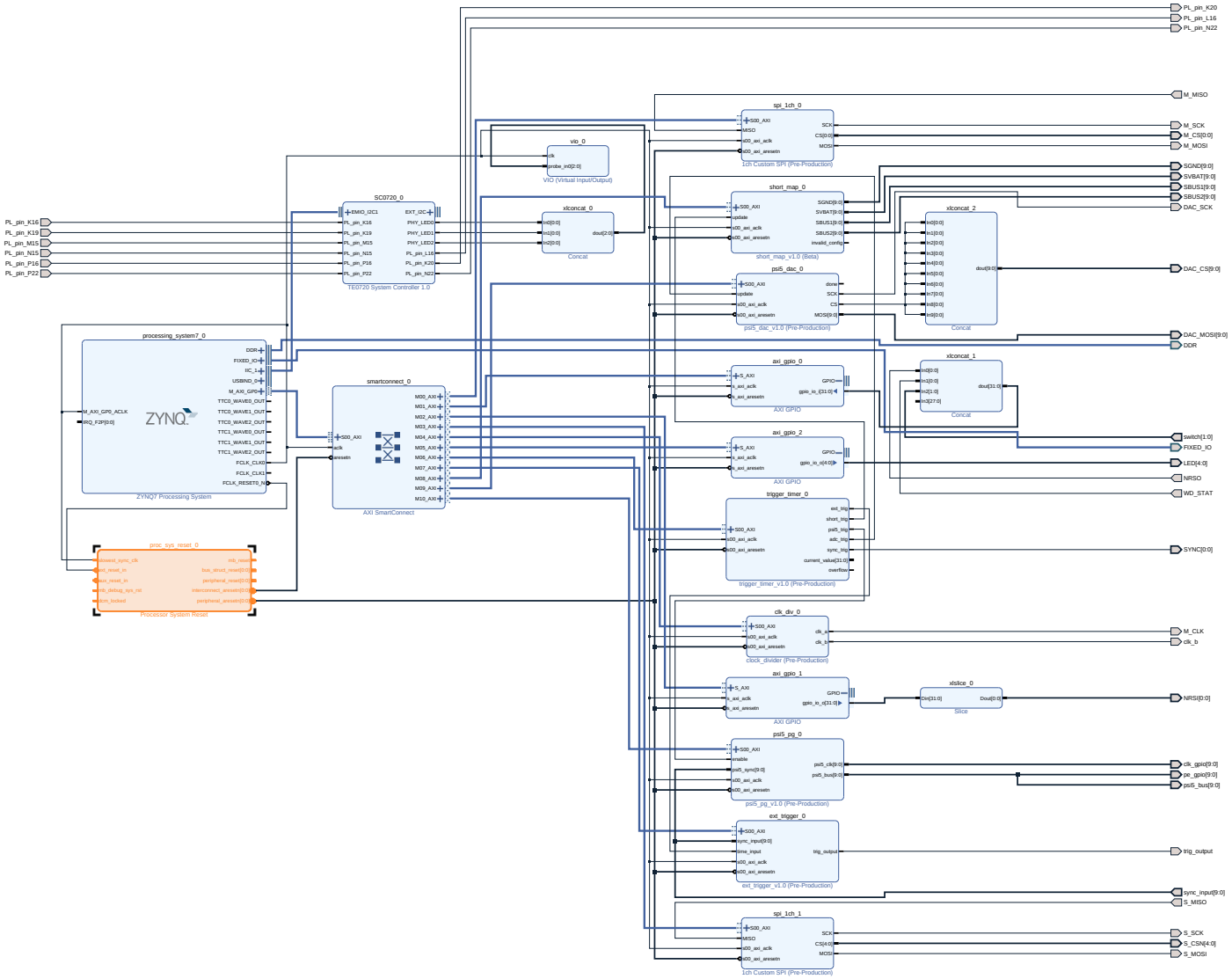
[Xil19a] Xilinx, Inc.: *Embedded Software Tools - PetaLinux Tools*. https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html. Version: 2019. – [Online; Abgerufen am 12.01.2019]

[Xil19b] Xilinx, Inc.: *Vivado Hardware Debug - Tools and Features*. https://www.xilinx.com/products/design-tools/vivado/debug.html#hardware. Version: 2019. – [Online; Abgerufen am 12.01.2019]

[ZS14] Zimmermann, W. ; Schmidgall, R.: *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur*. Springer Fachmedien Wiesbaden, 2014 (ATZ/MTZ-Fachbuch). – ISBN 9783658024192

Anhang

A.1 Top Level


PL_pin_K20
PL_pin_L16
PL_pin_N22
M_MISO
M_SCK
M_CS[0:0]
M_MOSI
SGND[9:0]
SVBAT[9:0]
SBUS1[9:0]
SBUS2[9:0]
DAC_SCK
DAC_CS[9:0]
DAC_MOSI[9:0]
DDR
switch[1:0]
FIXED_IO
LED[4:0]
NRSO
WD_STAT
SYNC[0:0]
M_CLK
clk_b
NRSI[0:0]
clk_gpio[9:0]
pe_gpio[9:0]
psi5_bus[9:0]
trig_output
sync_input[9:0]
S_MISO
S_SCK
S_CSN[4:0]
S_MOSI
PL_pin_K16
PL_pin_K19
PL_pin_M15
PL_pin_N15
PL_pin_P16
PL_pin_P22
SC0720_0
TE0720 System Controller 1.0
vio_0
VIO (Virtual Input/Output)
xlconcat_0
Concat
spi_1ch_0
1ch Custom SPI (Pre-Production)
short_map_0
short_map_v1.0 (Beta)
xlconcat_2
Concat
psi5_dac_0
psi5_dac_v1.0 (Pre-Production)
processing_system7_0
ZYNQ7 Processing System
smartconnect_0
AXI SmartConnect
axi_gpio_0
AXI GPIO
xlconcat_1
Concat
axi_gpio_2
AXI GPIO
trigger_timer_0
trigger_timer_v1.0 (Pre-Production)
proc_sys_reset_0
Processor System Reset
clk_div_0
clock_divider (Pre-Production)
axi_gpio_1
AXI GPIO
xlslice_0
Slice
psi5_pg_0
psi5_pg_v1.0 (Pre-Production)
ext_trigger_0
ext_trigger_v1.0 (Pre-Production)
spi_1ch_1
1ch Custom SPI (Pre-Production)

A.2 Device Tree

```
/include/ "system-conf.dtsi"
/ {

    reserved-memory {
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;
        rproc_0_reserved: rproc@3e000000 {
            no-map;
            reg = <0x3e000000 0x01000000>;
        };
    };

    amba {
        elf_ddr_0: ddr@0 {
            compatible = "mmio-sram";
            reg = <0x3e000000 0x400000>;
        };
    };

    remoteproc0: remoteproc@0 {
        compatible = "xlnx,zynq_remoteproc";
        firmware = "firmware";
        vring0 = <15>;
        vring1 = <14>;
        srams = <&elf_ddr_0>;
    };
    chosen {
        bootargs = "console=ttyPS0,115200 earlyprintk root=/dev/
            mmcblk1p2 rw rootwait uio_pdrv_genirq.of_id=generic-uio";
        stdout-path = "serial0:115200ns";
    };

};
/* QSPI PHY */
&qspi {
    #address-cells = <1>;
    #size-cells = <0>;
    status = "okay";
    flash0: flash@0 {
        compatible = "jedec,spi-nor";
        reg = <0x0>;
        #address-cells = <1>;
        #size-cells = <1>;
    };
};


/* ETH PHY */
&gem0 {
    phy-handle = <&phy0>;
    mdio {
        #address-cells = <1>;
        #size-cells = <0>;
        phy0: phy@0 {
            compatible = "marvell,88e1510";
            device_type = "ethernet-phy";
            reg = <0>;
        };
    };
```

```
60 };
61
62 /* USB PHY */
63
64 /{
65     usb_phy0: usb_phy@0 {
66         compatible = "ulpi-phy";
67         //compatible = "usb-nop-xceiv";
68         #phy-cells = <0>;
69         reg = <0xe0002000 0x1000>;
70         view-port = <0x0170>;
71         drv-vbus;
72     };
73 };
74
75 &usb0 {
76     dr_mode = "host";
77     //dr_mode = "peripheral";
78     usb-phy = <&usb_phy0>;
79 };
80
81 /* I2C need I2C1 connected to te0720 system controller ip */
82 &i2c1 {
83
84     iexp@20 {       // GPIO in CPLD
85         #gpio-cells = <2>;
86         compatible = "ti,pcf8574";
87         reg = <0x20>;
88         gpio-controller;
89     };
90
91     iexp@21 {       // GPIO in CPLD
92         #gpio-cells = <2>;
93         compatible = "ti,pcf8574";
94         reg = <0x21>;
95         gpio-controller;
96     };
97
98     rtc@6F {        // Real Time Clock
99         compatible = "isl12022";
100        reg = <0x6F>;
101    };
102 };
103
104
105 &spi_1ch_0 {
106     compatible = "generic-uio";
107 };
108
109 &spi_1ch_1 {
110     compatible = "generic-uio";
111 };
112
113 &clk_div_0 {
114     compatible = "generic-uio";
115 };
116
117 &axi_gpio_0 {
118     compatible = "generic-uio";
119 };
120
121 &axi_gpio_1 {
122     compatible = "generic-uio";
123 };
```

```
124
125 &axi_gpio_2 {
126     compatible = "generic-uio";
127 };
128
129 &ext_trigger_0 {
130     compatible = "generic-uio";
131 };
132
133 &short_map_0 {
134     compatible = "generic-uio";
135 };
136
137 &psi5_dac_0 {
138     compatible = "generic-uio";
139 };
140
141 &psi5_pg_0 {
142     compatible = "generic-uio";
143 };
144
145 &trigger_timer_0 {
146     compatible = "generic-uio";
147 };
```

A.3 Entwicklungsumgebung Schnellstartanleitung

Fachhochschule Dortmund
University of Applied Sciences and Arts

elmos

Verwendete Entwicklungsumgebung für die Implementierung des PSI5-Testsystems

Schnellstartanleitung

Inhaltsverzeichnis

1 FPGA / SoC Projekt

Um die Entwicklungsumgebung Xilinx Vivado zu starten, muss nach der Installation selbiger zunächst die Umgebung angepasst werden. Dazu wird das von Xilinx mitgelieferte Skript in einer Konsole geladen. Der entsprechende Befehl dazu lautet:

```
source /opt/Xilinx/Vivado/2018.2/settings64.sh
```

Gegebenenfalls ist dabei der Installationspfad oder die Versionsnummer anzupassen. Das Skript wird jedoch von Xilinx bei allen aktuelleren Vivado Versionen mit settings64.sh bezeichnet. Anschließend kann über den Befehl vivado, welchen die Umgebung nun kennt, die Entwicklungsumgebung Vivado zum Erstellen der FPGA / SoC Designs gestartet werden.

1.1 FPGA / SoC Projekt erstellen

Die Erstellung eines FPGA- bzw. SoC-Projekts für die Trenz ZYNQ SoC Module kann auf zwei Arten erfolgen. Die erste Variante ist eine manuelle Erzeugung eines Projekts und die manuelle Anpassung aller notwendigen Parameter. Dazu sollten zunächst die von Trenz bereitgestellten Boardpart Files in die Vivado Installation aufgenommen werden. Diese enthalten essentielle Definitionen von Parametern, wie den verwendeten SoC und dessen Stepping.
Die Boardpart Files werden von Trenz als Teil des Referenzdesigns bereitgestellt und können unter der folgenden URL bezogen werden:
`https://shop.trenz-electronic.de/Download/?path=Trenz_Electronic/Modules_and_Module_Carriers/4x5/TE0720/Reference_Design`
Die Installation der Boardpart Files erfolgt durch das Kopieren der Dateien aus diesem Pfad des Referenzdesigns:

```
<reference_design>/board_files/
```

und dem anschließenden Speichern unter dem folgenden Pfad der Installation der Entwicklungsumgebung Vivado:

```
/opt/Xilinx/Vivado/2018.2/data/boards/board_files/
```

Sind die Boardpartfiles installiert, kann in Vivado ein neues Projekt erzeugt werden. Bei der Erstellung des Projekts sollten die Module von Trenz nun bei der Boardauswahl zur Verfügung stehen. Ist dies nicht der Fall, sind gegebenenfalls die Rechte der kopierten Dateien anzupassen. Als Referenz für die Rechtevergabe können die anderen Boardpartfiles der Vivado Installation genutzt werden.

Nun kann mit der Entwicklung eines Designs durch das Integrieren des IP-Blocks für das Processing System in das Top-Level begonnen werden. Das Processing System repräsentiert die beiden ARM CPU-Kerne und wird nicht synthetisiert. Über den IP-Block können jedoch viele Einstellungen des Prozessorsystems und dessen Peripherie vorgenommen werden (Siehe Abschnitt 1.2). Da die Architektur jedoch komplex ist und die Anpassung vieler Parameter benötigt werden, um eine optimale Ausnutzung der vorhanden Komponenten des Trenz TE0720 Moduls mit dem Petalinux Kernel zu erreichen, existiert ein Referenzdesign, welches diese Anpassungen vornimmt. Es empfiehlt sich daher dieses Referenzdesign als Basis für alle Entwicklungen zu nutzen, da es sich um ein minimales Design handelt, welches die weiteren Entwicklungen nicht einschränkt, jedoch eine enorme Ersparnis des Entwicklungsaufwands bedeutet, da die Grundkonfiguration nicht vorgenommen werden muss.
Das Referenzdesign ist unter der folgenden URL verfügbar:
`https://shop.trenz-electronic.de/Download/?path=Trenz_Electronic/Modules_and_Module_Carriers/4x5/TE0720/Reference_Design`
Dabei ist zu beachten, dass die Version des Referenzdesigns zu der installierten Version der Vivado Entwicklungsumgebung passt. Ältere Versionen des Referenzdesigns können mit Einschränkungen auch von neueren Versionen der Entwicklungsumgebung genutzt werden. Umgekehrt ist diese Versionskompatibilität nicht gegeben.
Um das Referenzdesign zu nutzen, muss dieses zunächst in den gewünschten Pfad für das spätere Projekt entpackt werden und anschließend das folgende Skript des Referenzdesigns ausgeführt werden:

```
_create_linux_setup.sh
```

Dieses Skript bietet einen kleinen konsolenbasierten Assistenten für die Konfiguration und Erstellung des Projektes. In der Menüführung ist der Eintrag Module Selection Guide, project creation über die Ziffer 0 zu wählen und anschließend der Menüführung zu folgen. Der Assistent erzeugt daraufhin ein Projekt. Ist das Projekt erzeugt, wird von dem Skript des Assistenten automatisch Viviado gestartet und einige TCL-Skripte zur Konfiguration des Designs ausgeführt. Damit das Skript Vivado starten kann, muss die Umgebung der Konsole den Befehl vivado kennen. Dazu ist es notwendig, dass zunächst über den oben beschriebenen source Befehl die Umgebung in der Konsole geladen wird, in welcher anschließend auch das Skript zur Erzeugung des Projekts ausgeführt wird.
Das erzeugte Top-Level des Referenzdesigns sollte nun aus dem ZYNQ Processing System, einem Reset Controller und dem TE0720 System Controller bestehen. Der System Controller IP-Block repräsentiert den CPLD des TE0720 Moduls, welcher sich unter anderem um die Ansteuerung der LEDs, den I2C-Bus und weitere Komponenten kümmert. Das Projekt ist mit Abschluss der Ausführung der TCL-Scripte in Vivado fertig eingerichtet und kann nun durch den Entwickler um weitere Komponenten erweitert werden.

Alternativ kann direkt ein Bitstream und das zugehörige Hardware Definition File des vorhandenen minimalen Referenzdesigns erzeugt werden.

1.2 Einstellungen / Top Level

Alle Einstellungen des Processing Systems des ZYNQs, welches aus den beiden ARM-Core CPU-Kernen und den Peripheriekomponenten besteht, werden über das in Abbildung 1.1 dargestellte Kontextmenü des Processing System IP-Blocks im Top Level durchgeführt. Dazu zählt neben der Konfiguration von Speichercontrollern, Schnittstellen und PIN-Konfigurationen der Schnittstellen auch die Konfiguration der Taktquellen des Systems. Unter PL Fabric Clocks sind die PLL-Taktgeneratoren zu finden, welche in der programmierbaren Logik für die Implementierung von sequentieller Logik bereitstehen. Das PSI5-Testsystem nutzt nur die erste Taktquelle und teilt den Takt dieser gegebenenfalls innerhalb der IP-Blöcke herunter.

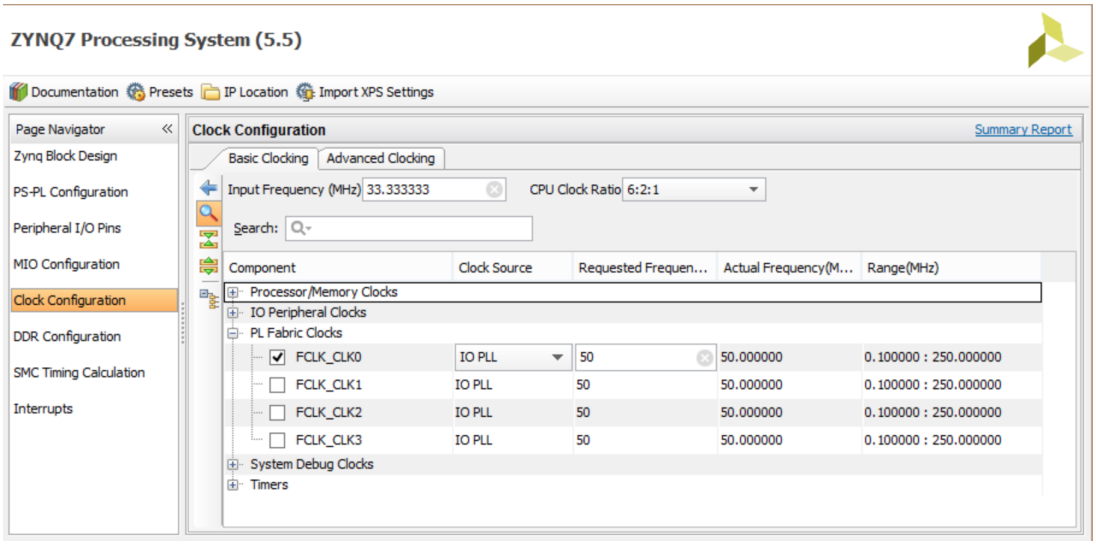


Abb. 1.1: Einstellungen des ZYNQ Processing System Blocks

Dabei ist zu beachten, dass die spätere Anwendung der hier ausgewählten Konfiguration der Komponenten, wie z.B. der PLL-Taktgeneratoren durch den First-Stage-Bootloader und nicht den Bitstream erfolgt. Werden also diese Einstellungen des ZYNQs angepasst, so muss der FSBL mit einer neuen Hardware Definition gebaut werden und dieser auf das Modul gespielt werden. Eine reine Änderung des Bitstreams ist nicht ausreichend.

1.3 Hardware Definition exportieren

Um einen FSBL zu erzeugen, benötigt das SDK eine Beschreibung der Hardware und Informationen, wie die Eigenschaften des Processing Systems zu setzen sind. Diese Beschreibung der Hardware und ihrer Eigenschaften sind in einem Hardware Definition File enthalten. Ein Hardware Definition File ist dabei nicht, wie die Bezeichnung zunächst andeutet, eine einzelne Datei, sondern ein Container mit unterschiedlichen Informationen. Um dieses Hardware Definition File zu erzeugen, nachdem die Einstellungen des Processing Systems angepasst wurden, kann, sofern es sich um ein durch den Assistenten von Trenz erzeugtes Projekt handelt, der folgende Befehl in der TCL-Konsole genutzt werden:

```
TE::hw_build_design -export_prebuilt
```

Der TCL-Befehl erzeugt dann in dem folgenden Pfad die Beschreibung der Hardware:

```
prebuilt\hardware\xxx
```

Das xxx bezeichnet, dabei einen Platzhalter im Pfad des Projekts, das durch ein von Trenz definiertes Kürzel ersetzt wird, welches die Hardwareversion beschreibt. Diese Methode des Exports beinhaltet keinen Bitstream, sondern nur die reine Hardware Definition. Soll ebenfalls ein Bitstream der implementierten logischen Funktionen im FPGA exportiert werden und anschließend in der Boot.bin eingebettet werden, so muss der Bitstream zunächst durch einen kompletten Synthese- und Implementierungsdurchlauf erzeugt werden. Ist dieser Schritt abgeschlossen, kann über das Menü File → Export → Export Hardware das Hardware Definition File exportiert werden. Das Menü bietet dabei die Option den Bitstream in das Hardware Definition File einzubinden. Wird als Pfad in diesem Menü Local to Project gewählt, befindet sich das Hardware Definition File in einem Order des Projektverzeichnisses, welcher mit [projektname.sdk] bezeichnet wird.

1.4 First Stage Bootloader

Der First Stage Bootloader wird in diesem System benötigt, um das Processing System zu konfigurieren und die programmierbare Logik mit dem Bitstream zu beschreiben. Die Erstellung und Kompilierung des First Stage Bootloaders findet nicht in Vivado statt, sondern im Xilinx xsdk. Dazu muss zunächst das xsdk über den Befehl xsdk gestartet werden. Wird dieser Befehl in der Konsole nicht gefunden, ist die zugehörige Umgebung vorher zu laden. Dazu wird der aus dem ersten Abschnitt dieses Kapitels bereits bekannte Befehl source genutzt. Allerdings ist in diesem Fall der Pfad so abzuändern, dass die settings64.sh im Installationspfad des xsdks genutzt wird.

Um den First Stage Bootloader zu generieren, ist über das Menü File → New → Application Project ein neues Softwareprojekt zu erzeugen. Die OS Platform ist dabei als Standalone zu wählen und die Hardwareplattform muss der Platform mit dem, im vorherigen Abschnitt exportierten, Hardware Definition File entsprechen. Existiert diese Plattform noch nicht, so kann gegebenenfalls über den Button New eine neue Hardwareplattform erzeugt werden, welche das spezifische Hardware Definition File des Systems nutzt. Als Prozessor muss der erste CPU-Kern des Processing Systems gewählt werden, daher ist für den Prozessor die Option ps7_cortexa9_0 zu wählen. Des Weiteren ist ein neues Board Support Package zu erzeugen. Sind alle Optionen gesetzt, kann im nächsten Schritt des Projektassistenten ein Template für die Softwareanwendung gewählt werden. Da ein First Stage Bootloader erzeugt werden soll, ist an dieser Stelle die Option Zynq FSBL zu wählen. Der Assistent generiert daraufhin ein Projekt mit allen benötigten Einstellungen und Quelldateien für einen First Stage Bootloader.
Da die Trenz Module mit ihren Hardwarekomponenten jedoch ein wenig von einem generischen ZYNQ-System abweichen, für welche Xilinx den FSBL bereitstellt, müssen einige Quelldateien angepasst und hinzugefügt werden. Andernfalls funktioniert unter anderem der Ethernet- und USB-Controller des Moduls nicht. Die zu modifizierenden Quelldateien sind main.c, fsbl_hooks.h und fsbl_hooks.c. Zusätzlich müssen noch die Quell- und Headerdateien te_fsbl_hooks.h/.c dem Projekt hinzugefügt werden. Diese benötigten Dateien befinden sich in dem Projektordner des Referenzdesigns von Trenz unter folgendem Pfad:

```
sw\_lib/sw\_apps/znyq\_fsbl
```

Anschließend kann der First Stage Bootloader kompiliert und die Binärdatei in einem späteren Schritt in das Bootimage eingebaut werden.

2 Petalinux

Die Konfiguration und das Kompilieren des Petalinux Kernels und Xilinx Rootdateisystems erfolgt vollständig über eine make-Umgebung mit entsprechenden mitgelieferten Skripten. Die Verwendung einer GUI ist nicht vorgesehen. Um die, für die Petalinux Tools benötigte, Umgebung zu laden, muss, wie bereits bei der Vivado IDE beschrieben, zunächst der folgende source Befehl genutzt werden:

```
source /opt/Xilinx/petalinux/settings64.sh
```

Die folgenden Abschnitte beschreiben flüchtig die benötigten Befehle zur Erstellung, Konfiguration, Kompilierung und dem Export eines Petalinux Projekts. Ein hilfreiches weitergehendes Dokument für die Verwendung der Petalinux Tools von Xilinx ist der Reference Guide UG1144, welcher unter folgender URL erhältlich ist:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1144-petalinux-tools-reference-guide.pdf

2.1 Petalinux Projekt erstellen

Ein neues Petalinux Projekt wird über den folgenden Befehl angelegt:

```
petalinux-create --type project --template <CPU_TYP> --name <
    PROJEKT_NAME>
```

Das Projekt wird dabei in dem Verzeichnis erzeugt, in welchem sich die Konsole gegenwärtig befindet. Durch diesen ersten Schritt wird zunächst ein generisches Projekt erstellt, welches nur an den Architektur der CPU angepasst ist, nicht jedoch an die eigentliche Hardware des vorliegenden Systems. Dazu zählen insbesondere Einstellungen der Schnittstellen und Taktfrequenzen. Damit das Petalinux Projekt an die Hardware des Systems angepasst werden kann, muss dem Petalinux Projekt über den folgenden Befehl das Hardware Definition File übergeben werden, welches in einem vorangegangenen Schritt exportiert wurde:

```
petalinux-config --get-hw-description <PFAD ZU HDF DATEI>
```

Dieser Schritt muss bei jeder Änderung der Hardware ebenso wie der Export der HDF-Datei wiederholt werden.

2.2 Petalinux Konfiguration

Die Konfiguration des Petalinux erfolgt in vier Schritten. Zunächst werden Systemeinstellungen für das Linux durchgeführt, anschließend wird der Bootloader konfiguriert, die Einstellungen des eigentlichen Kernels durchgeführt und abschließend der gewünschte Inhalt des Rootdateisystems angepasst.

Um die Systemeinstellungen vorzunehmen, wird dieser Befehl verwendet:

```
petalinux-config
```

Zunächst ist zu überprüfen, dass die beiden nachfolgenden Optionen, wie angegeben, gesetzt sind, damit die korrekte UART-Schnittstelle, welche mit dem USB-UART-Adapter des Trägerboards verbunden ist, verwendet wird:

```
CONFIG_SUBSYSTEM_SERIAL_PS7_UART_0_SELECT=y
CONFIG_SUBSYSTEM_SERIAL_IP_NAME="ps7_uart_0"
```

Als weitere Option muss spezifiziert werden, dass sich das Rootdateisystem auf einer SD-Karte befindet, da dieses bei dem System im zu einer SD-Karte identischen eMMC abgelegt werden soll. Die Option befindet sich dazu unter folgendem Eintrag im Konfigurationsmenü:

```
Image Packaging Configuration > Root filesystem type > SD CARD
```

Damit ist die Konfiguration der Systemeinstellungen abgeschlossen. Anschließend wird der Second Stage Bootloader, welcher in diesem System aus U-Boot besteht, konfiguriert. Der benötigte Befehl dazu lautet:

```
petalinux-config -c u-boot
```

Da der Bootloader noch einige Anpassungen nicht nur von Parametern des Konfigurationsmenüs, sondern auch von Headerdateien benötigt, sei an dieser Stelle auf die Wiki von Trenz unter der folgenden URL verwiesen:
`https://wiki.trenz-electronic.de/display/PD/TE0720+Test+Board#TE0720TestBoard-SoftwareDesign-PetaLinux`

Als letzter Konfigurationsschritt werden die Einstellungen des Kernels durchgeführt. Das zugehörige Konfigurationsmenü wird durch den folgenden Befehl geöffnet:

```
petalinux-config -c kernel
```

Da das Trenz Modul über eine I2C-RTC verfügt, ist es notwendig in der Kernelkonfiguration anzugeben, dass der Kernel die Funktionen für die Verwendung dieser RTC besitzen sollen. Dazu ist die folgende Option zu setzten:

```
CONFIG_RTC_DRV_ISL12022
```

Alle weiteren Optionen beziehen sich nicht auf die physikalische Hardware des Testsystems, sondern werden benötigt, um eine Trennung der beiden CPU-Kerne vornehmen zu können und einen rpmsg-Bus zwischen beiden erstellen zu können. Die Module-Unterstützung des Kernels wird benötigt, um zu einem späteren Zeitpunkt das remoteproc-Framework und die Treiber für den rpmsg-Bus nachladen zu können. Der entsprechende Eintrag ist wie folgt bezeichnet und muss gesetzt werden:

```
Enable loadable module support
```

Die Einstellungen für das remoteproc-Framework sind unter dem folgenden Eintrag zu finden und auf M für ein Modul-basierten Einsatz zu stellen:

```
 Device Drivers > Remoteproc drivers > Support ZYNQ remoteproc
```

Die Konfiguration des Rootdateisystems erfolgt über diesen Befehl:

```
petalinux-config -c rootfs
```

Diese Einstellungen und das Erzeugen des Petalinux Rootdateisystems ist notwendig auch wenn später z.B. das Rootdateisystem eines Debian Images verwendet wird, da sich in dem hier definierten Rootdateisystem die benötigten Treibermodule befinden. Diese müssen dann gegebenenfalls in das neue Rootdateisystem kopiert werden. Als Einstellung ist dabei folgendes zu aktivieren:

```
Filesystem Packages > misc > kernel-module-rpmsg-user
```

Diese Option integriert den benötigten Treiber für den Zugriff auf den rpmsg-Bus aus dem Userspace. Die Konfiguration ist damit abgeschlossen, sodass nach der Anpassung des Device Trees mit dem Kompilieren begonnen werden kann.

2.3 Device Tree

Der Device Tree muss angepasst werden, damit der Kernel Informationen erhält, wie die Hardware aufgebaut ist und wie er diese verwenden soll. Die zu bearbeitende Datei für Modifikationen des Device Trees befindet sich unter diesem Pfad:

```
project-spec/meta-user/recipes-bsp/device-tree/files/openamp-overlay.
    dtsi
```

In dieser Datei wird im Rahmen des PSI5-Testsystem die im Anhang enthaltenen Elemente des Device Trees vollständig kopiert.

2.4 Linux / U-Boot kompilieren

Das Kompilieren des Kernels wird über den folgenden Befehl gestartet:

```
petalinux-build
```

Dabei wird nicht nur der eigentliche Kernel kompiliert, sondern auch der Second Stage Bootloader U-Boot. Ebenfalls wird durch den selben Befehl automatische das Rootdateisystem erzeugt. Ist dieser Prozess abgeschlossen, befindet sich unter dem Pfad images/linux/ das Rootdateisystem, sowie die image.ub-Datei. Die image.ub-Datei ist das Linux-Image, welches aus dem Kernel-Image und dem Device Tree Blob besteht. Die image.ub-Datei wird in der Bootpartition des eMMCs abgelegt, während das Rootdateisystem sich in der zweiten Partition befindet. Alternativ kann die image.ub-Datei auch auf einer SD-Karte abgelegt werden, wenn von der SD-Karte gebootet werden soll. Die Anpassung der Einstellung, wo sich das Rootdateisystem befindet, erfolgt über den Bootargumente Eintrag des Device Trees.

2.5 boot.bin Export / Flashen

Aus dem First Stage Bootloader, dem Second Stage Bootloader U-Boot und dem FPGA Bitstream wird die boot.bin-Datei erzeugt, welche sämtliche Konfigurationen bis zum Start des Linux-Kernels übernimmt. Erzeugt wird die Datei dabei wie folgt:

```
petalinux-package --boot --fsbl <PFAD ZUR FSBL ELF> --fpga <PFAD ZUM
    BITSTREAM> --u-boot
```

Die entstehende boot.bin Datei kann entweder bei einem Bootvorgang von der SD-Karte aus auf selbiger abgelegt werden oder permanent in den QSPI-Flash des TE0720 Moduls geschrieben werden. Zum Beschreiben des QSPI-Flashs muss in der xsdk Entwicklungsumgebung unter dem Menüpunkt Xilinx Tools der Eintrag Program Flash gewählt werden. Dieser öffnet einen Assistenten zum Beschreiben des QSPI-Flashs, bei welchem unter dem Feld Image File die boot.bin-Datei angegeben wird. Zum Beschreiben des QSPI-Flashs muss der hw_server zur Kommunikation mit dem JTAG-Interface gestartet werden. Typischerweise genügt ein simpler Aufruf des folgenden Befehls:

```
hw_server
```

Gegebenenfalls sind die Rechte des auszuführenden Nutzers anzupassen, sodass der hw_server Zugriff auf das USB-JTAG-Interface erhält.

2.6 Petalinux starten

Der Petalinux Kernel kann entweder über den JTAG gestartet werden oder den normalen Bootprozess. Für Testzwecke bietet es sich an das Starten über den JTAG durchzuführen, da das ständige Beschreiben der SD-Karte oder des eMMCs mit dem Kernelimage entfällt. Der dazu benötigte Befehl lautet:

```
petalinux-boot --jtag --kernel --bitstream <PFAD ZUM BITSTREAM>
```

Für den normalen Bootprozess muss die boot.bin Datei in einem, von dem festen Bootrom des ZYNQ erreichbaren, Speicher abgelegt werden. Zeitgleich muss dem Bootrom mitgeteilt werden, wo sich die Datei bzw. ihr Inhalt befindet. Dazu werden auf dem Trägerboard die DIP-Schalter so gesetzt, dass der ZYNQ in dem richtigen Speicher nach der boot.bin Datei sucht. Zur Auswahl stehen dabei entweder die SD-Karte im Slot des Trägerboards oder der eMMC des Moduls. Die entsprechenden benötigten Schalterstellungen, sowohl für den JTAG-Zugriff als auch die Bootspeicherquelle sind unter der nachfolgenden URL von Trenz beschrieben:
`https://wiki.trenz-electronic.de/display/PD/TE0703+TRM#TE0703TRM-DIPswitches`

2.7 Petalinux / xsdk Anwendungsentwicklung

Die Entwicklung von Anwendungen für das Petalinux findet in der xsdk IDE statt. Eine neue Anwendung kann dabei über den Projektassistenten erzeugt werden. Als OS Platform wird in diesem Fall Linux angegeben, woraufhin sich die weiteren Einstellungen in ihrem Umfang einschränken. Ein Board Support Package oder die Auswahl eines CPU-Kerns ist nicht notwendig oder möglich, da der Linux Kernel dies verwaltet. Lediglich die Angabe der gewünschten Programmiersprache, der Toolchain und des Rootdateisystems sollte erfolgen. Die Toolchain befindet sich im gnu-Ordner des Installationspfads des SDKs. Das Rootdateisystem wird benötigt, da dieses die Kernel-Header und weitere Headerfiles enthält, welche als Referenz für die Anwendungsentwicklung benötigt werden. Das Rootdateisystem für die Entwicklung von Anwendungen besteht daher aus anderen Bestandteilen als das eigentliche Rootdateisystem, welches vom Linux Kernel verwendet

wird. Es kann über die beiden folgenden Befehle erzeugt werden:

```
petalinux-build --sdk
petalinux-package --sysroot
```

Sind diese beiden Befehle erfolgreich ausgeführt worden, befindet sich unter folgendem Pfad das Dateisystem für die Entwicklung von Linux Anwendungen:

```
/images/linux/sdk/sysroots
```

Bei der Ausführung dieser Schritte ist zu beachten, dass die Petalinux Tools einen Internetzugriff benötigen, um einige Bestandteile aus dem Internet nachzuladen. Haben die Petalinux Tools keinen Internetzugriff, wird dies durch entsprechende Fehlermeldungen angezeigt. Dabei wird auch immer die .bb-Datei angegeben, welche das fehlgeschlagene Rezept für die Erzeugung der entsprechenden Komponenten enthält. Das .bb-Rezept kann dabei so abgeändert werden, dass es auch offline funktioniert. Dazu muss der Eintrag des .bb-Rezepts, der die URL oder das Git Repository, unter welcher die Quellen der Komponenten zu beziehen sind, spezifiziert, durch einen FILE-Eintrag mit einem Pfad zu einer lokalen Kopie dieser Daten ersetzt werden. Die Abänderungen der .bb-Rezepte sind jedoch sehr mühsam und zeitaufwändig, sodass davon dringend abgeraten wird. Das aktuelle Image der virtuellen Maschine in der die Petalinux Tools installiert sind, enthält bereits ein vollständiges Rootdateisystem für die Entwicklung von Anwendungen, sodass dieses nicht mehr durchgeführt werden muss.
Für das Debuggen der Anwendungen empfiehlt es sich auf dem eigentlichen Zielsystem einen TCF-Server zu installieren, sodass direkt aus der xsdk IDE über TCP/IP ein Debugger gestartet werden kann.

3 FreeRTOS

Das xsdk von Xilinx beinhaltet eine bereits angepasste Version von FreeRTOS, sodass keine externen Quellen für das Echtzeitbetriebssystem benötigt werden. Im Gegensatz zum Linux Kernel findet die Verwaltung und Kompilierung von FreeRTOS aufgrund des bedeutend geringeren Umfangs nicht über eine make-Umgebung sondern in der xsdk IDE statt.

3.1 FreeRTOS Projekt erstellen

Die FreeRTOS-Instanz wird durch das Erstellen eines neuen Projekts erzeugt. Dazu wird wie auch für den FSBL aus Abschnitt 1.4 der Projektassistent gewählt. Als OS Platform wird in diesem Fall FreeRTOS in der modifizierten Xilinx Version gewählt. Als Hardware Platform, kann die bereits für den FSBL erstellte Plattform weiter genutzt werden. Das Board Support Package kann jedoch nicht weiterverwendet werden, da der FSBL und die FreeRTOS-Instanz unterschiedliche Einstellungen benötigen. Es muss daher ein neues Board Support Package für die FreeRTOS-Instanz erzeugt werden. Da die FreeRTOS-Instanz von dem zweiten CPU-Kern ausgeführt werden soll, ist als Prozessor die Option ps7_cortexa9_1 zu verwenden.
In dem nächsten Schritt des Assistenten stehen drei Beispiel-Projekte als Template zur Verfügung. Es empfiehlt sich eines dieser Projekte als Ausgangsbasis für die weitere Entwicklung zu nutzen, da diese Beispiele die rpmsg-Schnittstelle bereits konfigurieren. Andernfalls müssen bei einem blanken Projekt als Ausgangsbasis die benötigten Funktionen für die rpmsg-Schnittstelle von dem Entwickler hinzugefügt werden.
Hilfreich an dieser Stelle ist gegebenenfalls die ältere Version (2017.2) des Dokuments UG1186 von Xilinx, welches den Prozess besser beschreibt als neuere Ausgaben. Dieses Dokument ist über die folgende URL zu erhalten: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug1186-zynq-openamp-gsg.pdf` Dabei ist zu beachten, dass der Petalinux Abschnitt dieses Dokuments nicht mehr dem gegenwärtigen Stand entspricht und daher nicht als Referenz genutzt werden sollte.

3.2 Konfiguration

In den Einstellungen des Board Support Packages muss an der folgenden Stelle noch eine Option hinzugefügt werden, damit es zu keinen Ressourcenkonflikten kommt:

```
Settings > Overview > Drivers > ps7_cortexa9_1.
```

Die folgende Option ist an den bestehenden Eintrag anzuhängen:

```
-DUSE_AMP=1
```

Diese Option verhindert, dass die FreeRTOS-Instanz zwischen den beiden CPU-Kernen geteilte Ressourcen erneut initialisiert, da diese bereits von dem Petalinux initialisiert werden.

In den Einstellungen des Board Support Packages der FreeRTOS-Instanz befindet sich ein Eintrag, der mit tick_rate bezeichnet ist. Über diesen Eintrag kann die Anzahl der Tick-Interrupts pro Sekunde der FreeRTOS-Instanz bestimmt werden. Dabei ist zu beachten, dass bei einer zu hohen Tickrate einige Makros zur Umrechnung von Ticks in physikalische Zeiten nicht mehr funktionieren, da deren Auflösung erschöpft ist.

3.3 FreeRTOS starten

Um den zweiten CPU-Kern mit der FreeRTOS-Instanz zu starten, werden Befehle über das Pseudodateisystem sysfs genutzt. Zunächst muss angegebenen werden, an welcher Stelle sich die, durch die zweite CPU auszuführende, Binärdatei der FreeRTOS-Instanz befindet. Dazu wird der folgende Befehl genutzt:

```
echo freeRTOS\_test > /sys/class/remoteproc/remoteproc0/firmware
```

In diesem Beispiel soll die Binärdatei freeRTOS_test verwendet werden. Dabei ist zu beachten, dass die Angabe des Dateipfades immer relativ zu dem Pfad /lib/firmware erfolgt. Dies bedeutet, dass sich die Binärdatei unter folgendem Pfad befinden muss:

```
/lib/firmware/freeRTOS\_test
```

Ist die auszuführende Datei spezifiziert, kann mittels des folgenden Zugriffs über die sysfs Einträge des remoteproc Framworks die zweite CPU gestartet werden:

```
echo start > /sys/class/remoteproc/remoteproc0/state
```

Analog dazu kann durch die folgende Abwandlung des Befehls der zweite CPU-Kern gestoppt werden:

```
echo stop > /sys/class/remoteproc/remoteproc0/state
```

Soll eine Kommunikation über den rpmsg-Bus und den Beispiel-Treiber von Xilinx für einen Zugriff aus dem Userspace auf den rpmsg-Bus verwendet werden, ist das entsprechende Modul durch den folgenden Befehl zu laden:

```
modprobe rpmsg_user_dev_driver
```

A.4 CD:

Inhalt:

- **Dokumentation Masterthesis**