HSB
Hochschule Bremen
City University of Applied Sciences

Fakultät Elektrotechnik und Informatik

Studiengang Electronics Engineering - M.Sc.EE

Deutsches Zentrum für Luft- und Raumfahrt (DLR) Bremen

Institut für Raumfahrtsysteme

Entwicklung einer UVM-Testbench für SpaceWire IP-Cores

Masterarbeit

Verfasser : B. Eng. Philipp Schlemm

Matr.-Nr. 346360

Erstprüfer : Prof. Dr.-Ing. Stefan Wolter

Zweitprüfer und Betreuer : M. Sc. Kai Borchers (DLR)

Abgabetermin: 17.12.2018

Hochschule Bremen
City University of Applied Sciences

HSB

Hochschule Bremen Neustadtwall 30 D-28199 Bremen

Herrn
Philipp Schlemm
Nachtigalstraße 56
28217 Bremen

Abschlussarbeit / Master´s Thesis
346360

Bremen, 01.06.2018

Andrea Zebrowski
Sachbearbeiterin
Immatrikulations- und Prüfungsamt

Neustadtswall 30
D-28199 Bremen
T +49 421 5905 2373
F +49 421 5905 2351
andrea.zebrowski@hs-bremen.de
→ hs-bremen.de

Öffnungszeiten
MO 13:00–15:00 Uhr
DI + DO 09:00–12:00 Uhr

Sehr geehrter Herr Schlemm,

der Prüfungsausschuss für den Masterstudiengang Electronics Engineering hat Ihren Antrag vom 23.05.2018 bearbeitet und das Thema Ihrer Masterarbeit und die Bedingungen am 29.05.2018 wie folgt festgelegt / the examination board confirms:

Ihre Abschlussarbeit ist eine / Thesis is: Einzelarbeit / Individual work

Thema / Topic:

Entwicklung einer UVM-Testbench für SpaceWire IP-Cores

6 Monate / Months ab: 15.06.2018

Abgabetermin / Date of Submission: 17.12.2018

Als 1. Prüfer / Examiner wurde bestellt: Herr Prof. Dr. Stefan Wolter

Als 2. Prüfer / Examiner wurde bestellt: Herr Kai Borchers M.Sc. (extern)

Mit freundlichen Grüßen

Im Auftrag

Andrea Zebrowski

cc: Herr Prof. Dr. D. Kraus / Frau J. Jin Steinhardt / Herr Prof. Dr. S. Wolters / Herr K. Borchers

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbstständig verfasst habe.
Es wurden keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt. Die wörtlichen oder sinngemäß übernommenen Zitate bzw. Gedanken aus anderen Publikationen habe ich als solche kenntlich gemacht.
Diese Arbeit wurde zuvor weder in gleicher noch ähnlicher Form bei einem anderen Prüfer als Prüfungsleistung eingereicht.

Bremen, den 17.12.2018

_____

Unterschrift

Abstract

Thema der Masterarbeit
Entwicklung einer UVM-Testbench für SpaceWire IP-Cores

Stichworte
UVM, Testbench, SystemVerilog, SpaceWire, Kommunikationscontroller, Hardware-Verifikation

Kurzfassung
Die vorliegende Masterarbeit befasst sich mit der UVM-basierten Verifikation eines frei verfügbaren SpaceWire IP-Cores. Das Ziel der Arbeit besteht in der Entwicklung einer UVM-Testbench für jene IP-Cores. Die Testbench soll zur automatisierten Verifikation beliebiger SpaceWire IP-Cores dienen, die nach dem ECSS-E-ST-50-12C Standard entwickelt wurden. Im theoretischen Teil werden zunächst die für das Design benötigten UVM Grundlagen erklärt. Weiterhin ist eine Erläuterung der relevanten Inhalte des SpaceWire Standards enthalten. Darauf folgen Informationen zum SpaceWire Light IP-Core, der als bereits geprüftes DUT verwendet wird, sowie zum grundsätzlichen Testbench Entwurfskonzept. Der praktische Teil beginnt mit der Beschreibung des Verifikationsplans, der auf Basis einer Zusammenfassung des Standards erstellt wurde und das Verhalten des IP-Cores abdeckt. Schließlich wird die Entwicklung der Testbench, von der Grundstruktur über den Entwurf der Agents und Sequences bis zu abschließenden Tests zur Verifikation des IP-Cores ausführlich erläutert. Das Ergebnis der Arbeit ist eine parametrisierbare UVM-Testbench zur Verifikation von SpaceWire IP-Cores, für die die automatisierte Verifikation des DUT noch implementiert werden muss.

Title of the master thesis
Development of a UVM testbench for SpaceWire IP cores

Keywords
UVM, testbench, SystemVerilog, SpaceWire, communication controller, hardware verification

Abstract
The present thesis approaches the UVM based verification of a freely available SpaceWire IP core. The aim is to develop a UVM testbench for those IP cores. The testbench shall be utilized for the automated verification of arbitrary SpaceWire IP cores, developed in conformity with the ECSS-E-ST-50-12C standard. In the theoretical part UVM basics required for the design are explained. Furthermore an explanation of the relevant contents of the SpaceWire standard is included. After that information about the SpaceWire Light IP core, which is utilized as a verified DUT as well as the basic testbench design concept is provided. The practical part begins with the description of the verification plan, developed based on a summary of the standard and which covers the behavior of the IP core. Finally, detailed explanations regarding the testbench development, from it's fundamental structure and the design of the agents and sequences to final tests and the verification of the IP core are given. The result of the thesis is a parameterizable UVM testbench for the verification of SpaceWire IP cores for which automated verification of the DUT still needs to be implemented.

Vorwort und Danksagung

An dieser Stelle möchte ich zunächst einen besonderen Dank an Herrn M.Sc. Kai Borchers vom DLR in Bremen richten, der mir ein interessantes Thema für eine Masterarbeit unterbreitete und es mir ermöglichte, diese beim DLR in Bremen zu schreiben. Außerdem stand Herr Borchers während der Bearbeitungszeit der Masterarbeit und bei der Entwicklung der UVM-Testbench stets für Fragen und Antworten zur Verfügung. Aufgrund seiner fachlichen Kompetenz auf dem Gebiet der UVM basierten Verifikation, konnte er durch seine konstruktive Kritik, seine Anregungen und Unterstützung zur kontinuierlichen Verbesserung der Testbench beitragen.

Als nächstes bedanke ich mich bei Herrn Prof. Dr.-Ing. Stefan Wolter an erster Stelle für den Vorschlag einiger potentieller Themen für eine Abschlussarbeit an der Hochschule Bremen. Weiterhin danke ich Herrn Prof. Wolter für den Hinweis zur Kontaktaufnahme mit dem DLR, wodurch diese Arbeit zustande kam. Abschließend danke ich noch meiner Familie für ihre beständige Unterstützung und Zuversicht.

Bei der Suche nach einem passenden Thema für eine Masterarbeit im Bereich Hardware-Entwurf und Verifikation kontaktierte ich anfangs Herrn Prof. Wolter, da er bereits als Prüfer und Betreuer meiner Bachelorarbeit fungierte. Diese befasste sich mit der Hardware-Modellierung und Verifikation von CRC-Generatoren und -Checkern und hatte zum Ziel, einen parametrisierbaren und in Hardware synthetisierbaren CRC-Generator zu entwickeln. Das Ergebnis der Arbeit war ein frei parametrisierbarer und synthesefähiger Hardware-CRC-Generator in VHDL für beliebige Eingangsdatenwortbreiten (d.h. auch seriell) und CRC-Polynome.

Für die Abschlussarbeit an der Hochschule standen drei Themen zur Verfügung. Das erste befasste sich mit dem hochschulinternen Hardware MP3-Decoder Projekt. Hierbei ging es um die Kombination der einzelnen Hardware-Modelle des Decoders, die Verifikation einiger Stages und die abschließende Zusammenfassung der einzelnen Modelle des Decoders auf top-level Ebene inklusive der Verifikation des Gesamtdesigns und möglicher Fehlerkorrekturen. Beim zweiten Thema ging es um die Verifikation eines frei verfügbaren $\mathrm{I^2C}$-Bus Controllers ($\mathrm{I^2C}$ Master Core) inklusive Fehlererkennung und -korrektur. Das dritte Thema behandelte die Implementierung von Handshake-Verfahren gleichrangiger VHDL-Modelle. Auch hier sollte eine Verifikationsumgebung entwickelt und Tests auf geeigneter Hardware durchgeführt werden.

Nach Verabredung mit Herrn Borchers standen neben dem gewählten SpaceWire Thema zwei weitere Themen zur Auswahl. Dabei ging es beim ersten um die Entwicklung eines UVM NAND Agents zur Verifikation von NAND-Flash Controllern, entwickelt nach aktuellem ONFI Standard. Das zweite behandelte die Auslegung und Entwicklung eines BCH IP-Cores, der z.B. zum Schutz von NAND-Flash Pages verwendet wird. Der IP-Core sollte mindestens zwei Fehler korrigieren können, und die Berechnung der Fehlerstellen algebraisch durchführen (Berlekamp-Massey-Algorithmus).

Im Rahmen meiner Bachelorarbeit befasste ich mich bereits mit dem Hardware-Entwurf, daher sollte bei der Masterarbeit die Hardware-Verifikation im Vordergrund stehen. Die Wahl fiel letztendlich auf die Entwicklung der UVM-Testbench für SpaceWire IP-Cores, da so einerseits die Möglichkeit einer externen Abschlussarbeit bestand und andererseits für mich interessante Themen wie Kommunikationscontroller, Handshake-Verfahren, serielle Datenübertragung und Codierung enthalten sind.

Die Einarbeitung in die UVM Grundlagen gestaltete sich anfangs relativ schwierig, da in der mir

zur Verfügung stehenden Literatur, hier hauptsächlich das UVM-Cookbook der Mentor Verification Academy, das Thema meines Erachtens nicht sehr einsteigerfreundlich behandelt bzw. methodisch aufgearbeitet wird. Dies kommt spätestens bei Thema Sequences und Sequence Items zum Tragen. Der Leser findet hier häufig unterstützende Codebeispiele, die sich an einem konkreten Beispiel (AHB- und APB-Bus) orientieren. Dieses Beispiel zieht sich durch die wichtigsten Kapitel des Cookbooks und erschwert aufgrund seiner grundsätzlichen Gestaltung das Verständnis von UVM.
Der SpaceWire Standard hingegen wird gut verständlich beschrieben, obwohl einige Definitionen an mehreren Stellen mit ähnlichem Wortlaut auftauchen, die unterschiedlich interpretiert werden können. Ein Beispiel dafür ist die Beschreibung der Bedingungen für Zustandsübergange der Link-FSM.
Es stellte sich heraus, dass mein Verständnis primär durch eigenes Programmieren und durch die direkte Anwendung bei der Entwicklung der Testbench gefördert wurde, was sich jedoch nachteilig auf den zeitlichen Fortschritt auswirkte. Im Treiber des Data-Strobe Agents ist außerdem der komplette SpaceWire CODEC (FSM, Transmitter, Receiver) integriert. In der Konsequenz konnte der festgelegte Zeitplan leider nicht eingehalten werden, weshalb die automatisierte Verifikation des SpaceWire Light IP-Cores noch aussteht. Hierfür fehlen sowohl das Scoreboard als auch die Functional Coverage. Die Monitore der Agents bauen in der vorliegenden Version der Testbench Sequence Items für die Weitergabe an ein Scoreboard auf. Auch die Schnittstelle dahin ist bereits vorhanden. Zusammen mit den entwickelten Sequences und den beiden Basistests (normaler Funktionstest, Fehler-Funktionstest), wurde ein passender Abschlusspunkt erreicht.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Symbol- und Abkürzungsverzeichnis

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
API	Application Programming Interface
AVM	Advanced Verification Methodology
BFM	Bus Functional Model
CODEC	Coder/Decoder
DMA	Direct Memory Access
DS	Data-Strobe
DUT	Device Under Test
DUV	Device Under Verification
ECSS	European Cooperation for Space Standardization
EEP	Error End Of Packet marker
EMV	Elektromagnetische Verträglichkeit
EOP	End Of Packet marker
eRM	e Reuse Methodology
ESC	Escape Token
FCT	Flow Control Token
FIFO	First In - First Out Speicherverfahren
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
IF	Interface
IP-Core	Intellectual property core
L-Char	Link-Character
LSB	Least significant Bit
LVDS	Low Voltage Differential Signalling
MBit	Megabit (10^6 bit)
MSB	Most significant Bit
NAND	Negation der Konjunktion (UND-Verknüpfung)
N-Char	Normal Character

NULL	NULL Token
OSI	Open Systems Interconnection
OVM	Open Verification Methodology
P2P	Peer-to-Peer
PCB	Printed Circuit Board
RAM	Random Access Memory
RMAP	Remote Memory Access Protocol
RTL	Register-Transfer Level / Register-Transfer Logic
RX	Receiver (Empfänger)
SoC	System-on-Chip
SV	SystemVerilog
TB	Testbench
TC	Time-Code
TLM	Transaction-level modeling
TX	Transmitter (Sender)
UVM	Universal Verification Methodology
VHDL	Very High Speed Integrated Circuit Hardware Description Language
XOR	Exklusiv-Oder (Kontravalenz)

$\wedge$	Logische Und-Verknüpfung (Konjunktion)
$\vee$	Logische Oder-Verknüpfung (Disjunktion)
$\neg$	Negation

1 Einleitung

SpaceWire beschreibt ein Netzwerk zur Datenkommunikation im Weltraum, beispielsweise an Bord eines Satelliten. Es wird zur seriellen Datenübertragung mit Geschwindigkeiten bis zu 400 MBit/s (Vollduplex) verwendet, wobei sogenannte SpaceWire Nodes entweder direkt miteinander (P2P), oder über SpaceWire Router kommunizieren. Die Kommunikation kann innerhalb von bzw. zwischen Datenverarbeitungssystemen (z.B. mit Sensoren) stattfinden. Die Daten sind nach dem Data-Strobe (DS) Schema codiert (zwei Datenleitungen je Senderichtung) und werden schließlich per LVDS (vier Signalleitungen je Senderichtung) übertragen. Für das Routing von Datenpaketen wird das Prinzip des Wormhole Routing verwendet. Dabei wird stets ein komplettes Paket von einem Eingangsport zum jeweiligen Ausgangsport eines Routers, in Abhängigkeit des im Headerbyte des Pakets definierten Ports, weitergeleitet. Sofern ein Eingangsport gerade genutzt wird, blockiert dieser ein ankommendes Paket. Per Flow-Control wird sichergestellt, dass der Sender (Source Node) in diesem Fall keine weiteren Daten schickt. Die Portfreigabe erfolgt sobald das Paket vollständig versendet wurde.
SpaceWire ist standardisiert durch die ECSS, der European Cooperation for Space Standardization, und konzipiert für den Einsatz im Weltraum. Der Standard basiert zum Teil auf anderen Standards wie dem IEEE Std 1355-1995, IEEE Std 1394-1995 und dem IEEE Std 15963.3-1996. Er bessert allerdings an diversen Stellen nach, erstens, um ihn für Weltraumanwendungen geeignet zu machen und zweitens, um nicht eindeutige Formulierungen aus den IEEE-Standards zu korrigieren oder Fehler zu beseitigen. Erweiterungen im Standard betreffen vor allem die Robustheit bzw. Fehlertoleranz, EMV sowie den Stromverbrauch. SpaceWire findet weltweit Anwendung in diversen Raumfahrtmissionen der ESA und NASA. Die dem Standard zugrunde liegenden Protokolle regeln den Verbindungsaufbau (Aufbau von SpaceWire Links per NULL/FCT Handshake Protokoll), das Verhalten im Fehlerfall bzw. bei Verbindungsabbruch („Exchange of Silence“ Protokoll) und die Datenübertragung. Die Protokolle zur Kommunikation per SpaceWire sind auf Hardwareebene in SpaceWire IP-Cores umgesetzt und können danach in FPGA's implementiert werden. Der Begriff IP-Core (engl. intellectual property core) beschreibt eine urheberrechtlich geschützte Einheit, zum Beispiel ein Hardware-Modell oder einen integrierten Schaltkreis (Chip), die zur Nutzung dem Lizenzrecht unterliegt. Auf der Internetseite von OpenCores sind eine Vielzahl von IP-Cores für unterschiedlichste Anwendungen frei verfügbar, ebenso für SpaceWire.
Zur Verifikation solcher SpaceWire IP-Cores und Überprüfung der Konformität zum SpaceWire Standard (ECSS-E-ST-50-12C - Version 31. Juli 2008), wird im DLR in Bremen eine Testumgebung in Form einer (SystemVerilog) UVM-Testbench benötigt. Wie der Aufgabenbeschreibung für die Masterthesis zu entnehmen ist, soll die zu entwickelnde Testbench zur Überprüfung beliebiger SpaceWire IP-Cores dienen, die nach dem oben genannten Standard entwickelt wurden. Im Rahmen der Abschlussarbeit wird dann eine UVM-basierte Verifikation des SpaceWire Light IP-Cores durchgeführt. Der IP-Core besitzt zwei Schnittstellen (Interfaces), einmal das FIFO-Interface sowie das Data-Strobe Interface. Ersteres dient zur Kommunikation mit den beiden FIFOs (Sender und Empfänger) des IP-Cores, letzteres zum Senden/Empfangen von Daten auf low-level Data- und Strobe-Signal Ebene). Eine zusätzlich hinzugefügte Schnittstelle, ehemals Bestandteil des FIFO-Interfaces, wird für das Senden/Empfangen von Time-Codes verwendet und im Abschnitt 3.5.1 der Arbeit erläutert. Die Schnittstellen müssen von zwei bzw. jetzt drei UVM-Agents betrieben

werden. Die Überprüfung der korrekten und zum Standard konformen Funktionsweise des IP-Cores soll automatisiert ablaufen. Dies soll sowohl mit Hilfe von SystemVerilog Assertions in separierten Modulen realisiert werden, als auch durch einen Vergleich auf TLM-Ebene über ein UVM-Scoreboard erfolgen. Für die Arbeit sind im Detail folgende Teilbereiche festgelegt:
Zunächst ist ein Verifikationsplan in geeigneter Form zu erstellen, der die per SpaceWire Standard relevanten Anforderungen sowie das Verhalten des IP-Cores abdeckt. Dieser dokumentiert alle maßgeblichen Eckpunkte der Verifikation. Als zweites sind die zwei bzw. drei Agents, jeweils ein Agent pro Interface, zu entwickeln. Zur Generierung pseudorandomisierter Daten bzw. ausreichender Stimuli Variationen sind Sequences/Virtual Sequences (Top-level Sequences) zu beschreiben. Diese sollen dann in abgeleiteten Tests (Testfällen), basierend auf einem Basis-Test zur Konfiguration der Testumgebung, Stimuli über die Schnittstellen zum DUT bringen. Als nächstes soll das UVM-Scoreboard zum Vergleich von generierten bzw. gesendeten und empfangenen Daten (engl. prediction and comparison) entwickelt werden. Die beiden letzten Teilbereiche sind die Beschreibung des Functional Coverage zum Erfüllen des Verifikationsplans und abschließende Analyse der Ergebnisse aus der Verifikation des SpaceWire Light.
Im zweiten Kapitel dieser Arbeit folgt zunächst der Theorieteil. Hier wird zu Beginn eine Übersicht über UVM gegeben und die Frage geklärt, warum UVM für die Hardware-Verifikation geeignet ist. Es folgen Erläuterungen zur allgemeinen UVM-Testbench Architektur, z.B. zu den Basiskomponenten, Informationen zum Mechanismus bei der Verbindung der Testbench mit dem DUT, der Konfiguration der Testumgebung sowie zu UVM Sequences und Sequence Items. Der nächste Abschnitt 2.2 beschreibt die für die Entwicklung der Testbench relevanten Inhalte des SpaceWire Standards wie beispielsweise die SpaceWire Token, Flow Control, Protokolle und die Link-FSM. Darauf folgt eine etwas ausführlichere Beschreibung des SpaceWire Light IP-Cores. Hier stehen vor allem das Konzept, die Funktionsweise des Senders und Empfängers, das IP-Core Interface und die Konfiguration des DUT im Vordergrund.
Das dritte Kapitel der Arbeit beschreibt die Entwicklung der UVM-Testbench. Den Anfang macht der Abschnitt 3.1, in dem das ursprüngliche Design Konzept erklärt wird, auf dessen Grundlage die Testbench aufgebaut ist. Als nächstes geht es um die Erstellung des Verifikationsplans. Die darauf folgenden Abschnitte beschreiben die Phasen der Testbench-Entwicklung, beginnend mit dem Aufbau der Grundstruktur (Grundgerüst) inklusive aller Komponenten nach einer gewissen Entwicklungsrichtlinie, der Instanziierung des IP-Cores und der Beschreibung der Komponenten und Konfigurationsoptionen der Testbench. Weiterhin folgen detaillierte Informationen zum Design der Agents, insbesondere bezüglich des jeweiligen Drivers, Monitors und Sequence Items, sowie zum Aufbau der Sequences. Allen voran ist der Data-Strobe Agent wegen seiner Komplexität noch ausführlicher beschrieben, denn dieser enthält sowohl die SpaceWire Link-FSM als auch Sender, Empfänger, Encoder und Decoder. Im Abschnitt 3.8 werden die abgeleiteten Tests (Testfälle) für einen Funktionstest des SpaceWire Light IP-Cores erläutert.
Zum Schluss werden die Ergebnisse der Arbeit zusammen gefasst und ein Ausblick auf die Entwicklung des Scoreboards, mögliche Verbesserungen der Testbench sowie zur Functional Coverage gegeben.

Am Ende dieser Einleitung sei noch die projektspezifisch genutzte Soft- und Hardware aufgelistet:

- **Atom Open-Source Texteditor v1.26.1** inkl. diverser Packages/Add-Ons, zur Entwicklung der UVM-Testbench und als Editor des SystemVerilog Quellcodes.
- **Gedit Texteditor v3.18.3** für diverse Textdateien und zur Bearbeitung von Simulationsscripts (.do-Dateien) für QuestaSIM.
- **LyX Texteditor v2.3.1** als Editor für TeX/LaTeX-Dateien und zum Schreiben der Masterarbeit.
- **QuestaSIM v10.6d** von Mentor Graphics zur Simulation und Verifikation des SpaceWire Light IP-Cores und der UVM-Testbench.
- **SpaceWire Light IP-Core (OpenCores.org)** von Joris van Rantwijk als bereits geprüftes DUT in der UVM-Testbench.

2 Theorie

2.1 UVM Grundlagen

2.1.1 Übersicht

Die Universal Verification Methodology (UVM) ist eine im IEEE Std 1800.2-2017 Standard definierte fortschrittliche Methodik zur Verifikation integrierter Schaltkreise und SoC Designs und basiert auf dem IEEE Std 1800 Standard für SystemVerilog. UVM wurde durch die Accellera Systems Initiative standardisiert und stellt eine Erweiterung der von Mentor Graphics und Cadence entwickelten Open Verification Methodology (OVM) und anderer Methodiken (AVM, eRM u.a.) dar. Im Grunde ist UVM eine Bibliothek vordefinierter Basisklassen, also eine strukturierte Sammlung von SystemVerilog Klassen, die erweiterbar sind (Stichwort Vererbung) und als Vorlage (engl. template) zur Entwicklung objekt-orientierter Testbenches dienen. Die Vorteile von UVM liegen folglich in der Modularität, Skalierbarkeit und in einem klar definierten hierarchischen Aufbau. Die Methodik legt ihre Schwerpunkte vor allem auf die Wiederverwendbarkeit von Komponenten (engl. design reusability) und die Automatisierung von Testabläufen durch vordefinierte Funktionen und UVM Makros. Diese Vereinheitlichungen und Richtlinien des UVM Standards machen ihn als Grundlage für die Entwicklung von Testbenches äußerst interessant. UVM ist außerdem Open-Source, enthält neben der Klassenbibliothek ein Referenzhandbuch und wird ständig weiterentwickelt. In der aktuellen Version 1.2 der UVM-Klassenbibliothek sind im Vergleich zu vorherigen Versionen neue Funktionen hinzugefügt und Fehler behoben worden. Die Methodik wird mittlerweile von verschiedenen Unternehmen und bedeutenden Herstellern von Simulationssoftware wie Mentor Graphics, Cadence und Synopsys mit steigender Tendenz unterstützt.
Die oben genannten Gründe sprechen für den Einsatz von UVM als Grundlage für die Entwicklung einer Testbench. Im folgenden Abschnitt wird die allgemeine Architektur von UVM-Testbench beschrieben.

2.1.2 UVM-Testbench Architektur

Der grundlegende Unterschied zwischen VHDL/Verilog Testbenches und UVM-Testbenches ist, dass die klassischen Testbenches auf statischen Objekten (Module) aufbauen und daher vergleichsweise wenig Flexibilität besitzen. Statische Objekte werden zu Beginn einer Simulation erstellt und verweilen dann während der Simulationszeit unverändert in Speicher. UVM-Testbenches hingegen bauen auf SystemVerilog Klassen auf, wodurch dynamische Objekte und eine objekt-orientierte Programmierung eingeführt werden. Dieser Grad der Abstraktion ermöglicht ein wesentlich flexibleres Testbench-Design, da dynamische Objekte während der Simulationszeit im Speicher erstellbar und wieder abbaubar sind. Der Abbau erfolgt, falls nicht mehr auf sie referenziert wird. In der UVM-Testbench ist nur noch das Top-Level Testbench Modul als einziges statisches Objekt vorhanden.
Eine UVM-Testbench besitzt eine klare hierarchische Struktur, die stets identisch aufgebaut ist. In UVM sind Phasen definiert, die zum Aufbau der Struktur, zur Konstruktion der Sub-Komponenten und deren Verbindung untereinander, zur Durchführung der eigentlichen Simulation (der Tests) und zu abschließenden Analysezwecken am Ende der Simulation dienen. Die Phasen werden

nacheinander in einer bestimmten Reihenfolge durchlaufen. Siehe dazu das spätere Kapitel über UVM-Phasen.
Das UVM-Cookbook (Mentor Graphics (2013)) stellt zwei UVM-Testbench Architekturen bzw. Hierarchien vor. Die erste ist die sogenannte Integrationsstruktur und beschreibt die Integration-Level Testbenches. Sie enthält zwei Block-Level Testbenches, die im selben Projekt auf einer höheren Integrationsebene, im Hinblick auf Wiederverwendbarkeit, verbunden werden. Man spricht hier auch von vertikaler Wiederverwendung (engl. vertical reuse). Importiert man UVM-Komponenten in ein anderes Projekt, spricht man dagegen von horizontaler Wiederverwendung (engl. horizontal reuse). Die zweite Architektur ist die Basisarchitektur, auch Blockstruktur genannt und beschreibt die Block-Level Testbenches. Die in dieser Arbeit entwickelte UVM-Testbench besitzt eine solche Struktur. Daher wird sie im Folgenden ausführlicher erläutert.

2.1.2.1 Block-Level Testbench

Die UVM-Testbench mit Blockstruktur enthält auf der dritthöchsten hierarchischen Stufe (nach der Test Klasse und dem Top-Level Testbench Modul) eine Umgebung (engl. environment(env)) mit derselben Struktur. Die Block-Level Env dient als Container (Behälter) für alle weiteren Sub-Komponenten der Testbench. Direkte Sub-Komponenten sind beispielsweise die Agents und ein Scoreboard.

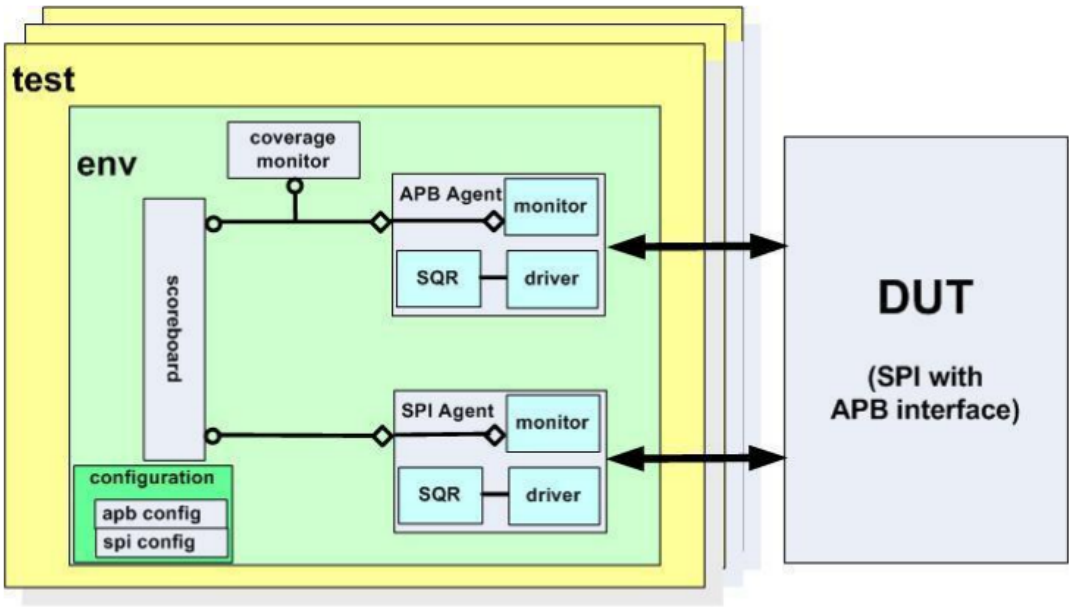

Abbildung 1: Block-Level UVM-Testbench - Hierarchische Struktur

Die Abbildung 1 aus dem UVM-Cookbook zeigt die hierarchische Struktur einer Block-Level Testbench. Nach dem Top-Level Modul und der Test Klasse, im weiteren Verlauf nur noch als Test oder Basistest bezeichnet, ist die (Top-Level) Env zu sehen. In diesem Fall sind zwei Agents inklusive Sub-Komponenten, das Scoreboard, ein Coverage Monitor und die Konfigurationsobjekte beider Agents enthalten.

2.1.2.2 Basiskomponenten

Die UVM Bibliothek enthält drei Hauptklassen für den Entwurf einer UVM-Testbench. UVM-Komponenten (uvm_components) dienen dem Aufbau der hierarchischen Struktur. UVM-Objekte (uvm_objects) sind die Grundlage zur Konfiguration der Testbench/Testumgebung. Der dritte Haupttyp sind die UVM-Transaktionen (uvm_transactions) zur Generierung von Stimuli und zu Analysezwecken. Die in diesem Abschnitt beschriebenen Basiskomponenten sind diejenigen Klassen einer UVM-Testbench, die durch Erweiterung (Vererbung) auf den UVM-Komponenten aufbauen und diese um zusätzliche Methoden und Funktionen, je nach Bedarf, ergänzen.
Auf oberster Ebene einer UVM-Testbench befindet sich das **Top-Level Testbench Modul**. Es ist ein einfaches SystemVerilog Modul und dadurch, wie bereits erwähnt, das einzige statische Objekt. Darin wird das Testobjekt (DUT/DUV) instanziiert und die Verbindung Testbench zu DUT über virtuelle Interfaces geschaffen. Weiterhin ist ein Initial-Block zum Aufruf der run_test() Methode zum Start der UVM-Phasen vorhanden. Handles für die virtuellen Interfaces werden hier in der UVM-Konfigurationsdatenbank (uvm_config_db) abgelegt und die Interfaces instanziiert. Das Top-Level Modul importiert relevante Packages, wie z.B. das UVM-, das Tests- und das Testparameter-Package. Das Tests-Package bindet alle abgeleiteten Tests zur Verifikation des DUT ein. Das Testparameter-Package enthält Parameter zur Konfiguration des DUT (Generics) und der Testbench. Außerdem sind im Top-Level Modul alle Taktgeneratoren untergebracht.
Die Test Klasse bzw. der **Basistest** (test_base, abgeleitet von uvm_test) ist die nächste Komponente in der Hierarchie und verantwortlich für die Konfiguration der Testbench, die Initialisierung des Konstruktionsprozesses durch den Aufbau der nächsten Hierarchiestufe (build() Methode der Test Klasse) und das Anlegen von Stimuli. Dies geschieht durch Starten der Hauptsequenz, wodurch ausgewählte Tests aufgerufen werden. Für jeden Testfall wird die Test Klasse erweitert, da der Aufbau- und Konfigurationsprozess stets identisch ist. Diese **erweiterten Tests** (abgeleitet von test_base) stellen also die Testfälle dar. Jeder erweiterte Test erstellt und startet dann die Sequences. Mit dem Aufbau der nächsten Hierarchiestufe, siehe oben, ist die Definition und der Aufbau der Env oder der Top-Level Env gemeint, falls wegen einer Integrationsstruktur mehrere Environments (jene der integrierten Block-Level Testbenches) vorhanden sind. Da der Aufbauprozess die Hierarchiebenen von oben nach unten durchläuft, werden somit alle weiteren Sub-Komponenten aufgebaut und konfiguriert. Die Klasse erstellt weiterhin die Konfigurationsobjekte von Sub-Komponenten (Env, Agents) und weist diesen Objekten Handles zu, die dann ebenso wie virtuelle Interface Handles in die uvm_config_db gesetzt werden. Das Konfigurationsobjekt der Environment wird verschachtelt aufgebaut, um Informationen über den Aufbau, die Verbindung und die Konfiguration von Sub-Komponenten in der Hierarchie nach unten durchzureichen.
Die **UVM Env** (environment, abgeleitet von uvm_env) oder auch Top-Level Env ist ein Container, der alle weiteren Sub-Komponenten einer UVM-Testbench auf der nächsten Hierarchiestufe zusammen fasst, z.B. um eine neue Integrationsebene zu schaffen. Die Komponenten der Env sind in der Regel die Agents und ein Scoreboard. Sie werden hier deklariert und Handles dafür angelegt. Das Environment Konfigurationsobjekt dient dazu, Informationen an die Agents zu übermitteln, die entscheiden welche zugehörigen Sub-Komponenten aufzubauen sind, wie sie funktionieren sollen und auf welche Weise sie zu verbinden sind. Dazu werden in der Env die Build- und Connect-

Phasen durchlaufen und zum Beispiel der Driver eines Agents mit seinem Sequencer verbunden, falls es sich um einen aktiven Agent handelt. Im Falle eines passiven Agents wird lediglich ein Monitor zur Beobachtung der Pin-Level Aktivitäten aufgebaut, konfiguriert und mit dem Analyse-Port verbunden. Die Build-Phase der Env sorgt dafür, dass weitere Build-Phasen benötigter Sub-Komponenten durchlaufen werden. Die Connect-Phasen sorgen dann für die Verbindung dieser Komponenten, wobei die Hierarchieebenen in entgegen gesetzter Richtung, von unten nach oben, durchlaufen werden.

Das **UVM Scoreboard** (uvm_scoreboard) ist eine Verifikationskomponente, die Vergleiche auf TLM-Ebene durchführt und das korrekte Verhalten des DUT überprüfen soll. Es erhält Sequence Items bzw. Transaktionen über die Analyse-Ports von mindestens zwei Agents, aufgebaut von deren Monitoren, und vergleicht sie miteinander. Dadurch kann ein Vergleich von zu erwartenden Werten mit Werten vom Ausgang des DUT stattfinden.

Ein **Predictor** (engl. Vorhersager) wird häufig im Zusammenhang mit einem Scoreboard eingesetzt. Er berechnet zu erwartende Werte aus den generierten Stimuli für den TLM-Vergleich.

Der **Functional Coverage Monitor** (uvm_subscriber) wird zum Sammeln von Functional Coverage Informationen während eines Tests verwendet. Dazu werden in ihm Covergroups definiert.

Der **UVM Agent** (uvm_agent) stellt eine der wichtigsten Komponenten der UVM-Testbench dar. Er ist die Verifikationskomponente zur Generierung und zur Überwachung von Transaktionen (engl. transactions) auf Pin-Level Ebene. Der Agent agiert also direkt auf Signalen der Interfaces, über die Testbench und DUT miteinander kommunizieren und befindet sich daher auf der niedrigsten Hierarchieebene. Die Agent Klasse selbst ist ein Container (Block) für das zugehörige Agent-Konfigurationsobjekt sowie weitere Verifikationskomponenten, die hier deklariert und Handles zugewiesen werden. Außerdem ist ein UVM Analyse-Port (uvm_analysis_port) zur Verbindung mit externen Analyse-Komponenten (Scoreboard) vorhanden. An diesem Port werden Informationen vom Monitor über Pin-Level Aktivitäten über dessen Analyse-Port weitergeleitet, das heißt beide Analyse-Ports sind miteinander verbunden. Das Agent-Konfigurationsobjekt beschreibt, welche Sub-Komponenten aufzubauen sind, ob es sich also um einen aktiven oder passiven Agent handelt und definiert ein Handle für das zugehörige virtuelle Interface, auf dem Driver und Monitor arbeiten.

Der **UVM Driver** (uvm_driver), auch Treiber genannt, setzt die Daten aus einer Serie von Sequence Items, die über eine Sequence definiert ist, in Pin-Level Transaktionen um. Der Driver ist so gesehen die aktive Komponente eines Agents. Er kommuniziert über ein virtuelles Interface direkt mit dem DUT. Der **UVM Sequencer** (uvm_sequencer) wird von Driver benötigt, um Sequence Items einer Sequence zu erhalten. Dazu werden die Sequence Item Ports von Sequencer und Driver während der Connect-Phase über den entsprechenden Agent verbunden. Der Sequencer ist die Schnittstelle zur Kommunikation zwischen einer Sequence und einem Driver und kontrolliert den Fluss von Sequence Items.

Der **UVM Monitor** (uvm_monitor) hingegen ist die passive Komponente und bildet das Gegenstück zum Driver. Der Monitor erzeugt also keine Pin-Level Transaktionen. Seine Aufgabe ist lediglich die Beobachtung der Pin-Level Aktivitäten auf dem (virtuellen) Interface und der Wiederaufbau von Sequence Items, die dann zu Analysezwecken beispielsweise zum Scoreboard, oder an andere Analysekomponenten, gesendet werden. Für den Aufbau der Sequence Items benötigt der Monitor Kenntnisse über grundliegende Protokolle, nach denen das DUT agiert.

Weitere mögliche Sub-Komponenten eines Drivers sind laut UVM-Cookbook ein Functional Coverage Monitor zum Sammeln spezifischer Coverage Informationen, ein Scoreboard und ein **Responder** (uvm_driver). Dieser ist eine passive Variante eines Drivers und reagiert nur auf Pin-Level Aktivitäten anstatt sie zu generieren.
SystemVerilog **Packages** werden bei UVM zur Zusammenfassung bzw. Gruppierung von Komponenten und Sub-Komponenten verwendet. Dabei wird die hierarchische Struktur der Testbench berücksichtigt und solche Definitionen und Klassen zusammengefasst, die in Relation zueinander stehen. Zum Beispiel schließt das Package eines Agents Sub-Komponenten wie Driver, Monitor und Sequencer, das zugehörige Konfigurationsobjekt sowie das Sequence Item ein. Ein Package kann außerdem auf einer nächsthöheren Hierarchiestufe die Packages der darunter liegenden Stufe inkludieren.

2.1.2.3 UVM-Phasen

Bei der Durchführung der Tests und zur Verifikation des DUT werden verschiedene UVM-Phasen durchlaufen. Diese dienen dem geregelten Ablauf bei der Konstruktion der Testbench und der anschließenden Simulation. Sie sind in drei Standardphasen bzw. Kategorien eingeteilt. Die erste Kategorie beschreiben die Build-Phasen (build & connect). Dazu gehören die Build-Phase selbst, die Connect-Phase und die End-of-Elaboration Phase. Während dieser Phasen vor der eigentlichen Simulation wird die Testbench aufgebaut, konfiguriert und die einzelnen Sub-Komponenten miteinander verbunden. Für diese Vorgänge wird keine Simulationszeit benötigt. Mit den in der zweiten Kategorie enthaltenen Run(-time) Phasen beginnt die Simulation durch den Start der abgeleiteten Tests. Die einzelnen Phasen sind in der Abbildung 2 dargestellt. Die letzte Kategorie sind die Cleanup- oder Analyse-Phasen. Sie ermöglichen das Sammeln der Simulationsdaten, die Auswertung der Testergebnisse und das Erstellen eines Berichts.

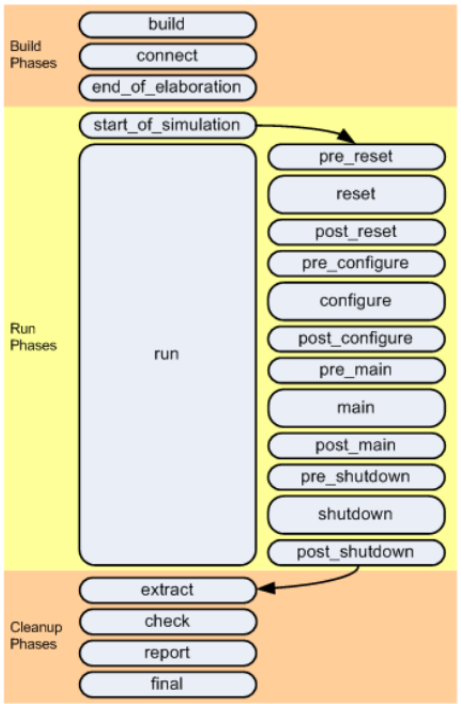

Abbildung 2: Übersicht der UVM-Phasen

Wie Mentor Graphics (2013), S. 50 zu entnehmen ist, werden beim Großteil aller Testbenches für gewöhnlich nur die reset, configure, main und shutdown Phase durchlaufen. Die folgende Abbildung verdeutlicht die Vorgänge während der Build-Phase.

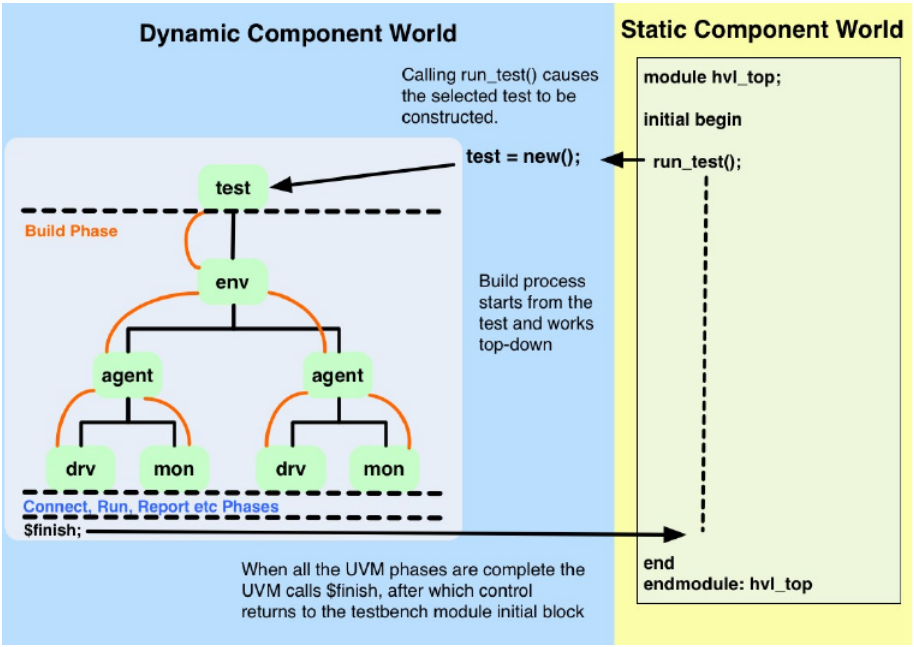

Abbildung 3: UVM-Testbench - Build-Phase

Zu erkennen ist der Start des Aufbaus der Testbench ausgehend vom Test. Dies geschieht in der Hierarchie von oben nach unten. Es folgen alle weiteren Phasen, wie zuvor beschrieben, beginnend mit der Connect-Phase. Sind alle Phasen durchlaufen, gelangt man zum Top-Level Testbench Modul zurück.

2.1.2.4 UVM-Factory

Die UVM-Factory ermöglicht das Ersetzen einer Komponente oder eines Objekts eines bestimmten Typs durch eine Komponente oder ein Objekt eines abgeleiteten Typs. Dies erlaubt dann beispielsweise den einfachen Austausch von Testbench-Komponenen im Sinne der Wiederverwendbarkeit. Der Vorteil der UVM-Factory liegt darin, dass die Struktur der Testbench bzw. der Quellcode unverändert bleiben kann. Damit das funktioniert, müssen einige Regeln eingehalten werden. Zunächst muss jede Komponente und jedes Objekt in der Factory registriert werden, indem z.B. am Anfang der Klasse das entsprechende UVM-Factory Registrierungsmacro (‘uvm_component_utils(<comp>) bzw. ‘uvm_object_utils(<obj>)), mit der Komponente oder dem Objekt als Parameter, aufgerufen wird. Für parametrisierte Komponenten und Objekte existieren ebenfalls passende UVM-Macros (‘uvm_component_param_utils(<comp>) bzw. ‘uvm_object_param_utils(<obj>)).
Außerdem sind beim Aufruf des class constructors („new“) Standardwerte (engl. constructor defaults) festzulegen, die sich unterscheiden, je nachdem ob es sich um eine Komponente oder ein Objekt handelt. Der class constructor wird dafür in eine Funktion new() eingebettet. Diese constructor defaults sind nötig, da die class constructors der erweiterten (vererbten) Klassen

uvm_component (für Komponenten) und uvm_object (für Objekte) selbst virtuelle Methoden sind. Deren Prototypenvorlage muss berücksichtigt werden. Der benötigte Basiscode sieht wie folgt aus.

Für eine Komponente:

```
function new(string name = "<component_name>", uvm_component parent = null);
    super.new(name, parent);
endfunction: new
```

Für ein Objekt:

```
function new(string name = "<object_name>");
    super.new(name);
endfunction: new
```

Die dritte Regel betrifft das Erstellen von Komponenten und Objekten während der Build-Phase. Wie bereits erwähnt, enthalten Konfigurationsobjekte Informationen darüber, welche Sub-Komponenten benötigt und erstellt werden müssen. Diese sind ebenso wie Objekte mittels create() Methode aufzubauen. Der Basiscode dafür ist der folgende.

Für eine Komponente:

```
<comp_handle_name> = <component_name>::type_id::
    create("<comp_handle_name>", this);
```

Für eine parametrisierte Komponente:

```
<comp_handle_name> = <component_name> #(param1_val, param2_val,...)::type_id::
    create("<comp_handle_name>", this);
```

Für ein Objekt:

```
<obj_handle_name> = <object_name>::type_id::create("<obj_handle_name");
```

Für ein parametrisiertes Objekt:

```
<obj_handle_name> = <object_name> #(param1_val, param2_val,...)::type_id::
    create("<obj_handle_name");
```

Der dargestellte Basiscode ist Bestandteil der (erweiterten) Klasse jeder Komponente bzw. jedes Objekts.
Ein weiterer Punkt im Zusammenhang mit der Factory sind Factory Overrides. Wenn Komponenten oder Objekte den Factory Regeln entsprechend per create() Methode aufgebaut werden, wird die Typen-ID als Referenz für das zurück gegebene zugehörige Handle verwendet. Mit einem Factory Override wird für diesen Vorgang eine andere Typen-ID benutzt. Das resultierende Handle verweist dadurch auf einen anderen Objekttyp. So besteht die Möglichkeit, auf einen abgeleiteten bzw. erweiterten Typ über das ursprüngliche Handle zu verweisen. Damit können Sub-Typen von Komponenten oder Objekten erstellt werden, die auf einem Standard-Typ aufbauen und auf diesen referenzieren. Mit dem unten dargestellten Basiscode aus dem UVM-Cookbook lassen sich Typ- und Instanz-Overrides für Komponenten und Objekt-Overrides realisieren.

Typ-Override einer Komponente:

```
<original_type>::type_id::set_type_override(<sub_type>::get_type(), 1);
```

Typ-Override einer parametrisierten Komponente:

```
<original_type> #(param1_val, param2_val,...)::type_id::
    set_type_override(<sub_type> #(param1_val, param2_val,...)::get_type(), 1);
```

Bei Instanz-Overrides ist für Komponenten und parametrisierte Komponenten der hierarchische Pfad des Originaltyps anzugeben.

Instanz-Override einer Komponente:

```
<original_type>::type_id::set_inst_override(<sub_type>::get_type(), "<path>");
```

Instanz-Override einer parametrisierten Komponente:

```
<original_type> #(param1_val, param2_val,...)::type_id::set_inst_override(
    <sub_type> #(param1_val, param2_val,...)::get_type(), "<path>");
```

Der hierarchische Pfad ist vollständig anzugeben:

```
<path> = uvm_test_top.m_env.m_ds_agent.m_driver
```

Mit diesem Beispiel ist ein Override des Drivers des Data-Strobe Agents möglich.

2.1.3 TB-DUT Kommunikation

Wie bereits nebensächlich erwähnt, kommunizieren die Testbench und das DUT über sogenannte virtuelle Interfaces miteinander. Normalerweise wird jedem Agent genau ein Interface zugewiesen, das in einem logischen Zusammenhang stehende Ports/Signale des DUT enthält. Die Interfaces sind im Top-Level Testbench Modul statisch instanziiert. Statische Objekte und deren Inhalte sind jedoch nicht dynamisch referenzierbar. SystemVerilog bietet dafür (bislang) keine direkten Maßnahmen. Das bedeutet die Ports des DUT lassen sich nicht direkt mit der Testbench verbinden. Abhilfe schafft der Einsatz eines virtuellen Interfaces. Über virtuelle Interfaces sind in der (dynamischen) Umgebung einer UVM-Testbench dann Inhalte der (statischen) Interface-Instanz dynamisch referenzierbar. Dazu wird ein virtuelles Interface-Handle zur Interface-Instanz benötigt.

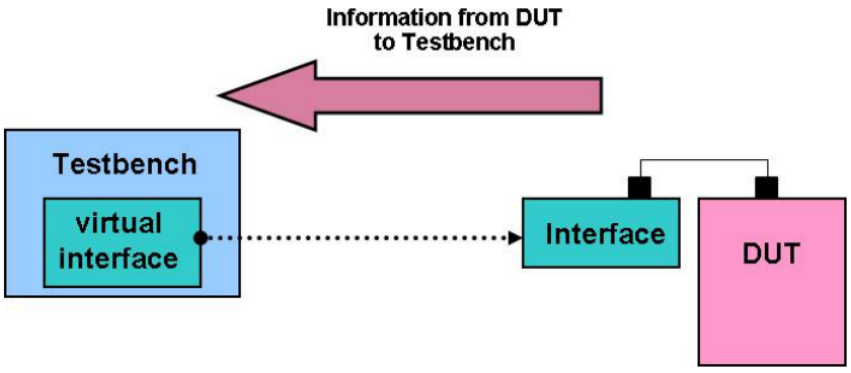

Abbildung 4: Allgemeine TB-DUT Kommunikation

Auf diese Weise wird die Verbindung der Ports des DUT über die Verbindung mit den Signalen der Interface-Instanz und die Referenz darauf über das Handle des virtuellen Interface in der Testbench hergestellt. Bei der Verwendung virtueller Interfaces muss als nächstes der Testbench mitgeteilt werden, wo sich die Interface-Instanz befindet bzw. auf welcher Hierarchieebene der Zugriff stattfindet. Im UVM-Cookbook (Mentor Graphics (2013)), S.66 wird empfohlen, die UVM-Konfigurationsdatenbank (uvm_config_db) zu verwenden, um das virtuelle Interface-Handle an die Testbench weiter zu reichen. Der folgende Code setzt das viruelle Interface-Handle auf Top-Level Ebene in die UVM-Konfigurationsdatenbank.

```
uvm_config_db #(virtual <if_name>)::set(null, "uvm_test_top",
    "<entry_lookup_name>", <vif_handle_name>);
```

Als Parameter wird der Konfigurationsdatenbank der Name der Interface-Instanz mit voran gestelltem Schlüsselwort **virtual** übergeben. Der erste Übergabeparameter der set() Methode regelt den hierarchischen Startpunkt des Zugriffs (null entspricht Top-Level). Der zweite Parameter ist ein String mit dem Instanz-Namen für die Zugriffslimitierung (uvm_test_top limitiert den Zugriff auf die Top-Level Ebene, * bedeutet Zugriff ab Top-Level abwärts und agent1* ab Agent1 inkl. Sub-Komponenten). Der nächste Parameter ist das Lookup-Label des Datenbankeintrags. An letzter Stelle steht dann der Name des **zu speichernden Werts** vom Typ #(type <if_name>), in diesem Fall vom Typ virtual.
Die Information über den Ort der Interface-Instanz sowie das virtuelle Interface-Handle werden für jeden Testfall im Basistest über das Konfigurationsobjekt eines Agents an den jeweiligen Agent weiter gereicht. Durch die Verschachtelung der Konfigurationsobjekte unter Berücksichtigung der Hierarchie können diese Informationen gezielt an jeden Ort der Testbench gelangen. Mehr dazu im Abschnitt 2.1.4 über die Konfiguration und den Aufbau der Testumgebung. Die folgende Abbildung verdeutlicht den beschriebenen Sachverhalt.

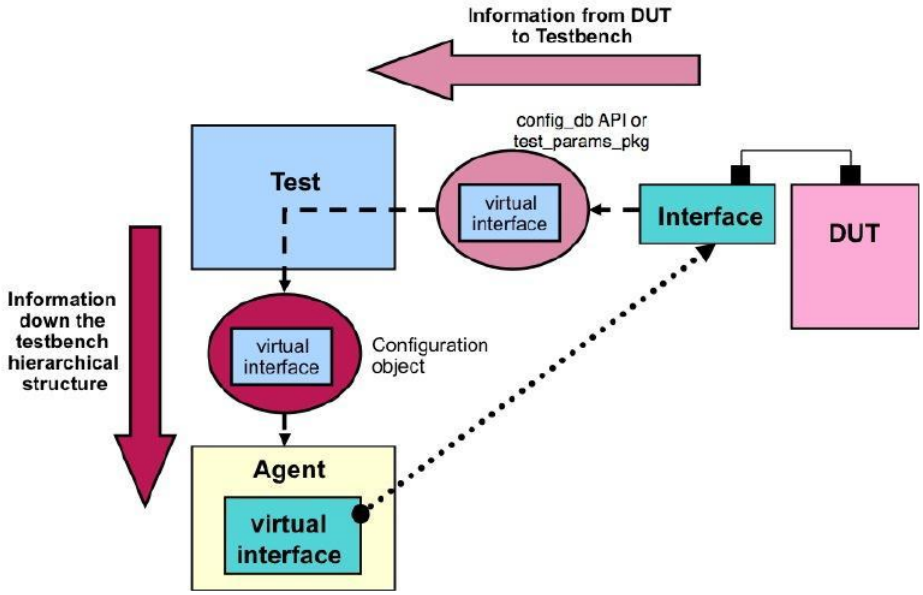

Abbildung 5: TB-DUT Kommunikation - uvm_config_db

Die gestrichelte Linie repräsentiert die geschaffene indirekte (virtuelle) Verbindung zwischen den

Komponenten der Testbench, hier der Agent und den Ports des DUT über die Interface-Instanz. Damit beispielsweise der Agent die Information über den Ort der Interface-Instanz und das Handle erhält, wird im Agent das zugehörige Konfigurationsobjekt aus der Datenbank per get() Methode bezogen, in das zuvor das virtuelle Interface-Handle im Basistest gesetzt wurde.

```
uvm_config_db #(<agent_cfg_obj_name>)::get(this, "",
    "<entry_lookup_name>", <agent_cfg_obj_handle_name>);
```

Wie man sieht, ist der Aufbau identisch zur Variante mit set() Methode, um Einträge in die Datenbank zu setzen. Der Unterschied liegt im vierten Parameter beider Methoden. Bei der get() Methode ist dieser die **Zielvariable** von Typ #(<agent_cfg_obj_name>), hier ein Objekt (uvm_object), der der jeweilige Wert aus der Datenbank mit Lookup-Label <entry_lookup_name> mit identischem Typ zugewiesen wird. Der hierarchische Startpunkt des Zugriffs ist stets „this“, also die aktuelle Ebene (Agent). Eine Zugriffslimitierung ist nicht vorhanden, weshalb der Instanz-Name entfällt.
Ein weiterer Punkt hinsichtlich der Kommunikation zwischen DUT und Testbench ist die Weitergabe von Konfigurationsparametern. Im UVM-Cookbook, S 134 wird empfohlen, ein Testparameter Package für Parameter anzulegen, die sowohl vom DUT als auch von der Testbench verwendet werden. Gemeinsame Parameter liegen somit an einem zentralen Ort. Dadurch wird sichergestellt, das Parameteränderungen nur an einer Stelle stattfinden. Diese wirken sich dann beidseitig auf Testbench und DUT aus, wenn das Package an entsprechender Stelle importiert wird. Einseitige Parameteränderungen werden dadurch vermieden. Eine weitere Möglichkeit neben dem Import des Package ist die Weitergabe der Parameter über die Konfigurationsobjekte der Komponenten auf Basistestebene. Beispielsweise können so VHDL Generics als Parameter zum Treiber eines Agents über dessen Konfigurationsobjekt gelangen.

2.1.4 Konfiguration und Aufbau der Testumgebung

Nicht nur beim Entwurf einer UVM-Testbench ist es ratsam, Komponenten und Module so konfigurierbar und wiederverwendbar wie möglich zu gestalten. Dies wird durch den Gebrauch von Variablen und Parametern mit aussagekräftiger Bezeichnung anstelle direkter Zahlenwerte, wo auch immer möglich, erreicht. Dieser Konvention folgend werden bei UVM Variablen und Parameter zur Konfiguration der Testumgebung systematisch in Konfigurationsobjekten, abgeleitet von der uvm_object Klasse, untergebracht. Folgt man dem UVM-Cookbook (Mentor Graphics (2013)), S. 127 soll jede zu konfigurierende Testbench Komponente ein eigenes Konfigurationsobjekt erhalten. Dieses Objekt beinhaltet Informationen darüber, wie die Komponente selbst aufzubauen ist, welche Funktion sie im Gesamtkontext erfüllen soll und wie Sub-Komponenten, falls vorhanden, zu konfigurieren und miteinander zu verbinden sind. Da eine UVM-Testbench eine klare hierarchische Struktur besitzt, müssen diese Informationen von der obersten bis zur untersten Ebene gelangen. Die Idee dabei ist, Konfigurationsobjekte bzw. Konfigurationen zu verschachteln und die Informationen in der Testbench Hierarchie nach unten weiter zu reichen. Die Weitergabe erfolgt mit Hilfe der UVM-Konfigurationsdatenbank (uvm_config_db API), die bereits im vorherigen Abschnitt gebraucht wurde. In diese können prinzipiell beliebige Objekte unterschiedlich Typs, Variablen, Parameter und natürlich auch virtuelle Interface-Handles und Konfigurationsobjekte (Klasse) per

set() Methode gespeichert und per get() Methode wieder gelesen werden. Sie wird so verwendet, wie bereits im Beispiel mit get() Methode für das Konfigurationsobjekt eines Agents gezeigt.
Auf oberster Ebene der Testbench Hierarchie befindet sich nach dem Top-Level Testbench Modul der Test bzw. die Basistest Klasse. Sie erhält Konfigurationsparameter primär aus dem Testparameter Package oder aus der Konfigurationsdatenbank (virtuelle Interface Handles, die dort vom Top-Level Testbench Modul aus abgelegt wurden). Da sich auf der nächsten Hierarchiebene die Env inklusive aller Sub-Komponenten der Testbench befindet, wird im Test ein verschachteltes Env-Konfigurationsobjekt erzeugt. In dieses fließen in der Regel die per create() Methode aufgebauten Konfigurationsobjekte der Agents und der Env selbst ein. Zunächst werden also die Konfigurationsobjekte erstellt, danach die virtuellen Interface Handles per get() aus der Datenbank bezogen und den zugehörigen Konfigurationsobjekten der Agents zugewiesen. Der nächste Schritt ist jeweils die Verankerung der Agent-Konfigurationsobjekte im Env-Konfigurationsobjekt. Per set() Methode wird abschließend das verschachtelte Objekt in die Konfigurationsdatenbank gesetzt. Am Ende der Build-Phase des Tests besteht nun eine Konfiguration, wie sie in Abbildung 6 beispielhaft dargestellt wird (schwarzer Pfeil).

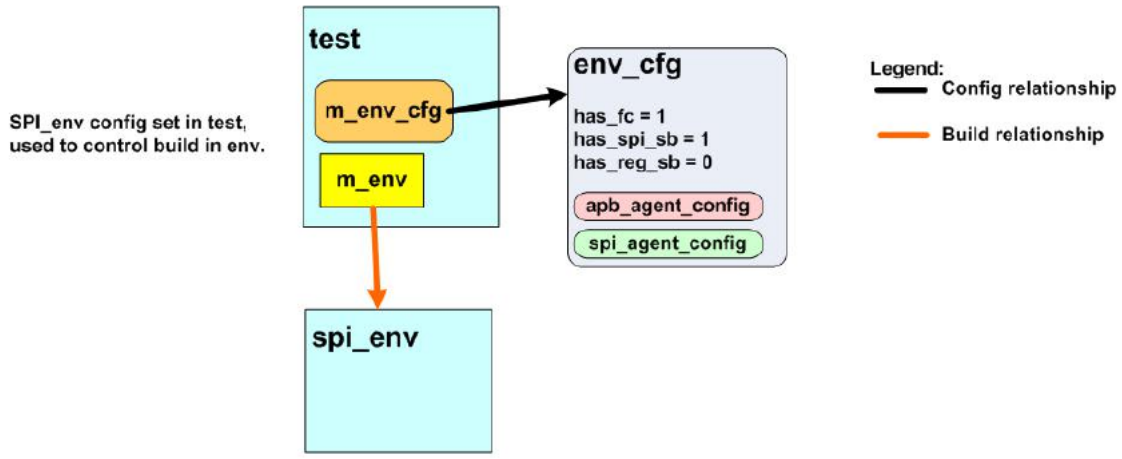

Abbildung 6: Verschachtelung des Env-Konfigurationsobjekts

Das verschachtelte Env-Konfigurationsobjekt enthält nun alle Informationen über die Sub-Komponenten, wie sie aufzubauen, zu verbinden und für den jeweiligen Testfall zu konfigurieren sind. In der Env wird nun während der Build-Phase das Konfigurationsobjekt aus der config_db geholt und basierend auf den darin enthaltenen Informationen benötigte Sub-Komponenten aufgebaut. Für aufzubauende Sub-Komponenten werden dann die Konfigurationsobjekte aus dem Env-Konfigurationsobjekt bezogen und in die config_db gesetzt. Über diese haben die Sub-Komponenten dann ihrerseits per get() Methode Zugriff auf alle notwendigen Variablen und Parameter zum weiteren Aufbau der Testumgebung inkl. Konfiguration.

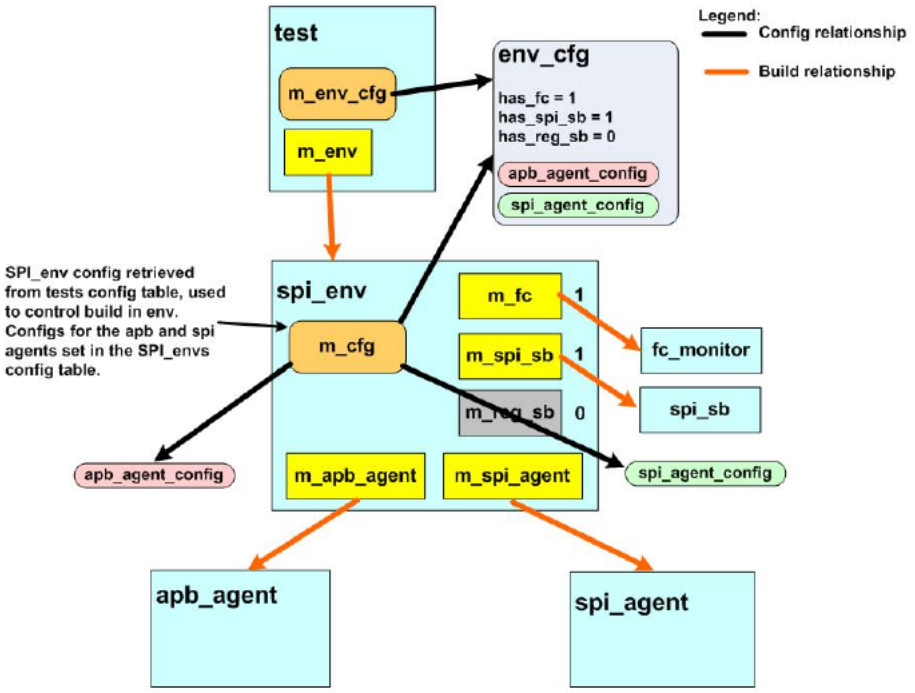

Abbildung 7: Testbench Hierarchie - Weitergabe der Konfigurationsparameter

In der Abbildung 7 ist die Konfiguration der Testbench am Ende der Build-Phase der Env zu sehen. Sie zeigt außerdem beispielhaft, wie Informationen über den Aufbau von Sub-Komponenten in der Hierarchie nach unten weitergereicht werden (schwarze Pfeile).
Sind alle Build-Phasen der Sub-Komponenten abgeschlossen, starten die Connect-Phasen in umgekehrter Reihenfolge, um alle Sub-Komponenten gemäß den Vorgaben zu verbinden. Danach beginnt die eigentliche Simulation mit den Run-Phasen. Die folgende Abbildung zeigt, wie eine Testumgebung zu diesem Zeitpunkt im Regelfall aufgebaut ist. Da ein Driver im passiven Modus auch als Responder fungieren kann, ist eine zusätzliche Verbindungslinie zu sehen.

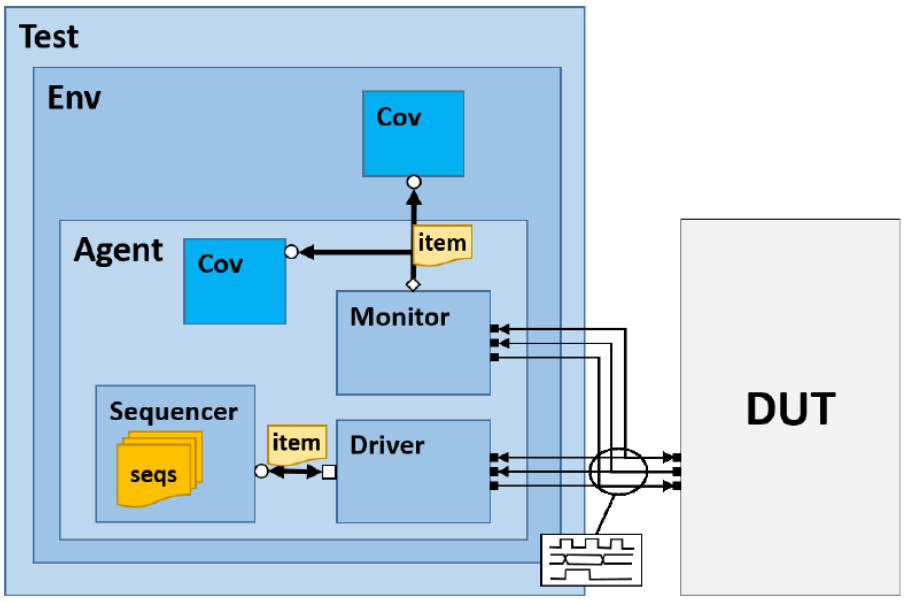

Abbildung 8: Block-Level UVM-Testbench - Testfall Beispiel

Der erweiterte Test wird ausgeführt und die darin definierten Sequences in der gewünschten Reihenfolge gestartet. Die durch die Sequences generierten Sequence Items werden dann über den Sequencer an den Driver weitergeleitet, der diese unter Berücksichtigung zugrunde liegender Protokolle in Pin-Level Transaktionen umwandelt und über das virtuelle Interface zum DUT schickt. Währenddessen beobachtet der Monitor die Pin-Level Aktivitäten auf dem Interface. Er generiert Sequence Items in Abhängigkeit von gesendeten (Request Items, TB $\Rightarrow$ DUT) und empfangenen (Response Items, TB $\Leftarrow$ DUT) Daten. Die Verbindung Sequencer $\Leftrightarrow$ Driver erfolgt über den seq_item_port des Driver und den seq_item_export des Sequencers. Auf TLM-Ebene gelangen die aufgebauten Items des Monitors dann entweder zu Agent-internen, oder über den analysis_port des Agents zu externen Analyse-Komponenten (z.B. Scoreboard).

2.1.5 UVM-Sequences und Sequence Items

In diesem Abschnitt soll ein kurzer Überblick über die wichtigsten Eigenschaften und Funktionen von Sequences und Sequence Items gegeben werden.
UVM-Sequences und Sequence Items sind Objekte, die auf den UVM-Transaktionen basieren (uvm_transactions) und dienen in einer UVM-Testbench zur Erzeugung von Stimuli. Im Gegensatz zu klassischen VHDL/Verilog Testbenches ist die Stimuli-Erzeugung bei UVM, ebenso wie beim strukturellen Aufbau und bei der Konfiguration, objekt-orientiert. Dadurch wird eine wesentlich flexiblere Generierung mit einer Reihe neuer Optionen ermöglicht. Beispielsweise können in SystemVerilog Objekte und deren Inhalte und damit auch Sequences und Sequence Items in definierbaren Grenzen pseudo-randomisiert werden (Stichwort: constrained randomization). Über die Testfälle lassen sich verschiedene Sequences in einer festgelegten Reihenfolge, d.h. sequenziell, aber auch parallel (fork-join) oder in zufälliger Reihenfolge starten. Jede Sequence besitzt einen **body()** Task. Dieser dient einerseits der Erzeugung von Sequence Items, andererseits aber auch zur Erzeugung (**create()** Methode) und zum Ausführen weiterer Sequences (**start()** Methode der Sequence). Somit lässt sich eine hierarchische Struktur mit beliebig vielen Sequences, d.h. mit beliebiger Komplexität, realisieren. Im UVM-Cookbook, S. 190 wird erklärt, dass die Stimuli-Erzeugung generell im erweiterten Test mittels einer einzigen übergeordneten Sequenz gestartet wird. Eine solche Steuersequenz oder auch Top-Level Sequence wird als virtuelle Sequenz (engl. virtual sequence) bezeichnet. Sie erzeugt nicht selbst Stimuli, sondern dient lediglich dazu, den Start weiterer untergeordneter Sequences auf einem oder mehreren Sequencern zu initiieren. Diese Sub-Sequenzen starten wiederum weitere Low-Level Sequences zur Generierung von Stimuli. Damit die Sequences Zugriff auf Testbench-Komponenten haben und die Sequence Items letztendlich zum Driver gelangen, wo ihre Inhalte in Pin-Level Aktivitäten umgewandelt werden, wird der Sequencer benötigt. Die folgende Abbildung zeigt den Datenfluss von einer Sequence bis zum DUT (request) und zurück (response).

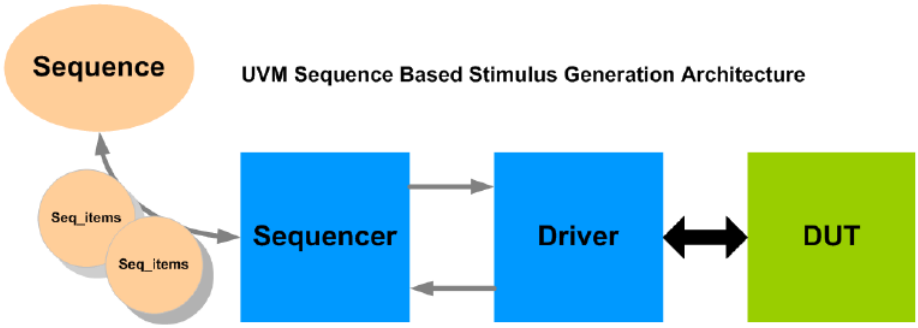

Abbildung 9: Stimuli-Erzeugung per UVM-Sequence

Um eine Sequence zu starten, sind die folgenden Schritte durchzuführen: Im erweiterten Test wird die Sequence zunächst den UVM-Factory Regeln entsprechend per create() Methode erstellt. Es folgt die Konfiguration der Sequence. Hier werden z.B. Startwerte festgelegt, Variablen zur Bestimmung der Anzahl zu generierender Sequence Items definiert und Felder des Sequence Items per randomize() Methode randomisiert. Beim Start der Sequence per **start()** Methode wird zunächst festgelegt, auf welchem Sequencer sie auszuführen ist. In UVM ist es möglich, Sequences auf mehreren Sequencern zu starten. Als nächstes wird der **body()** Task aufgerufen, der eine weitere Kette von Ereignissen, wie oben beschrieben, auslöst. Wie generierte Sequence Items schließlich über den Sequencer zum Driver gelangen, wird später erläutert. Man bezeichnet Sequences als transiente Objekte, da ihre Existenz während der Simulationszeit begrenzt ist. Daher können sie jederzeit erstellt und wieder verworfen werden. Weiterhin besteht die Möglichkeit, mehrere Sequences gleichzeitig über einen Driver laufen zu lassen. Für diesen Fall besitzt UVM Algorithmen (Methoden), die willkürlich entscheiden, welches Sequence Item welcher Sequence als nächstes transferiert wird, da nur ein Item zur Zeit Zugriff auf den Driver haben kann.
Sequence Items sind ebenfalls transiente Objekte. Sie stellen die kleinste Einheit zur Erzeugung von Stimuli dar und sind die elementaren Bestandteile aus denen sich eine Sequence zusammen setzt. Das Sequence Item muss daher alle Daten enthalten, die der Driver benötigt um notwendige Pin-Level Transaktionen durchzuführen. Da sie zum transferieren von Daten sowohl in Sende- als auch Empfangsrichtung (request/response) verwendet werden, unterscheidet man entsprechend Request- bzw. Response Items. Sie bauen letztendlich auf ein und demselben Sequence Item auf, nutzen bei ihrer Verwendung aber unterschiedliche Felder (Variablen) des Items. Um Response Items einem zugehörigen Request Item zuordnen zu können und erstere zurück zur ursprünglichen Sequenz zu leiten, besitzen beide die gleiche Sequence-ID. Im UVM Cookbook sind auf S. 193 Informationen über die Inhalte der Felder eines Sequence Items zu finden. Hinsichtlich der Erzeugung zufälliger Stimuli wird die Vereinbarung getroffen, Variablen, die randomisiert werden sollen, mit dem Schlüsselwort **rand** zu deklarieren. Diese finden dann bei den Request Items Verwendung. Variablen zur Aufnahme empfangener Daten (response) sind ohne dieses Schlüsselwort zu deklarieren. Um Einschränkungen bei der Erzeugung pseudo-randomisierter Stimuli zu bewirken, kann ein Sequence Item SystemVerilog constraints enthalten. So werden gewisse Grenzen für Zahlenwerte festgelegt, oder nur ein Teil eines Arrays randomisiert. Dazu zwei einfache Beispiele:

```
// Beispiel 1 - Variable null_cnt kann nur Zufallswerte im Bereich
// 1 bis 150 annehmen

constraint c_null_cnt {null_cnt inside {[1:150]};}
```

```
// Beispiel 2 - Drei Bits des Arrays tcode_tx sind stets 0

constraint c_tcode_tx {tcode_tx[8] == '{0};
                       tcode_tx[1] == '{0};
                       tcode_tx[0] == '{0};
                       }
```

Weitere Bestandteile eines Sequence Items sind Methoden, die gewisse Funktionen zur Bearbeitung von Datenvariablen implementieren. Zu diesen gehören eine do_copy(), do_compare(), convert2string(), do_print() und do_record() Funktion. Welche Funktionen diese im einzelnen implementieren, ist der folgenden Tabelle zu entnehmen.

Tabelle 1: Methoden eines Sequence Items und ihre Funktion

Sequence Item Methode	Funktion
do_copy()	Erstellen einer Kopie (deep copy) bestimmter Variablen
do_compare()	Vergleich zweier Datenobjekte (bzw. Variablen) gleichen Typs
convert2string()	Darstellung der Werte von Variablen als formatierter String
do_print()	Ausgabe des formatierten Strings
do_record()	Unterstützt Darstellung eines Datenobjekts als Transaktion

Damit der Transfer von Sequence Items über den Sequencer zum Driver (request) und auch zurück (response) funktioniert, muss eine bidirektionale Verbindung beider Komponenten bestehen. Ein Sequencer kommuniziert immer nur mit einem Driver.

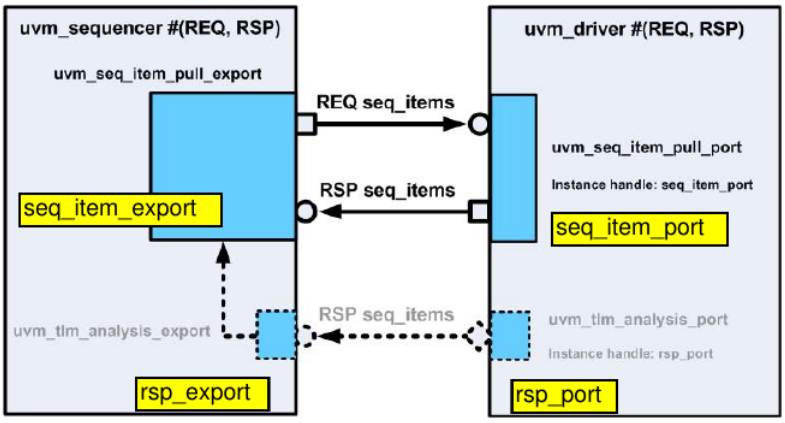

Abbildung 10: Bidirektionale Verbindung zwischen Sequencer und Driver

Gemäß Abbildung 10 ist der uvm_seq_item_pull_port des Drivers mit dem uvm_seq_item_pull_export des zugehörigen Sequencers zu verbinden. Die unidirektionale Verbindung (uvm_tlm_analysis_port zu uvm_tlm_analysis_export) wird nicht mehr benötigt. Die gelben Label in der Grafik beschreiben die Namen der zugehörigen Handles.
Nach dem Start der Sequence werden in deren body() Task Sequence Items entsprechend der Konfiguration (Anzahl, etc.) generiert. Für jedes erzeugte Sequence Item wird jetzt ein Handshake-Protokoll zur Kommunikation mit dem Driver durchgeführt. Dazu folgt der Aufruf der **start_item()** Methode mit dem Handle des Sequence Items als Übergabeparameter. Die Methode blockiert, bis die Verbindung zum Driver besteht. Das Sequence Item bzw. die darin enthaltenen **rand** Variablen werden dann gemäß der Vorgabe randomisiert und das Item abgeschickt. Zuletzt blockiert die **finish_item()** Methode dann, bis der Driver das Item verarbeitet und seinen Teil des Protokolls erfüllt hat. Seitens des Drivers läuft das Protokoll wie folgt ab: Per Aufruf der **get_next_item()** oder **try_next_item()** Methode wird versucht, das nächste Request Item über den Sequencer zu beziehen. get_next_item() blockiert, bis ein Sequence Item vorhanden ist. Bei try_next_item() handelt es sich um eine nicht blockierende Variante der ersten Methode. Sie liefert als Rückgabeparameter einen Null-Pointer, sofern zum Zeitpunkt des Aufrufs kein Sequence Item zur Verfügung steht. Mit der abschließenden nicht blockierenden **item_done()** Methode wird das Handshake-Protokoll abgeschlossen. Der erneute Aufruf von get_next_item() oder try_next_item() (falls der vorherige try_next_item() Aufruf erfolgreich war) vor dem Aufruf von item_done() verletzt das Protokoll und stellt einen Fehler dar. Der erneute Aufruf von item_done() vor einem get_next_item() oder try_next_item() schließt das Handshake-Protokoll ab, ohne ein Response Item an den Sequencer zu schicken. Weiterhin sind im Sequence Item die peek(), get() und put() Methoden vorhanden. Im UVM Cookbook wird von der Verwendung dieser Methoden abgeraten. Das soeben erläuterte Handshake-Verfahren ist in der folgenden Abbildung zum besseren Verständnis dargestellt.

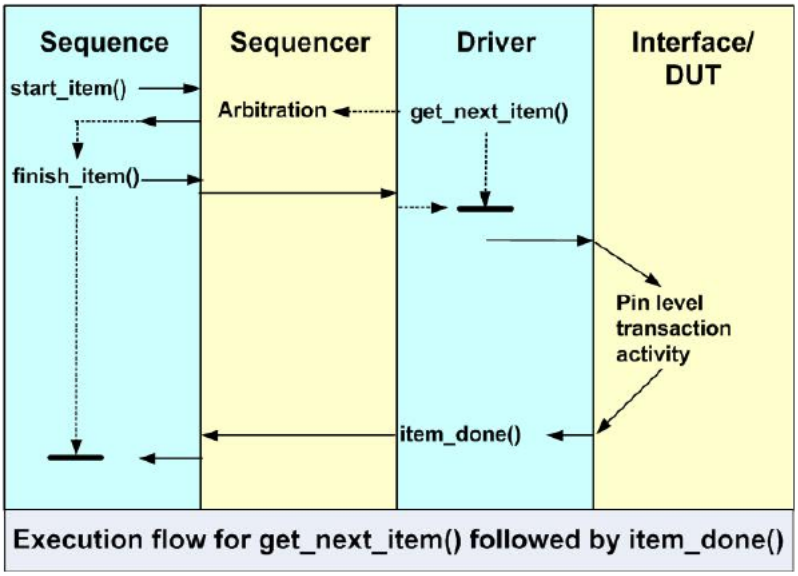

Abbildung 11: Sequence Item Handshake-Protokoll zwischen Sequence und Driver

Das Handshake-Protokoll beginnt mit dem Aufruf von start_item() in der Sequence. Diese

Methode blockiert, bis der Treiber per get_next_item() oder try_next_item() das nächste Item über den Sequencer bezieht. finish_item() blockiert dann so lange, bis der Driver das zuvor per get_next_item() oder try_next_item() bezogene Item verarbeitet, d.h. die geforderten Pin-Level Transaktionen ausgeführt hat und seinerseits das Protokoll per item_done() abschließt. Zwischen start_item() und finish_item() in der Sequence dürfen keine Aktionen stattfinden, die Simulationszeit beanspruchen.

2.2 SpaceWire ECSS-E-ST-50-12C Standard

Der ECSS E-ST-50-12C Standard (Version 31. Juli 2008) ist die Grundlage für den Entwurf der im Rahmen dieser Abschlussarbeit entwickelten UVM-Testbench für SpaceWire IP-Cores. Obwohl der Standard schon einige Jahre existiert, ist er laut Internetseite der ECSS unter der Rubrik „Active Engineering standards“ immer noch aktuell. Zwar existiert eine dritte Auflage (Revision) mit der Bezeichnung ECSS E-ST-50-12C Rev1 vom November 2014, allerdings ist diese bislang noch nicht offiziell veröffentlicht worden. Darin enthaltene mögliche Änderungen, Erweiterungen oder Verbesserungen des Standards wurden daher nicht berücksichtigt. Die folgenden Abschnitte sollen einen Überblick über Inhalte des SpaceWire Standards geben, die für die Entwicklung der Testbench relevant und außerdem für das Verständnis von SpaceWire notwendig sind.

2.2.1 Übersicht

Der Standard ist im Kern in die für die Definition von SpaceWire maßgeblichen Protokollebenen (engl. protocol levels) unterteilt. Diese sind in ihrer Bezeichnung an die Schichten (Layer) des OSI-Referenzmodells angelehnt. Zu den Protokollebenen des Standards gehören die physikalische Ebene (Physical level), die Signalebene (Signal level) die Zeichen- oder Tokenebene (Character level), die Vermittlungsebene (Exchange level), die Paketebene (Packet level) sowie die Netzwerkebene (Network level). Auf physikalischer Ebene, die für die Entwicklung der UVM-Testbench keinerlei Bedeutung hat, sind SpaceWire Kabel und deren Leiter, Verbinder und deren Belegung, die Kabelmontage und der Verlauf von Leiterbahnen auf Leiterplatten (PCBs) und Backplanes spezifiziert. Das Kapitel Signal level enthält Definitionen bezüglich der Signalcodierung, zum Verfahren zur Übertragung von SpaceWire Signalen (LVDS), zu Datenübertragungsraten und zum Signal-Skew und -Jitter und Informationen zu deren Auswirkung bei unterschiedlichen Übertragungsraten. Im Kapitel Character level sind die bei SpaceWire zur Daten- und Paketübertragung, zur Datenflusssteuerung und zur Übertragung von Time-Codes (Zeitgeber) verwendeten SpaceWire Token definiert. Auf diese wird im Kapitel 2.2.3 näher eingegangen. Weiterhin wird hier das Verfahren zur Erkennung von Paritätsfehlern erläutert, da es eng mit der Definition der Token verbunden ist. Anschließend ist noch das Bit-Erkennungsmuster des firstNULL Tokens festgelegt, welches zum Handshake-Protokoll der Link-Initialisierung gehört. Am Ende sind noch Informationen bezüglich der Codierung von Token im Zusammenhang mit dem Interface zwischen Host und Sender/Empfänger des SpaceWire Links und dem Time-Interface zwischen Host und Sender/Empfänger enthalten. Auf der Vermittlungsebene (Kapitel Exchange Level) sind die Protokolle zur Link-Initialisierung, zur Datenflusssteuerung, zur Fehlererkennung und das Protokoll für den Fehlerfall, das den Umgang mit fehlerhaft übertragenen Daten bzw. Paketen

und die Re-Initialisierung des Links regelt, definiert. Token werden kategorisiert, ein beispielhaftes Blockdiagramm eines SpaceWire Link Interfaces inklusive Signalbezeichnungen dargestellt und die Funktion der einzelnen Blöcke detailliert beschrieben. Weiterhin wird das Zustandsdiagramm der Link-FSM abgebildet. Die einzelnen Zustände sowie die Bedingungen für Zustandsübergange werden erläutert. Abschließend sind Ausnahmebedingungen, die bei der Link-Initialisierung oder bei der Datenübertragung auftreten können (z.B. falls nur eines der beiden SpaceWire Signale übertragen wird) und Vorgaben zur Verteilung der Systemzeit über ein SpaceWire Netzwerk spezifiziert. Das nächste Kapitel (Packet level) gibt Auskunft darüber, wie bei SpaceWire Daten zur Übertragung über einen bestehenden Link in SpaceWire Pakete aufzuteilen sind. Die letzte Protokollebene des Standards ist die Netzwerkebene. Im zugehörigen Kapitel Network level wird die Struktur eines SpaceWire Netzwerks, das darin verwendete Verfahren (wormhole routing) zur Datenflusskontrolle und das Verfahren, das festlegt wie Pakete über ein SpaceWire Netzwerk vom Sender zum Empfänger übertragend werden, definiert. Außerdem ist hier die Vorgehensweise im Falle eines Link-Fehlers oder eines Fehlers auf Netzwerkebene erklärt.
Das zuletzt beschriebene Kapitel hat für die Entwicklung der UVM-Testbench ebenfalls keine Bedeutung, da sie zur direkten Kommunikation (P2P) mit einem SpaceWire IP-Core ausgelegt wurde und daher SpaceWire Pakete, die zusätzliche Routing-Informationen enthalten, nicht unterstützt bzw. erkennt. Die Testbench ist im Prinzip ein Endpunkt in einem SpaceWire Netzwerk, also eine Schnittstelle zu einem Hostystem und eine Quelle von SpaceWire Paketen, auch als SpaceWire Node bezeichnet. Als Endpunkt kommuniziert sie über eine direkte bidirektionale Verbindung (SpaceWire Link) mit einem anderen Endpunkt. Der als geprüftes und verlässliches DUT verwendete SpaceWire Light IP-Core besitzt ebenfalls keine Routing-Fähigkeiten.

2.2.2 Data-Strobe (DS) Codierung

Wie bereits erwähnt, nutzt SpaceWire das LVDS-Verfahren zur Übertragung von Signalen. Ein SpaceWire Link besitzt zwei Signalleitungen (Data (D) und Strobe(S)) je Senderichtung. Weil Daten im Vollduplex-Modus übertragen werden können, also ein gleichzeitiges Senden und Empfangen möglich ist, existieren zwei Data-Strobe Paare, d.h. vier Signalleitungen pro Link. Bei LVDS wird ein Signal als Differenzsignal übertragen, wozu zwei Signalleitungen (+/-) nötig sind.

$$2 \text{ x } (\underbrace{\mathrm{D}^+, \mathrm{D}^-}_{\text{Paar 1}}, \underbrace{\mathrm{S}^+, \mathrm{S}^-}_{\text{Paar 2}})$$

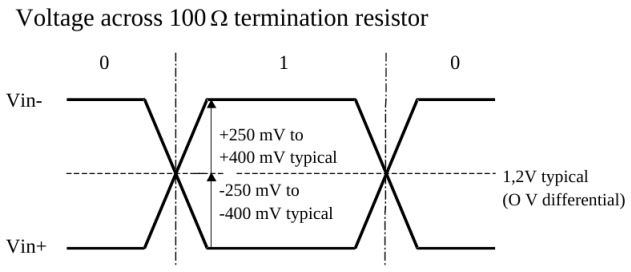

Abbildung 12: Datenübertragung per LVDS - Spannungsverlauf

Es sind daher jeweils zwei Differenzsignal-Paare je Senderichtung vorhanden. Dies entspricht vier Paaren im Vollduplex-Modus und damit insgesamt acht Signalleitungen pro Link. Abbildung 12 zeigt den typischen Spannungsverlauf bei einer Datenübertragung mittels LVDS. Das Differenzsignal bzw. die Differenzspannung wird aus den beiden Spannungen Vin- und Vin+ gebildet, die wenige hundert Millivolt betragen. In Abhängigkeit des Logikpegels des zu übertragenden Signals ändert sich die Polarität von Vin- und Vin+. Dadurch ändert sich ebenfalls die Polarität der Differenzspannung. Eine negative Differenzspannung entspricht einer logischen 0 und umgekehrt eine positive Differenzspannung einer logischen 1.
Das Data-Strobe Codierungsschema sieht folgendermaßen aus. Das digitale Data-Signal wird unverändert übertragen. Eine 0 des zu codierenden Bitstreams wird als logische „0“ und eine 1 als logische „1“ übertragen. Das Strobe-Signal wird als Funktion des Data-Signals S = f(D) übertragen. Der Wert des Strobe-Signals ändert sich immer genau dann, falls sich das Data-Signal von einem auf den nächsten Takt nicht ändert. In diesem Fall folgte auf eine logische „0“ also eine weitere 0 oder auf eine logische „1“ eine weitere 1. Wird eine Serie von Nullen oder Einsen übertragen, ändert Strobe seinen Wert bei jeder Taktflanke. Dieser Sachverhalt wird durch die folgende Abbildung verdeutlicht.

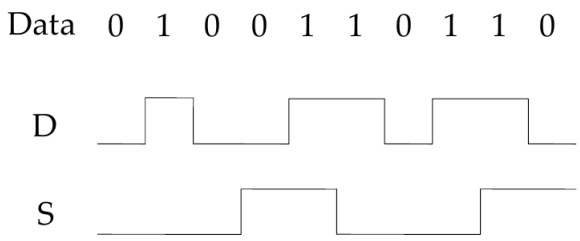

Abbildung 13: SpaceWire Data-Strobe Codierungsschema

Durch die DS-Codierung ändern sich die Werte von Data und Strobe theoretisch nie gleichzeitig. Bei einer realen Datenübertragung, bei der beispielsweise Signal-Skew und Jitter eine Rolle spielen, kann ein gleichzeitiger Wechsel jedoch durchaus vorkommen. Diese ungewollte Verfälschung von Daten soll laut SpaceWire Standard nicht zur Blockierung des Receivers führen. Stattdessen wird früher oder später ein Paritätsfehler im Receiver detektiert und der Link neu aufgebaut. Nach einem Fehler oder Reset werden Data und Strobe zurück gesetzt. Ist danach das erste gesendete Bit eine 0, folgt der erste Signalwechsel auf der Strobe-Leitung. Data und Strobe dienen außerdem durch die logische XOR-Verknüpfung beider Signale zur Taktrückgewinnung (Rx Clock). Der resultierende Takt ist das Maß für die Geschwindigkeit der ankommendenden Daten , da die Taktfrequenz von der aktuellen Übertragungsrate abhängt.

2.2.3 SpaceWire Token und Flow Control

Zur Übertragung von Paketen und zur Steuerung des Datenflusses zwischen zwei SpaceWire Nodes sowie zur Zeitsynchronisation in einem SpaceWire Netzwerk sind im SpaceWire Standard Zeichen (Token) definiert. Diese sind bestimmte elementare Bitsequenzen bzw. Bitmuster mit einem Steuerbit (data-control flag), das bei der Dekodierung zur Identifikation eines Tokens dient. Die Token sind in drei Kategorien eingeteilt, data characters, control characters und control codes. Bei

den data characters handelt es sich um Token, die Bytes repräsentieren. Die control characters, von denen es vier gibt, dienen entweder der Steuerung des Datenflusses (flow control), als Indikator für das Ende eines Pakets bzw. den Anfang eines neuen Pakets und gleichzeitig für ein fehlerfrei oder fehlerhaft übertragenes Paket oder zur Bildung weiterer Codes. Die control codes, von denen es zwei gibt, werden entweder zum Versenden von Time-Codes oder zur Aufrechterhaltung eines Links genutzt. Die folgende Abbildung gibt eine Übersicht über die SpaceWire Token.

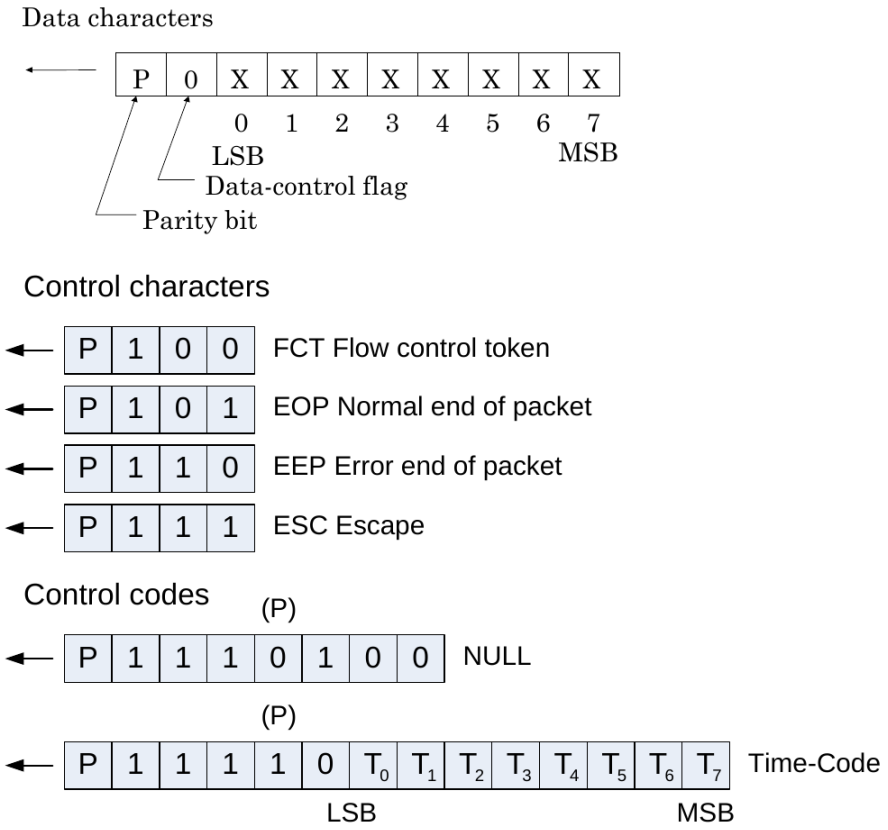

Abbildung 14: SpaceWire data characters, control characters und control codes

Die Pfeile geben die Sendereihenfolge an. Jedes Token beginnt mit einem Paritätsbit. Mehr zum Thema Paritätsbit, Paritätsfehler und parity coverage im Abschnitt 2.2.6. Das zweite Bit ist stets das data-control flag eines Tokens. Eine logische 0 identifiziert das Token als data character und eine logische 1 als control character. Der verbleibende Teil des 10 Bit breiten data characters ist das Datenbyte. Die beiden letzten Bits eines control characters (control bits) werden zur Codierung verwendet. „00“ repräsentiert das FCT (Flow control Token) zur Datenflusssteuerung, „01“ das EOP, „10“ das EEP und „11“ das ESC. EOP und EEP markieren ein fehlerfrei bzw. fehlerhaft übertragenes Paket. ESC ist das Escape-Token, das lediglich zur Bildung der control codes verwendet wird. Die control codes sind das NULL Token und der Time-Code. Das NULL Token setzt sich aus einem ESC gefolgt von einem FCT zusammen. Das Paritätsbit des FCT ist in diesem Zusammenhang wegen der Paritätsart und der parity coverage stets 0. NULL ist Bestandteil der NULL-Erkennungssequenz (NULL detection sequence) bei der Link-Initialisierung und dient außerem der Aufrechterhaltung eines Links, sofern darüber gerade keine Daten (Pakete, control characters oder Time-Codes) gesendet werden. Der Time-Code wird aus einem ESC gefolgt von einem data character gebildet.

Das Paritätsbit des data characters ist wegen der Art der Parität und der parity coverage stets 1. Datenbytes werden bei SpaceWire mit dem niedrigwertigsten Bit zuerst (LSB first) übertragen. Die beiden Bits mit der höchsten Stellenwertigkeit sind für zukünftige Versionen des Standards reserviert und daher auf 0 zu setzen. Die sechs verbleibenden Bits, T_0 bis T_5, stellen den Wert des Time-Codes dar. Zwei weitere Begriffe, die im Standard auftauchen, sind Link characters (L-Chars) und Normal characters (N-Chars). Die L-Chars sind die beiden control characters FCT und ESC sowie die control codes (NULL und der Time-Code). Zu den N-Chars gehören die verbleibenden control characters EOP und EEP sowie data characters. Die N-Chars bildet also die SpaceWire packages. Für den Fall, dass empfangene oder zu sendende Daten in Form von Steuerbits, gefolgt von einem Datenbyte über das Interface eines Hostsystems transferiert werden, empfiehlt der SpaceWire Standard, ECSS-E-ST-50-12C (2008), S. 55 folgendes Codierungsschema zur Darstellung von EOP oder EEP als Byte.

Tabelle 2: Codierung von Rx- und Tx-Daten bzgl. Hostsystem Interface

Steuerbit	Codierte Datenbits (MSB ... LSB)	Bedeutung
0	xxxxxxxx	Datenbyte
1	00000000	EOP
1	00000001	EEP

Dieses Codierungsschema wird auch für Datenbytes, EOPs und EEPs in den beiden Rx- und Tx-FIFOs des SpaceWire Light IP-Cores und später in der UVM-Testbench zum Senden von Daten über das FIFO-Interface angewandt.
Im Folgenden wird die Funktionsweise der Datenflusssteuerung (Flow Control) bei SpaceWire und die damit verbundene Funktion des FCT im Detail beschrieben. Im Allgemeinen dient die Datenflusssteuerung dem Management des Empfangsbuffers eines Receivers beim Empfang von data characters/Paketen über einen Link bzw. innerhalb eines Netzwerks und soll den Überlauf des Buffers verhindern. Der Buffer ist entweder Bestandteil eines Endpunkts (Node) oder eines SpaceWire Routers. Das FCT wird als Indikator für freien Speicherplatz im Buffer gesendet. Im Standard ist definiert, dass ein gesendetes FCT einem freien Speicherplatz für acht N-Chars (data character, EOP oder EEP) im Empfangsbuffer entspricht. Ist aktuell mehr freier Speicherplatz im Buffer vorhanden, können auch mehrere FCTs gesendet werden. Ein empfangenes FCT erlaubt im Umkehrschluss das Senden von acht weiteren N-Chars. Um die Anzahl gesendeter bzw. empfangener FCTs zu erfassen, sind zwei Zähler vorhanden, einer für empfangene FCTs (Tx credit count im Sender) und einer für gesendete FCTs (Rx credit count im Empfänger). Die Zählerwerte repräsentieren also sozusagen Guthaben. Der Bezeichnung entsprechend dient der aktuelle Wert des Tx credit count der Freigabe zum Senden von N-Chars bzw. der Wert des Rx credit count als Indikator, wie viele N-Chars empfangen werden können, bzw. wie viele zu erwarten sind. Da ein FCT dem Speicherplatz für acht N-Chars entspricht, wird der Tx credit count für jedes empfangene FCT und der Rx credit count für jedes gesendete FCT um 8 erhöht. Ein gesendetes N-Char verringert

das Guthaben zum Senden von N-Chars um 1 (Tx credit count -1). Der Empfang eines N-Chars verringert das Guthaben zum Empfangen von N-Chars (Rx credit count -1).
Der Maximalwert beider Zähler ist 56, was sieben FCTs bzw. dem Speicherplatz für 56 N-Chars entspricht. Im Falle eines leeren Empfangsbuffers, der Speicherplatz für mehr als 56 N-Chars besitzt, sollen daher nie mehr als sieben FCTs auf einmal gesendet werden. Dies ist zum Beispiel während der Link-Initialisierung möglich, denn beide Zählerwerte sind nach einem Reset oder nach einem Empfängerfehler, der zum Abbau des Links führt, auf 0 zu setzen. Das Inkrementieren oder Dekrementieren der Zählerwerte bei bestehender Verbindung kann natürlich dazu führen, dass diese ihren Minimal- oder Maximalwert erreichen. Ist der Tx credit count aktuell 0, so ist kein Guthaben zum Senden von N-Chars vorhanden. In diesem Fall soll der Sender so lange mit dem Senden weiterer N-Chars warten, bis ein FCT empfangen wird, dadurch der Tx credit count wieder um acht erhöht wird und ein weiteres Senden möglich ist. Ist der Wert des Rx credit count aktuell 49 oder größer, darf wegen seines Maximalwerts kein weiteres FCT gesendet werden. Das Senden eines oder mehrerer FCTs ist erst dann wieder möglich, falls durch gesendete N-Chars der Rx credit count aktuell kleiner als 49 ist. Es gibt zwei Szenarien, in denen der Tx credit count über seinen Maximalwert bzw. der Rx credit count in den negativen Bereich gesetzt werden könnte. Hierbei handelt es sich um einen Fehler, der im Abschnitt 2.2.6 beschrieben wird.
Die aktuellen Werte der Zähler sind entscheidende Kriterien dafür, ob ein N-Char oder ein FCT gesendet werden kann. Die Entscheidung darüber, welches Token als nächstes gesendet wird, ist letztendlich von der im Standard definierten Priorität abhängig. Die höchste Priorität haben die zeitgebenden Time-Codes. Mit sinkender Priorität folgen dann die FCTs, N-Chars und NULLs. Gibt es eine Anfrage zum Senden eines Time-Codes, wird dieser unmittelbar nach dem nächsten Token gesendet. Ist es aktuell nicht erlaubt FCTs oder N-Chars zu senden oder sind keine N-Chars im Tx Buffer vorhanden, wird ein NULL Token gesendet.

2.2.4 SpaceWire Link-FSM

Die State Machine ist neben Transmitter und Receiver das zentrale Modul zur Steuerung eines SpaceWire Link-Interfaces. Die Eingangssignale bestimmen die Zustandsübergänge in der Link-FSM. Die Ausgangssignale steuern Transmitter und Receiver. Die Abbildung 15 zeigt ein beispielhaftes Blockdiagramm eines SpaceWire Interfaces. Die Namen der Eingangssignale entsprechen denen im Zustandsdiagramm der State Machine. Der Timer-Block repräsentiert einen Zeitgeber zur Generierung zweier Timeouts (6,4 μs und 12,8 μs). Diese sind laut Standard toleranzbehaftet (5,82 μs - 7,22 μs bzw. 11,64 μs - 14,33 μs). Die Transmit Clock bestimmt die Datenübertragungsrate. Der Receive Clock Recovery Block dient zur Taktrückgewinnung. Die restlichen Signale sind mögliche Interface-Signale des Hostsystems, beispielsweise Steuersignale für den Sende- und Empfangsbuffer, Signale des Zeit-Interfaces oder Datensignale. Die Ausgangssignale der State Machine schalten den Transmitter oder Receiver ein oder aus (Enable_Tx bzw. Enable_Rx), setzen sie per Reset zurück, oder erteilen die Freigabe zum Senden von N-Chars, NULLs, FCTs oder Time-Codes. Die Eingangssignale sind die Flags des Receivers zur Anzeige empfangener Tokens oder Signale des Hostsystems und des Timers.

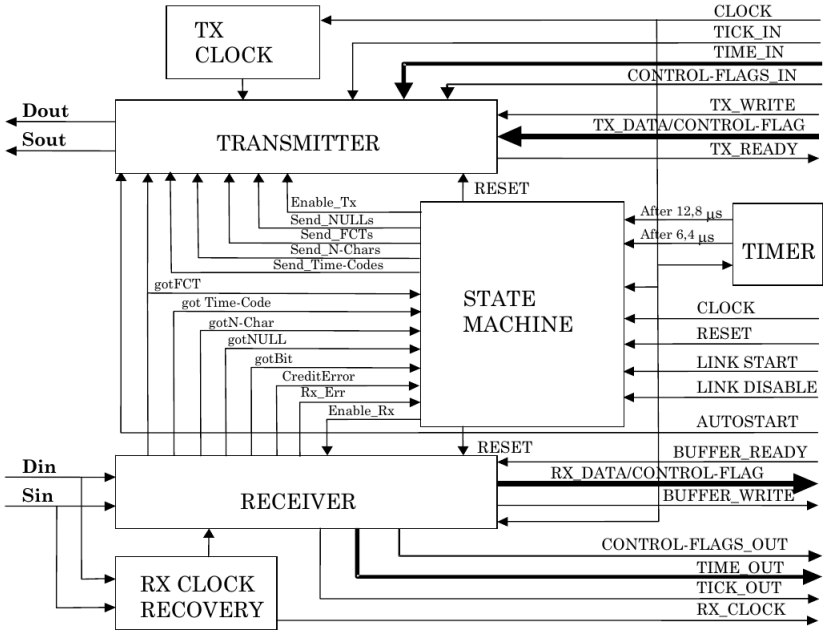

Abbildung 15: Blockdiagramm eines SpaceWire Link-Interfaces

Die Link-FSM besitzt sechs Zustände: ErrorReset, ErrorWait, Ready, Started, Connecting und Run. Die ersten zwei bzw. fünf Zustände sind Bestandteil des „Exchange of silence“ Protokolls bzw. des Protokolls zur Link-Initialisierung (NULL/FCT Handshake). Der Zustand Run beschreibt eine aktive Verbindung (Link). Die folgende Abbildung zeigt das Zustandsdiagramm der State Machine inklusive der Bedingungen für Zustandsübergange.

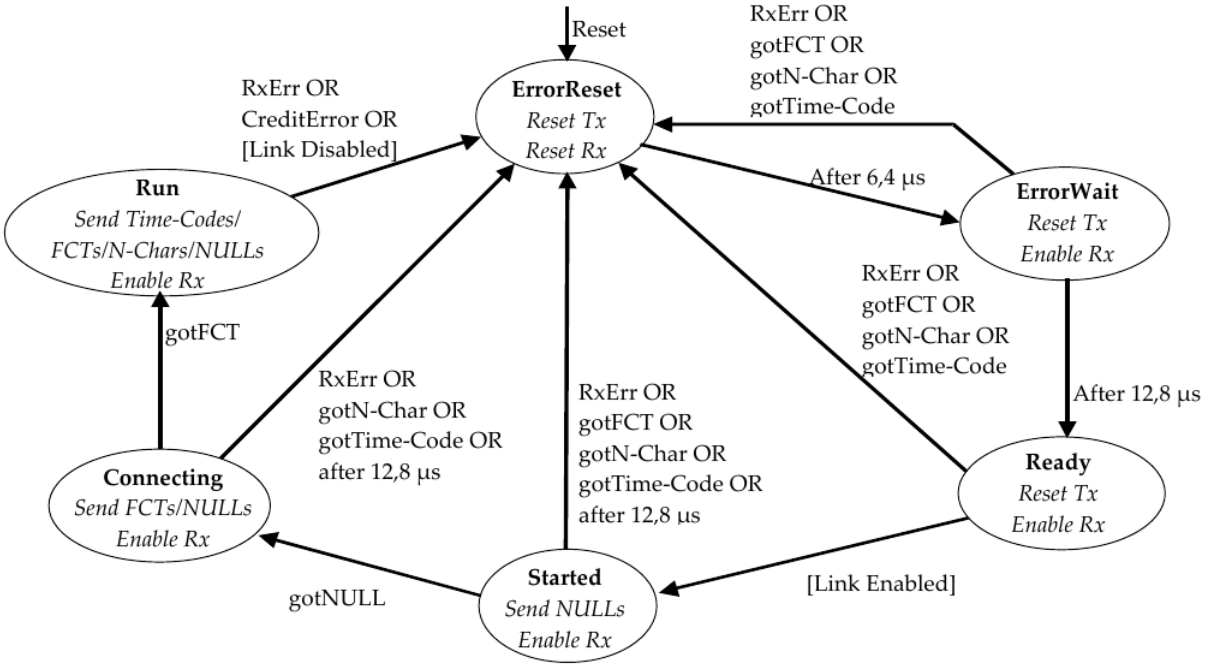

Abbildung 16: Zustandsdiagramm der SpaceWire Link-FSM

In den Zustand ErrorReset wird aufgrund eines Link-Resets gewechselt. Dieser Reset wird entweder durch ein systemseitiges Reset-Signal, durch ein Signal zum Beenden einer aktiven Verbindung

([Link Disabled]), durch einen Fehler während der Initialisierung (Zustände ErrorWait, Ready, Started oder Connecting) oder durch einen Fehler während einer bestehenden Verbindung (Zustand Run) ausgelöst. Durch das Reset-Signal kann die State Machine unmittelbar in den ErrorReset Zustand versetzt werden, die dort mindestens so lange, wie das Signal aktiv ist, verweilt. Ist es nicht mehr aktiv, startet das 6,4 μs Timeout. Wird in diesen Zustand gewechselt, sind Transmitter und Receiver zu deaktivieren und sie bleiben deaktiviert, so lange das Reset-Signal aktiv ist. In den Zustand ErrorWait wird bedingungslos nach den 6,4 μs gewechselt. Im Zustand ErrorWait wird der Receiver aktiviert. Auf diese Weise kann bereits hier oder in einem der Folgezustände Ready oder Started das gotNULL Flag durch den Empfang eines NULL Tokens gesetzt und ein Verbindungsabbruch erkannt werden. In den Folgezustand Ready wird bedingungslos nach 12,8 μs gewechselt. Falls in ErrorWait ein Fehler der Kategorie I (die drei Kategorien sind weiter unten erläutert) während des 12,8 μs Timeouts auftritt, wird in den Zustand ErrorReset gewechselt. Die FSM verweilt im Zustand Ready, so lange kein Fehler auftritt und das [Link Enabled] Signal nicht aktiv ist. Tritt ein Fehler der Kategorie I auf, wird den in den Zustand ErrorReset gewechselt, ansonsten in den Zustand Started, sobald [Link Enabled] aktiv ist. Im Zustand Started wird zusätzlich der Transmitter aktiviert und aktiv versucht, einen Link aufzubauen. Dies geschieht durch das Senden von NULL Tokens. Falls ein Fehler der Kategorie I auftritt, oder 12,8 μs vergehen, ohne dass ein NULL Token empfangen und dadurch das gotNULL Flag gesetzt wurde (das Flag hätte bereits in den beiden vorherigen Zuständen durch den Empfang eines NULLs gesetzt werden können), wird in den Zustand ErrorReset gewechselt. In den Zustand Connecting wird gewechselt, sobald gotNULL gesetzt ist. Im Zustand Started ist vor diesem Wechsel mindestens ein NULL Token zu senden. Im Zustand Connecting wird mit dem Senden von FCTs begonnen. Falls ein Fehler der Kategorie II auftritt, oder 12,8 μs vergehen, ohne dass ein FCT empfangen und dadurch das gotFCT Flag gesetzt wurde, wird in den Zustand ErrorReset gewechselt. In den Zustand Run wird gewechselt, sobald gotFCT gesetzt ist. Im Zustand Connecting ist vor diesem Wechsel mindestens ein FCT zu senden. Man hat hier unter Berücksichtigung des Maximalwerts des Rx credit counts die Möglichkeit, initial maximal sieben FCTs zu senden, wenn man voraussetzt, dass am anderen Ende des Links kein Empfangsbuffer vorhanden ist, also empfangene Daten nicht gepuffert werden. Siehe dazu S. 63 des SpaceWire Standards. Falls nach den sieben gesendeten FCTs bislang kein FCT empfangen wurde, werden bis Ablauf des 12,8 μs Timeouts NULLs gesendet, sofern zwischenzeitlich kein Fehler auftritt. Im Zustand Run ist dann zusätzlich das Senden und Empfangen von Time-Codes und Paketen (N-Chars) möglich, da der Link in beide Richtungen aufgebaut ist. Falls ein Fehler der Kategorie III auftritt, wird wieder in den Zustand ErrorReset gewechselt.
Die Tabelle 3 listet die Bedeutung der einzelnen Flags im Zustandsdiagramm sowie in den Fehlerkategorien auf. Abschließend folgt die Definition der Fehlerkategorien I - III durch ihre Logikgleichungen.

Empfängerfehler:

$$\text{RxErr} = \text{errdisc} \vee \text{errpar} \vee \text{erresc}$$

LE (LinkEnable):

$$[\text{Link Enabled}] = \neg[\text{Link Disabled}] \wedge ([\text{LinkStart}] \vee ([\text{AutoStart}] \wedge \text{gotNULL}))$$

Tabelle 3: Flags des FSM-Zustandsdiagramm und der Fehlerkategorien und ihre Bedeutung

Flag	Bedeutung
[Link Enabled]	Link-Aufbau starten
[Link Disabled]	Bestehende Verbindung beenden / keine neue Verbindung aufbauen
[LinkStart]	Anfrage zum Aufbau eines Links
[AutoStart]	Option zum automatischen Aufbau eines Links
gotFCT	FCT empfangen
gotN-Char	N-Char (data character, EOP oder EEP) empfangen
gotTime-Code	Time-Code empfangen
gotNULL	NULL empfangen
RxErr	Empfängerfehler (Receiver error)
FB	Erstes Bit nach Reset empfangen
errdisc	Disconnect error (RxErr)
errcred	Credit error
errpar	Parity error (RxErr)
erresc	Escape error (RxErr)

Fehlerkategorie I:

$$FK\_1 = (FB \wedge errdisc) \vee gotNULL \wedge (errpar \vee erresc \vee gotFCT \vee gotN\text{-}Char \vee gotTime\text{-}Code)$$

Fehlerkategorie II:

$$FK\_2 = RxErr \vee gotN\text{-}Char \vee gotTime\text{-}Code$$

Fehlerkategorie III:

$$FK\_3 = RxErr \vee errcred \vee [Link\ Disabled]$$

Die Flags [Link Disabled], [LinkStart] und [AutoStart] können software- oder hardwareseitig über das Interface des Hostsystems gesetzt werden. Laut Logikgleichung für [Link Enabled] hat das [Link Disabled] Flag, sofern es gesetzt ist, Vorrang vor [AutoStart] und [LinkStart]. Mit [AutoStart] lässt sich alternativ zu [LinkStart] ein einseitiger Verbindungsaufbau, von einer Seite des Links ausgehend, realisieren. Falls [AutoStart] auf der gegenüber liegenden Seite gesetzt ist, wird hier so lange mit dem Verbindungsaufbau gewartet, bis die andere Seite NULLs sendet.

2.2.5 Link-Initialisierung und Protokolle

Wie bereits im vorherigen Abschnitt erwähnt, sind alle Zustände der SpaceWire Link-FSM, mit Ausnahme des Zustands Run, Bestandteil von zwei essentiellen Protokollen, die die Kommunikation bei SpaceWire steuern. Das erste Protokoll ist das NULL/FCT Protokoll zur Initialisierung eines Links. Es handelt sich dabei um ein Handshake-Protokoll zum Aufbau der bidirektionalen Verbindung zwischen zwei SpaceWire Nodes zur Übermittlung von Daten und Steuerinformationen. Das zweite Protokoll ist das sogenannte „Exchange of Silence“ Protokoll. Es beschreibt die bilaterale und

gegenseitige (auf beiden Seiten des Links) Erkennung von Fehlern, die bei SpaceWire definiert sind, sowie das Verfahren zur Link Re-Initialisierung. Das Protokoll ist außerdem eng verbunden mit zu ergreifenden Maßnahmen im Fehlerfall (Stichwort error recovery procedure, siehe Kapitel 2.2.7), um Datenverlust zu vermeiden. Diese Maßnahmen stellen im Prinzip ein weiteres Protokoll dar. Während der Link-Initialisierung wird bei SpaceWire stets mit einer festen Datenübertragungsrate von 10 MBit/s gesendet. Die Transmitter sind also beispielsweise mit 10 MHz zu takten. Danach ist theoretisch jede Übertragungsrate innerhalb der im Standard definierten Grenzen, von 2 MBit/s bis 400 MBit/s, einstellbar.
Zur Erklärung der Vorgänge während des „Exchange of Silence“ sowie des NULL/FCT Handshake-Protokolls folgt eine Abbildung aus dem SpaceWire Standard. Hierbei handelt es sich um ein beispielhaftes Szenario.

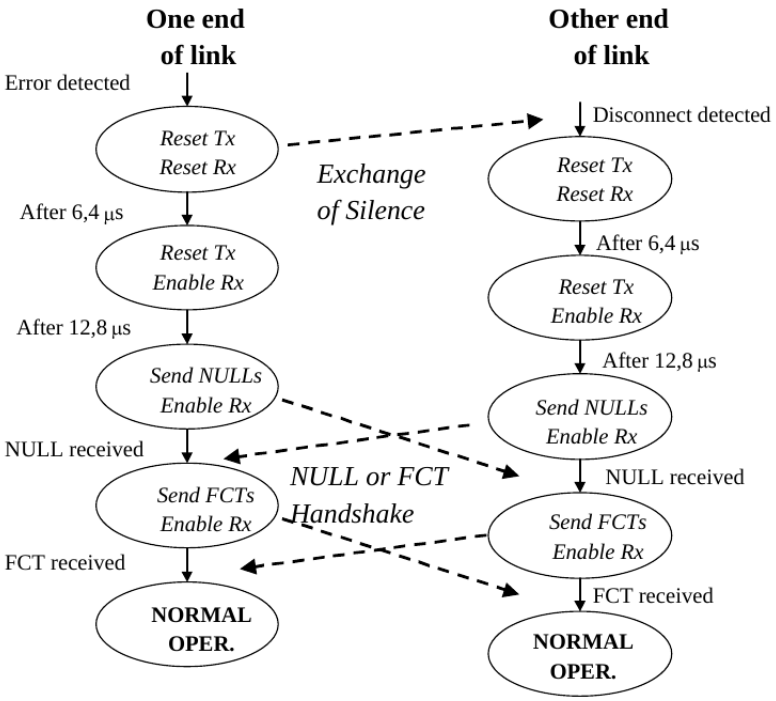

Abbildung 17: „Exchange of Silence“ und NULL/FCT Handshake-Protokoll

Man stelle sich das Zustandsdiagramm der Link-FSM und die Zustände (ErrorReset, ErrorWait, Ready, Started, Connecting und Run) von zwei Teilnehmern vor. Man nehme außerdem an, dass bereits eine bidirektionale Verbindung zwischen zwei Teilnehmern A und B besteht. Beide State Machines befinden sich daher im Zustand Run. Im normalen Betrieb werden Daten und Steuerinformationen gesendet und empfangen. Während der Übertragung wird vom Teilnehmer (A) entweder ein Fehler detektiert oder auf dieser Seite der Link explizit durch das [Link Disabled] Flag beendet (Fehlerkategorie III). Die State Machine dieses Teilnehmers wechselt in den Zustand ErrorReset, deaktiviert Transmitter und Receiver und stoppt die Datenübertragung. Die Deaktivierung des Transmitters bewirkt ein Zurücksetzen von Data und Strobe (siehe dazu Abschnitt 6.3.2 des Standards). Auf der gegenüber liegenden Seite des Links (B) bewirkt dies kurze Zeit später einen Empfängerfehler (Disconnect error) im Receiver. Die State Machine dieses Teilnehmers wechselt ebenfalls in den Zustand ErrorReset und deaktiviert Transmitter und Receiver. Auch hier stoppt die Datenübertragung. Es folgt eine beidseitige Sendepause. Die zuletzt genannten

Vorgänge beschreiben das „Exchange of Silence“ Protokoll. Beide Teilnehmer versuchen jetzt nach ca. 19,2 μs (12,8 μs + 6,4 μs) wieder einen Link mit der gegenüberliegenden Seite aufzubauen. Diese Wartezeit soll sicherstellen, dass beide Teilnehmer (nach einem Fehler) wieder bereit sind Tokens zu empfangen. Während dieser Zeit, in der beide State Machines die Zustände ErrorReset, ErrorWait und Ready durchlaufen, sind die Transmitter beider Teilnehmer deaktiviert.
Die folgenden Vorgänge sind Bestandteil des NULL/FCT Handshake Protokolls zum Link-Aufbau. Nach einem Fehler oder Reset befindet sich eine der beiden State Machines (A) zuerst im Zustand ErrorReset. Nach ca. 6,4 μs wird der Receiver wieder aktiviert. Ab dem Zustand ErrorWait wird mit der Detektierung eines NULL Tokens begonnen, um das gotNULL Flag zu setzen. Die Detektierung dieses ersten NULLs nach einem Fehler oder Reset, auch als firstNULL bezeichnet, ist der erste Teil des Handshake-Protokolls. Das gotNULL Flag ist genau dann zu setzen, wenn die sogenannte NULL detection sequence erkannt bzw. empfangen wurde. Diese Sequenz besteht aus dem NULL Token und einem weiteren Paritätsbit. Zur Berechnung des Paritätsbits siehe Abschnitt 2.2.6.

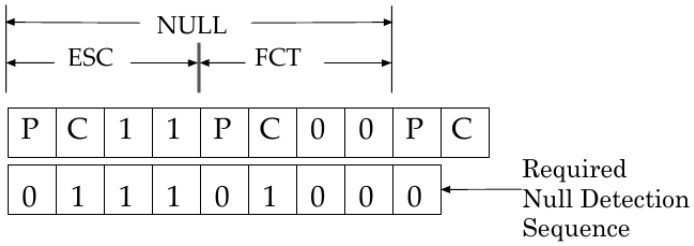

Abbildung 18: SpaceWire NULL detection sequence

Nach der Erkennung der Bitsequenz 011101000 wird mit der Token-Decodierung fortgefahren. Ist die Bedingung zum Aufbau eines Links ([Link Enabled] Flag gesetzt) erfüllt, befindet sich die State Machine eines Teilnehmers (A) schließlich als erstes im Zustand Started, beginnt mit dem Senden von NULLs und wartet auf den Empfang des ersten NULLs. Auf der gegenüberliegenden Seite (B) wird das erste NULL spätestens im Zustand Started detektiert und gotNULL gesetzt. Die State Machine (B) befindet sich einige Zeit später ebenfalls im Zustand Started und beginnt mit dem Senden von NULLs. Falls gotNULL bereits in einem vorherigen Zustand gesetzt wurde, ist hier mindestens ein NULL zu senden, um dem anderen Teilnehmer (A) die Erkennung zu ermöglichen. Dieser erkennt daraufhin das NULL im Zustand Started, wechselt in den Zustand Connecting und beginnt mit dem Senden von FCTs. Die gegenüberliegende Seite (B) empfängt ein gesendetes NULL von (A), wechselt n den Zustand Connecting und beginnt seinerseits mit dem Senden von FCTs. Die Seite (A) empfängt dann, dem zeitlichen Verlauf dieses Beispiels entsprechend, zuerst das von (B) gesendete FCT und wechselt in den Zustand Run. Auch (B) empfängt einige Zeit später das von (A) gesendete FCT und wechselt ebenfalls in den Zustand Run. Durch das hier beschriebene NULL/FCT Handshake-Protokoll ist die beidseitige Verbindung zu diesem Zeitpunkt sichergestellt. Der normale Betrieb wird fortgesetzt.
Das dargestellte Beispiel beschreibt nur eines von diversen möglichen Szenarien und damit zeitlichen Abläufen. Die Abläufe variieren, je nachdem in welchem Zustand sich die State Machine der gegenüberliegenden Seite aktuell befindet. Dadurch können einige Ausnahmesituationen entstehen, die ab S. 78 (Kapitel 8.10) im SpaceWire Standard beschrieben sind. Auf diese wird hier aber nicht näher eingeganen, da sie letztendlich entweder zur Detektierung eines Fehlers und

damit zur Re-Initialisierung des Links oder zur erfolgreichen Re-Initialisierung im zweiten Versuch führen.

2.2.6 Fehlerarten und Fehlererkennung

SpaceWire definiert insgesamt acht Fehlerarten. Zu diesen gehören die Network level errors, Link errors, Receiver errors, Parity errors, Escape errors, Disconnect errors, Credit errors und Character sequence errors.
Ein **Network level error** ist ein Fehler auf Netzwerk- bzw. Paketebene. Dazu gehört der Link error, der Empfang eines EEPs sowie eine ungültige Zieladresse im Paketheader.
Der **Receiver error**, auch als RxErr bezeichnet, ist per Definition entweder ein Disconnect error, Parity error, oder Escape error. Diese Art von Fehler wird dem Namen entsprechend im Receiver erkannt.
Ein **Link error** ist per Definition ein Fehler auf Vermittlungsebene (Exchange level). Zu dieser Fehlerart gehören die Parity errors, Escape errors, Disconnect errors, Credit errors und Character sequence errors.
Der **Parity error** signalisiert ein falsches Paritätsbit. Bei SpaceWire soll das Paritätsbit ungerade Parität erzeugen. Wie bereits gezeigt, besitzt jedes SpaceWire Token ein Paritätsbit. Dieses deckt jedoch nicht das Token selbst, sondern das vorherige Token, d.h. das zuvor gesendete Token ab (man beachte die Sendereihenfolge der Bits). Dies ist durch die Paritätsabdeckung, also diejenigen Bits, die zur Bestimmung der Parität relevant sind, begründet. Ein empfangenes Token ist folglich immer genau dann gültig, wenn das Paritätsbit im nachfolgenden Token ungerade Parität erzeugt. Die Abbildung verdeutlicht den dargestellten Sachverhalt.

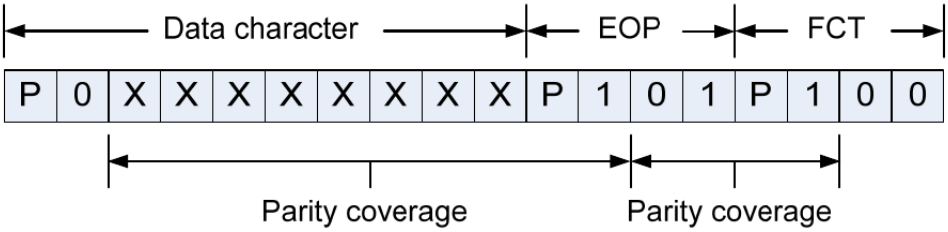

Abbildung 19: SpaceWire Parity coverage

Die Parität deckt immer das Datenbyte eines data characters oder die control bits eines control characters inklusive dem Paritätsbit und dem data-control flag des nachfolgenden Tokens ab. In der Abbildung folgt auf das EOP ein FCT. Das Paritätsbit des FCTs ist daher laut Definition richtigerweise auf 1 zu setzen, um ungerade Parität zu erzeugen. Das EOP wäre damit gültig. Ist das Paritätsbit 0, liegt hinsichtlich der Paritätsabdeckung gerade Parität und damit ein Parity Error vor. Das EOP ist dann ungültig. Die Erkennung von Paritätsfehlern im Receiver wird aktiviert, sobald das erste NULL nach einem Reset oder Fehler empfangen wird (gotNULL Flag gesetzt). Dies zeichnet sich auch in der Logikgleichung FK_1 (Fehlerkategorie I) ab. Bei der NULL detection sequence ist das Paritätsbit des NULLs auf 0 zu setzen. Es ist das erste zu empfangende Bit nach einem Reset/Fehler und bewirkt den ersten Signalwechsel auf der Strobe-Leitung.
Ein **Escape error** wird im Receiver genau dann erkannt, wenn nach dem Empfang eines gültigen

ESC ein weiteres gültiges ESC, EOP oder EEP empfangen wird. Das ESC Token dient lediglich der Bildung eines NULL Token (ESC gefolgt von FCT) oder eines Time-Codes (ESC gefolgt von einem data character). Nach einem ESC folgt also entweder ein FCT oder ein data character. Ansonsten liegt ein Fehler vor. Die Erkennung eines Escape errors wird mit dem Empfang des ersten NULL nach einem Reset/Fehler aktiviert.
Der **Disconnect error** dient der Erkennung eines Verbindungsabbruchs. Er wird im Receiver genau dann erkannt, wenn nach dem Empfang eines gültigen Bits kein neues gültiges Bit innerhalb eines Zeitfensters von 0,85 μs empfangen wird. Dieses Timeout ist toleranzbehaftet und kann zwischen 0,727 μs und 1,0 μs liegen. Die Erkennung eines Disconnect errors wird mit dem Empfang des ersten Bits nach einem Reset/Fehler aktiviert. Dies zeichnet sich wieder in der Logikgleichung FK_1 ab.
Der **Credit error** bezieht sich auf die zuvor genannten Zählerwerte des Tx credit counts bzw. Rx credit counts bei der Datenflusssteuerung. Er wird im Receiver genau dann erkannt, wenn beim Empfang eines FCT der Wert des Tx credit count aktuell größer als 48 ist. In diesem Fall würde der Wert über seinen Maximalwert von 56 steigen. Dieses Szenario kann z.B. auftreten, wenn Übertragungsfehler durch das Paritätsbit nicht erkannt werden und dadurch fälschlicherweise ein FCT anstelle des ursprünglich gesendeten Tokens decodiert wird. Ein Credit error wird außerdem beim Empfang eines N-Chars erkannt, falls der Wert des Rx credit counts aktuell 0 ist. Das bedeutet, N-Chars sind aktuell nicht zu erwarten, da alle N-Chars, entsprechend zuvor gesendeter FCTs, bereits empfangen wurden. Credit errors sind im Allgemeinen in undetektierten Fehlern begründet. Die Erkennung eines Credit errors wird aktiviert, sobald der Link in beide Richtungen besteht (Zustand Run). Siehe dazu auch die Logikgleichung FK_3.
Ein **Character sequence error** ist ein Fehler, der das NULL/FCT Handshake-Protokoll verletzt. Er wirkt sich daher nur während der Link-Initialisierung aus. Zu Beginn der Initialisierung soll der Receiver alle N-Chars und L-Chars bis auf NULL ignorieren, da sie mit der Erkennung des NULLs beginnt. Ist diese detektiert, so stellt der Empfang eines FCTs vor dem Senden eines NULLs laut Protokoll einen Fehler dar. Ebenso stellt der Empfang von N-Chars und Time-Codes bevor sowohl ein NULL als auch ein FCT empfangen wurde einen Fehler dar, da diese nur bei bestehender Verbindung zu senden sind. Der Character sequence error wird durch die State Machine der Link-FSM abgefangen. Dies wird durch die beiden Logikgleichungen der Fehlerkategorien I und II deutlich.

2.2.7 Link Error Recovery Schema

Das Link Error Recovery Schema beschreibt die zu ergreifenden Maßnahmen im Fehlerfall und wird im Kapitel 11 des SpaceWire Standards erläutert. Diese stellen eine Reihe von Vorkehrungen zur Vermeidung von Datenverlusten dar. Die im vorherigen Abschnitt erläuterten Fehler führen in jedem Fall zum Abbruch der Verbindung und zur anschließenden Link Re-Initialisierung, sofern [Link Enabled] auf beiden Seiten des Links gesetzt ist. Im SpaceWire Standard, S. 108, sind folgende Maßnahmen definiert.

1. Fehlererkennung (Disconnect error, Parity error, Escape error, Credit error, Character sequence error)
2. Abbruch der Verbindung („Exchange of silence“ Protokoll)
3. Bzgl. des Empfangsbuffers/Rx-FIFO: Falls das zuvor empfangene N-Char kein EOP war, füge hier ein EEP am Ende ein
4. Bzgl. des Sendebuffers/Tx-FIFO: Lösche die verbleibenden data characters des aktuell zu sendenden Pakets inklusive dem zugehörigen EOP
5. Link Re-Initialisierung
6. Übertragung des nächsten Pakets

Für den Fall, dass in Punkt 3. das zuvor empfangene N-Char ein EOP oder EEP war, kann ein EEP eingefügt werden, was allerdings nicht notwendig ist. Ein weiteres empfangenes EOP oder EEP nach einem EOP oder EEP entspricht einem leeren Paket. Das zweite EOP oder EEP wird dann verworfen. Das eingefügte EEP beendet das aktuelle Paket vorzeitig und markiert es als nicht vollständig bzw. fehlerhaft übertragenes Paket. In Punkt 4. wird der Rest des zu sendenden Pakets verworfen. Dies verhindert das Senden eines unvollständigen Pakets nach der Link Re-Initialisierung. Falls der Empfangsbuffer bzw. das Rx-FIFO zum Zeitpunkt des Fehlers voll war, kann keine Link Re-Initialisierung erfolgen, so lange nicht Platz für mindestens 9 N-Chars vorhanden ist (Platz für EEP + 8 N-Chars, der für das Senden eines FCTs im Rahmen des Handshake-Protokolls bzw. zum Einfügen des EEP erforderlich ist). Die folgende Abbildung aus dem Standard veranschaulicht die beschriebenen Maßnahmen.

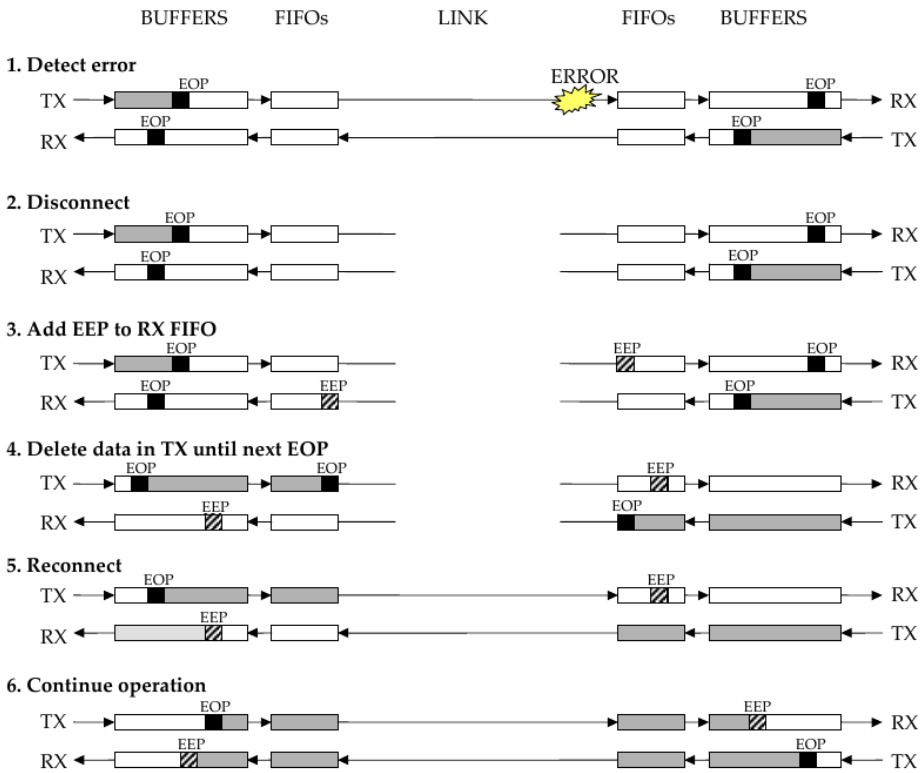

Abbildung 20: SpaceWire Link Error Recovery Schema

Was mit dem unvollständig bzw. fehlerhaft übertragenen Paket im Empfangsbuffer/Rx-FIFO geschieht, ist im Standard nicht definiert. In Abhängigkeit der Wichtigkeit der darin enthaltenen Informationen wird es entweder verwendet oder verworfen.

2.3 SpaceWire Light IP-Core

2.3.1 Übersicht

Der SpaceWire Light IP-Core ist das DUT, welches in der in dieser Arbeit entwickelten UVM-Testbench eingesetzt wird. Dabei handelt es sich um einen synthesefähigen und auf gängigen FPGAs implementierbaren VHDL IP-Core. Er implementiert den SpaceWire Codec (Transmitter/Receiver), besitzt ein einfaches FIFO-Interface und wurde auf Basis des ECSS-E-ST-50-12C SpaceWire Standards vom niederländischen Softwareingenieur Joris van Rantwijk entwickelt. Der SpaceWire Light gilt als freie Software und ist über die GNU General Public License (GPL) lizenziert. Das Ziel bei der Entwicklung dieses IP-Cores war es, eine verlässliche Implementierung des SpaceWire Codes in Übereinstimmung mit dem Standard zur Verfügung zu stellen. Laut SpaceWire Light Dokumentation, van Rantwijk (2013), bietet der IP-Core keine weiteren Features wie z.B. das Routing von Paketen innerhalb eines SpaceWire Netzwerks oder den Fernzugriff auf den Speicher anderer SpaceWire Nodes zur Konfiguration per RMAP. Der SpaceWire Light IP-Core dient als Testobjekt, da er laut Dokumentation bereits auf verschiedenen FPGAs (Xilinx Spartan-3 und Virtex-5) über eine Verbindung zu einem kommerziellen SpaceWire Produkt getestet wurde und somit als zuverlässig gilt. Er ist allerdings weder für kritische Anwendungen geeignet noch wurde er auf strikte Übereinstimmung mit dem Standard getestet.
Obwohl die Entwicklung des IP-Cores abgeschlossen und er voll funktionsfähig ist, befindet er sich laut Entwicklungsstand noch in der Beta-Phase. Aktuell (Anfang Dez. 2018) sind im Bugtracker auf der offiziellen Projektseite keine Bugs gemeldet. Der SpaceWire Light IP-Core kann nach kostenloser Registrierung über die Internetseite von Open Cores unter der Projekt-Rubrik Communication controller frei herunter geladen werden (https://opencores.org/projects/spacewire_light).

2.3.2 Konzept

2.3.2.1 Transmitter

Der Transmitter des SpaceWire Light IP-Cores sendet bei aktiver Verbindung (bestehender Link) Data-Strobe codierte SpaceWire data characters, control characters und control codes. Der IP-Core besitzt zwei verschiedene Implementierungen des Transmitters. Die erste nutzt ausschließlich den Systemtakt (clk) zum Senden von Daten und wird als allgemeine Implementierung (**generic** implementation) bezeichnet. Die maximal mögliche Datenübertragungsrate entspricht hier der Frequenz des Systemtakts. Die aktuelle Übertragungsrate ist durch einen Taktteiler, dessen Wert durch ein 8 Bit breites Interface-Signal dargestellt wird, jederzeit einstellbar. Dies entspricht einem Integerwert von 0 bis 255. Der Wert 0 teilt den Systemtakt nicht, der Wert 1 halbiert ihn, usw.. Hierbei ist zu beachten, dass die minimale Bitrate bei SpaceWire 2 MBit/s beträgt. Diese soll die Erkennung eines Disconnect errors sicherstellen, für den ein Zeitfenster von 0,85 μs festgelegt

ist ($\mathrm{T} = \frac{1}{\mathrm{f}} = \frac{1}{2\,\mathrm{MHz}} = 0,5\,\mu\mathrm{s}$ pro Bit). Die zweite Implementierung nutzt zum Senden von Daten einen separaten Takt (txclk), der wesentlich höher als der Systemtakt sein kann. Dadurch sind wesentlich höhere Übertragungsraten möglich. Die maximale Rate entspricht der Frequenz des separaten Takts. Der Takt kann maximal dem Fünffachen des Systemtakts entsprechen. Die Übertragungsrate ist bei dieser Implementierung also maximal um den Faktor 5 höher. Sie wird schnelle Implementierung (**fast** implementation) genannt. Auch der separate Takt lässt sich, wie oben beschrieben, über dasselbe Signal teilen und die Übertragungsrate in Senderichtung einstellen. Der Faktor 5 ergibt sich aus der notwendigen Synchronisation des Datentransfers zwischen den beiden Taktdomänen bei der Verwendung der schnellen Implementierung, da der separate Takt lediglich zur Umsetzung der Tokens in Data-Strobe Signale im schnellen Transmitter verwendet wird. Die zu sendenden Tokens werden jedoch über den Systemtakt getaktet zur Übermittlung in einen Buffer gesetzt, aus dem mit dem separaten Takt gelesen wird. Der Faktor 5 stellt sicher, das der Buffer nie leer wird. In diesem Fall wäre der SpaceWire Light gezwungen, zwischenzeitlich NULLs in den Bitstream einzufügen, obwohl zu sendende Daten vorhanden sind.

2.3.2.2 Receiver

Der Receiver des SpaceWire Light IP-Cores decodiert die über den Bitstream bei aktiver Verbindung empfangenen SpaceWire data characters, control characters und control codes. Er besitzt ein Front-End zur Detektierung gültiger Data-Strobe codierter Bits, die von dort aus zur Decodierung weiter geleitet werden. Der IP-Core besitzt zwei verschiedene Implementierungen des Receiver Front-Ends, die beide auf dem Prinzip des synchronen Oversamplings basieren. Beim synchronen Oversampling werden Samples des Data- und Strobe-Signals synchron zu einem Takt mit hoher Frequenz genommen. Die Samples dienen dann der Erkennung gültiger Signal-Übergänge auf der Leitung, bzw. der Erkennung gültiger Bits. Ein weiteres Verfahren nutzt zum Sampling der Signale den aus Data und Strobe durch XOR-Verknüpfung zurück gewonnen Takt. Dieses wird jedoch im SpaceWire Light nicht verwendet. In der Regel ist die Samplingfrequenz wesentlich höher als die Rate ankommender Bits. Samples sollten mit mindestens der doppelten Frequenz genommen werden, um eine fehlerfreie Decodierung der Bits zu ermöglichen. Die erste Implementierung nutzt ausschließlich den Systemtakt (clk) zum Sampling von Data und Strobe und wird wieder als allgemeine (**generic**) Implementierung bezeichnet. Zur fehlerfreien Decodierung darf die Bitrate daher maximal der halben Frequenz des Systemtakts entsprechen, bei einem Systemtakt von 100 MHz also maximal 50 MBit/s auf Empfängerseite. Die zweite Implementierung des Receiver Front-Ends nutzt zum Sampling einen separaten Takt (rxclk), der wesentlich höher als der Systemtakt sein kann und wird schnelle (**fast**) Implementierung genannt. Dadurch sind wesentlich höhere Bitraten ankommender Daten möglich. Die maximale Rate entspricht der Frequenz des separaten Takts, die wiederum maximal dem Vierfachen des Systemtakts entsprechen kann. Data und Strobe Samples werden bei dieser Variante zu beiden Taktflanken des separaten Takts genommen und gültige Bits in Einheiten von 1 - 4 Bits (rxchunk) zur Decodierung weiter geleitet. Daraus ergibt sich der Faktor 4.

2.3.3 IP-Core Interface

Der SpaceWire Light IP-Core besitzt zwei Interfaces. Die erste Instanz nennt sich Spwstream und implementiert den SpaceWire Codec mit einem einfachen FIFO-basierten RAM Block Interface. Die Instanz besteht aus dem Receiver, dem allgemeinen oder schnellen Receiver Front-End, dem allgemeinen oder schnellen Transmitter, der SpaceWire Link-FSM sowie dem Tx- und Rx-FIFO, die aus Dual-Port RAM Blöcken bestehen. Beide FIFOs speichern 9 Bit breite Datenwörter (Datenbyte inkl. einem Bit zur Unterscheidung von Datenbytes und den End-of-Packet Markern EOP und EEP). Die zweite hier nicht weiter relevante Instanz Spwamba implementiert den SpaceWire Codec mit einem AMBA (AHB und APB) Interface. Das APB Interface dient zur Konfiguration und Steuerung des IP-Cores über Register und zum Auslesen von Statusinformationen. Über ein AHB Interface werden Paketdaten im DMA Modus übertragen. Laut SpaceWire Light Dokumentation ist diese Instanz prädestiniert für den Einsatz im Zusammenhang mit einem LEON3 Prozessor.

2.3.3.1 Funktionsweise

Das Spwstream Interface besitzt zwei Dual-Port RAM Blöcke als FIFOs, die zum Transfer empfangener bzw. zu sendender SpaceWire Pakete dienen. Die Tiefe der beiden FIFOs ist per VHDL Generics einstellbar. Das Rx-FIFO hat eine Tiefe im Bereich von 64 - 16384 Bytes, das Tx-FIFO im Bereich von 4 - 16384 Bytes. Zur weiteren Erklärung der Funktionsweise sei in der folgenden Abbildung das Blockdiagramm der Spwstream Instanz gegeben.

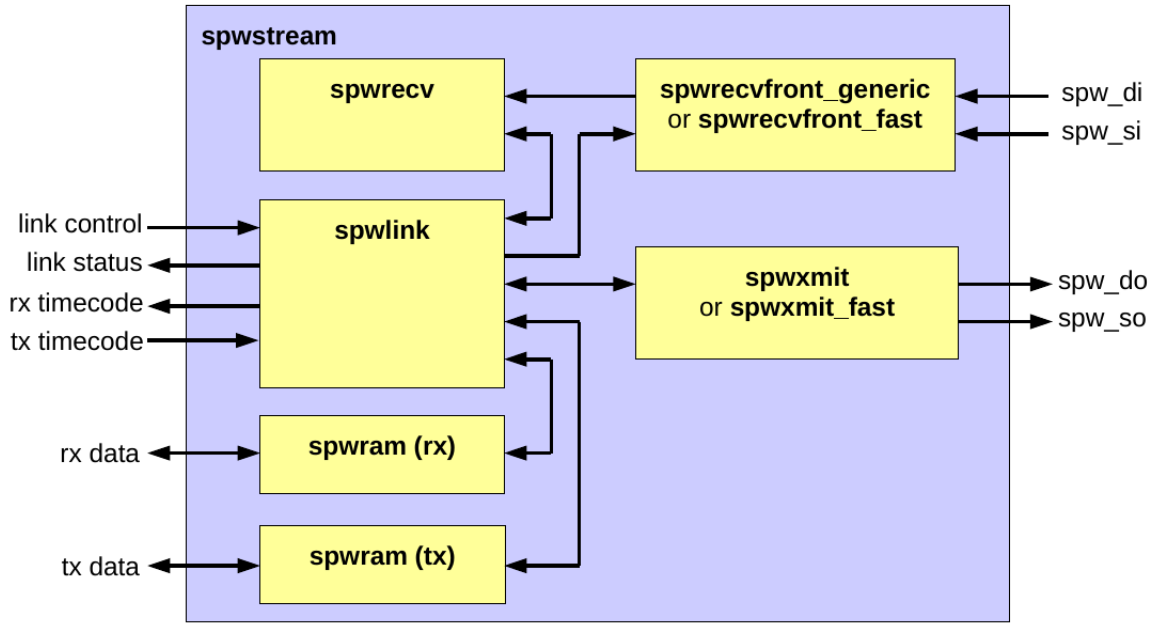

Abbildung 21: Blockdiagramm des Spwstream Interface

Die spwlink Instanz enthält die SpaceWire Link-FSM zur Steuerung des Links. Die Eingänge dieser Instanz entsprechen den Flags [Link Disabled], [LinkStart] und [AutoStart] (link control). Außerdem werden über diese Instanz zu sendende Time-Codes (tx timecode) eingelesen. Die Ausgänge von spwlink geben Auskunft über den aktuellen Zustand der Link-FSM (Started, Connecting, etc.) und über aufgetretene Fehler (Disconnect error, Parity error, etc.) (link status), bzw. geben empfangene

Time-Codes (rx timecode) aus.
Wie auch im beispielhaften Blockdiagramm eines SpaceWire Link-Interfaces (Abbildung 15) zu sehen ist, bewirkt das tick_in Signal des Interfaces das Senden eines Time-Codes, der aus time_in und ctrl_in besteht. tick_out signalisiert den Empfang eines Time-Codes bestehend aus time_out und ctrl_out. Der SpaceWire Light IP-Core signalisiert jeden empfangenen Time-Code per tick_out Signal. Wie auch in der Dokumentation beschrieben, entspricht dies nicht der Definition zum Umgang mit Time-Codes im Standard. Nach dieser gilt ein empfangener Time-Code nur dann als gültig, wenn dessen Zählerwert um 1 größer ist, als der des zuvor empfangenen Time-Codes. Ansonsten ist er ungültig. Nur ein gültiger Time-Code ist per tick_out anzuzeigen. Der interne Zählerwert einer SpaceWire Node oder eines SpaceWire Routers wird jedoch auch bei einem ungültigen Time-Code mit dessen Zählerwert aktualisiert. Siehe dazu Kapitel 8.12.2 auf Seite 84 des Standards.
Zu sendende (tx data) Pakete gelangen über ein einfaches Handshake-Protokoll in das Tx-FIFO (spwram(tx)) und werden dann, gesteuert durch die spwlink Instanz, an den Transmitter (spwxmit - allgemeine Implementierung, spwxmit_fast - schnelle Implementierung) weiter gereicht. Dieser setzt zu sendende SpaceWire Tokens auf Low-Level Ebene in Data-Strobe codierte Signale um. Die Bitrate ist bei einem bestehenden Link in beide Richtungen (Zustand Run) jederzeit über den Taktteiler einstellbar.
In der Dokumentation des SpaceWire Light wird beschrieben, was im Fehlerfall mit den verbleibenden Daten eines aktuell zu sendenden Pakets im Tx-FIFO passiert. Dort wird auf Seite 6 beschrieben, dass diese Daten nicht verworfen werden, falls der Link explizit durch das linkdisable Signal (entspricht [Link Disabled] Flag) beendet wird. Dieses Vorgehen verletzt das Link Error Recovery Schema, nach dem diese Daten des Pakets inklusive des zugehörigen EOP zu verwerfen sind und hat das Senden eines unvollständigen Pakets nach der Link Re-Initialisierung zur Folge. Auf der gegenüber liegenden Seite wird der Abbruch der Verbindung als Disconnect Error erkannt, was zum Einfügen eines EEPs in das Rx-FIFO führt.
Auf der Empfängerseite werden im Receiver Front-End (spwrecfront_generic - allgemeine Implementierung, spwrecfront_fast - schnelle Implementierung) über Data und Strobe gültige Bits mittels Oversampling erkannt und zur Decodierung in Einheiten von 1 - 4 Bits (spwrecfront_fast) an die spwrecv Instanz weitergereicht. Decodierte Token gelangen von dort aus zur spwlink Instanz, die gültige N-Chars in das Rx-FIFO (spwram(rx)) schreibt. Durch ein einfaches Handshake-Protokoll können die Paketdaten dann aus dem Rx-FIFO gelesen werden.

2.3.3.2 Konfiguration

Die Spwstream Instanz des SpaceWire Light IP-Cores wird über VHDL Generics konfiguriert. Die folgende Tabelle zeigt alle Einstellungsmöglichkeiten und enthält außerdem Informationen zu Default-Werten.

Tabelle 4: Konfiguration der Spwstream Instanz (SpaceWire Light IP-Core)

VHDL Generic	Beschreibung	Datentyp	Default
sysfreq	Systemfrequenz in Hz	real	/
txclkfreq	Sendefrequenz in Hz	real	0
rximpl	Receiver Front-End Implementierung	custom VHDL type	impl_generic
rxchunk	Bit Einheiten (Receiver Front-End)	integer	1
tximpl	Transmitter Implementierung	custom VHDL type	impl_generic
rxfifosize_bits	Tiefe des Rx-FIFOs	integer	11 (2048 Bytes)
txfifosize_bits	Tiefe des Tx-FIFOs	integer	11 (2048 Bytes)

sysfreq ist die Frequenz des Systemtakts. Dieser Wert muss mit der Frequenz des clk-Signals übereinstimmen. sysfreq wird zur internen Berechnung der Timeouts in Zusammenhang mit der Link-Initialisierung (6,4 μs und 12,8 μs) und des Timeouts für den Disconnect error (0,85 μs), sowie zur Berechnung eines Taktteilerwerts zur Erzeugung einer Übertragungsrate von 10 MBit/s während der Link-Initialisierung verwendet.
txclkfreq ist die Frequenz des separaten Takts, der bei der schnellen Implementierung des Transmitters zum Senden von Daten verwendet wird. Dieser Wert muss mit der Frequenz des txclk-Signals übereinstimmen. Er wird zur Berechnung eines Taktteilerwerts zur Erzeugung einer Übertragungsrate von 10 MBit/s während der Link-Initialisierung verwendet. txclkfreq wird nur berücksichtigt, falls tximpl = impl_fast.
rximpl dient zur Auswahl der Implementierung des Receiver Front-Ends. rximpl = impl_generic wählt die allgemeine (generic) Implementierung aus. Zum Sampling von Data und Strobe wird der Systemtakt (clk) verwendet. Der separaten Takt (rxclk) wird nicht gebraucht. Die Rx-Bitrate darf maximal der halben Frequenz des Systemtakts entsprechen. rximpl = impl_fast wählt die schnelle (fast) Implementierung aus. Zum Sampling von Data und Strobe wird der separate Takt (rxclk) benötigt. Die Rx-Bitrate darf maximal der Frequenz von rxclk oder dem rxchunk-fachen der Frequenz des Systemtakts (clk) entsprechen, je nachdem welcher Wert niedriger ist.
Mit **rxchunk** lässt sich die Anzahl zu empfangender Bits pro Systemtaktzyklus einstellen. Falls rximpl = impl_generic, so ist rxchunk auf 1 zu setzen. Falls rximpl = impl_fast, kann rxchunk auf Werte zwischen 1 - 4 gesetzt werden. Die Rx-Bitrate ist durch das rxchunk-fache der Frequenz des Systemtakts (clk) begrenzt.
tximpl dient zur Auswahl der Implementierung des Transmitters. tximpl = impl_generic wählt die allgemeine (generic) Implementierung aus. Zum Senden von Daten wird der Systemtakt (clk) verwendet. Der separate Takt (txclk) wird nicht gebraucht. Die Tx-Bitrate entspricht entweder der Frequenz des Systemtakts oder der Frequenz die sich durch einen Wert des Taktteilers (txdivcnt) größer 0 ergibt: $\mathrm{f_{Tx}} = \mathrm{clk} / (\mid \mathrm{txdivcnt} \mid +1)$. tximpl = impl_fast wählt die schnelle

(fast) Implementierung aus. Zum Senden von Daten wird der separate Takt (txclk) benötigt. Die Tx-Bitrate entspricht entweder der Frequenz des separaten Takts (maximal fünffache Frequenz des Systemtakts) oder der Frequenz die sich durch einen Wert des Taktteilers (txdivcnt) größer 0 ergibt: $\mathrm{f_{Tx}} = \mathrm{txclk} \,/\, (\mid \mathrm{txdivcnt} \mid +1)$.
Mit **rxfifosize_bits** lässt sich die Tiefe des Rx-FIFOs bestimmten. Werte zwischen 6-14 sind erlaubt ($2^6 - 2^{14}$ Bytes).
Mit **txfifosize_bits** lässt sich die Tiefe des Tx-FIFOs bestimmten. Werte zwischen 2-14 sind erlaubt ($2^2 - 2^{14}$ Bytes).

2.3.3.3 Interface Signale

Die beiden folgenden Tabellen zeigen eine Übersicht über die Signale des Interfaces (VHDL Ports) der Spwstream Instanz. Es folgen Erläuterungen zu den einzelnen Signalen.

Tabelle 5: Spwstream Interface-Signale (SpaceWire Light IP-Core) - Eingänge

VHDL Port-Signal	Beschreibung	Input/Output
clk	Systemtakt	In
rxclk	Separater Takt für das Receiver Front-End	In
txclk	Separater Takt für den Transmitter	In
rst	System-Reset	In
autostart	Automatischer Aufbau eines Links	In
linkstart	Anfrage zum Aufbau eines Links	In
linkdis	Link trennen / keinen neuen Link aufbauen	In
txdivcnt [7:0]	Taktteilerwert (Transmitter)	In
tick_in	Signal zum Senden eines Time-Codes	In
ctrl_in [1:0]	Control flags des zu sendenden Time-Codes	In
time_in [5:0]	Wert des zu sendenden Time-Codes	In
txwrite	Tx-FIFO Signal	In
txflag	Tx-FIFO Signal	In
txdata [7:0]	Zu sendendes Datenbyte (Tx-FIFO)	In
rxread	Rx-FIFO Signal	In
spw_di	Data-In Signal	In
spw_si	Strobe-In Signal	In

clk ist das high-aktive Signal des Systemtakts und wird immer benötigt. **rxclk** ist der separate Takt für die spwrecfront_fast Instanz und wird daher nur bei der Auswahl der schnellen Implementierung (impl_fast) des Receiver Front-Ends benötigt. **txclk** ist der separate Takt für die spwxmit_fast Instanz und wird daher nur bei der Auswahl der schnellen Implementierung (impl_fast) des Transmitters benötigt. **rst** ist das globale high-aktive Signal zum Reset des Systems. Dadurch werden alle Daten im Rx- und Tx-FIFO gelöscht. **autostart** ist die Option zum automatischen Aufbau des Links nach Erkennung der NULL detection sequence. Mit **linkstart** wird der Zeitpunkt zum Versuch des Aufbaus eines Links gesteuert. Das Signal hat Priorität vor autostart. **linkdis** trennt einen bestehenden Link und verhindert die Link (Re)-Initialisierung, solange das Signal aktiv ist. Es hat Priorität vor linkstart und autostart. **txdivcnt** ist der Wert des Taktteilers für den Takt des Transmitters zur Einstellung

der Tx-Bitrate bei bestehender Verbindung. Dieser teilt entweder clk, den Systemtakt (impl_generic) oder txclk, den separaten Takt (impl_fast). Zum Senden eines Time-Codes ist das **tick_in** Signal eine Systemtaktperiode lang zu setzen. Dafür wird der aktuelle Wert der control flags (**ctrl_in**) und der eigentliche Wert des Time-Codes (**time_in**) verwendet. Mit **txwrite** wird der aktuelle Wert von **txdata** in das Tx-FIFO geschrieben, sofern **txrdy** zu diesem Zeitpunkt gesetzt ist (das Tx-FIFO ist nicht voll). Hat **txflag** zum selben Zeitpunkt den Wert 0, so wird der Inhalt von txdata als Datenbyte interpretiert, also ein Datenbyte in das Tx-FIFO geschrieben. Hat txflag den Bedingungen entsprechend zu diesem Zeitpunkt den Wert 1, so wird der Inhalt von txdata entweder als EOP, sofern txdata = 0x00, oder als EEP, sofern txdata = 0x01, interpretiert und ein EOP bzw. EEP geschrieben. Im Normalfall wird jedoch von einer SpaceWire Node kein fehlerhaftes Paket, markiert durch ein EEP, gesendet. **txhalff** ist gesetzt, wenn das Tx-FIFO mindestens halb voll ist. **spw_di** und **spw_si** sind die Signale von Data bzw. Strobe in Empfangsrichtung.

Tabelle 6: Spwstream Interface-Signale (SpaceWire Light IP-Core) - Ausgänge

VHDL Port-Signal	Beschreibung	Input/Output
txrdy	Tx-FIFO Signal	Out
txhalff	Tx-FIFO Signal	Out
tick_out	Signal zur Anzeige eines empfangenen Time-Codes	Out
ctrl_out [1:0]	Control flags eines empfangenen Time-Codes	Out
time_out [5:0]	Wert eines empfangenen Time-Codes	Out
rxvalid	Rx-FIFO Signal	Out
rxhalff	Rx-FIFO Signal	Out
rxflag	Rx-FIFO Signal	Out
rxdata [7:0]	Empfangenes Datenbyte (Rx-FIFO)	Out
started	Link-FSM Signal	Out
connecting	Link-FSM Signal	Out
running	Link-FSM Signal	Out
errdisc	Disconnect error (RxErr)	Out
errpar	Parity error (RxErr)	Out
erresc	Escape error (RxErr)	Out
errcred	Credit error	Out
spw_do	Data-Out Signal	Out
spw_so	Strobe-Out Signal	Out

Beim Empfang eines neuen Time-Codes ist das **tick_out** Signal eine Systemtaktperiode lang gesetzt. **ctrl_out** und **time_out** entsprechen dem Wert der control flags bzw. dem eigentlichen Wert des zuletzt empfangenen Time-Codes. Mit **rxread** wird der aktuelle Wert von **rxdata** aus dem Rx-FIFO gelesen, sofern **rxvalid** zu diesem Zeitpunkt gesetzt ist (das Rx-FIFO ist nicht leer). Hat **rxflag** zum selben Zeitpunkt den Wert 0, so entspricht der Inhalt von rxdata einem Datenbyte, es wird also ein Datenbyte aus dem Rx-FIFO gelesen. Hat rxflag den Bedingungen entsprechend zu diesem Zeitpunkt den Wert 1, so entspricht der Inhalt von rxdata einem EOP, sofern rxdata = 0x00, oder einem EEP, sofern rxdata = 0x01. Dadurch wird ein EOP oder EEP gelesen. **rxhalff** ist gesetzt, sofern das Rx-FIFO mindestens halb voll ist. Die Signale **started**, **connecting** und **running** zeigen an, dass sich die Link-FSM aktuell im Zustand Started, Connecting bzw. Run befindet. **errdisc**,

errpar, **erresc** und **errcred** signalisieren einen Disconnect error, Parity error, Escape error bzw. Credit error. **spw_do** und **spw_so** sind die Signale von Data bzw. Strobe in Senderichtung.

3 Entwicklung der Testbench

Das folgende Kapitel enthält detaillierte Erläuterungen zur Entwicklung der UVM-Testbench. Zunächst wird im Abschnitt 3.1 das Designkonzept der Testbench vorgestellt. Als nächstes wird auf die Entwicklung eines Verifikationsplans zur Überprüfung der Funktion von SpaceWire IP-Cores eingegangen. Darauf folgen Informationen zum Aufbau der Grundstruktur der Block-Level Testbench. Die Abschnitte 3.4, 3.5 und 3.6 enthalten Erläuterungen zur Entwicklung wichtiger Sub-Komponenten der Testbench. Der Abschnitt 3.7 beschreibt den Entwurf von Sequences, die für abschließende Funktionstests des SpaceWire Light IP-Cores verwendet wurden.

3.1 Designkonzept

Im beispielhaften Blockdiagramm des SpaceWire Link Interfaces sind bereits ausreichende Informationen zu benötigten Interface-Signalen zur Kommunikation zwischen dem Transmitter, dem Receiver, der Link-FSM und einem Hostsystem enthalten. Beim Hostsystem handelt es sich um diejenige Instanz am Ende eines Links, die zu sendende Daten in Form von Paketen definiert und empfangene Daten speichert. Die Position des Hostsystems würde folglich die Testbench einnehmen müssen, um den Vergleich von gesendeten und empfangenen Daten über einen SpaceWire Link und damit letztendlich die Verifikation eines SpaceWire IP-Cores zu ermöglichen. Dazu muss die Testbench einerseits den Part des Hostsystems auf der Seite des DUT (des SpaceWire Light IP-Cores), als auch andererseits den Part des Hostsystems auf der gegenüber liegenden Seite des Links übernehmen. Dies macht hier gemäß des Blockdiagramms die vollständige Implementierung des SpaceWire Codecs inklusive Transmitter, Receiver und Link-FSM erforderlich. Insgesamt repräsentieren Testbench und DUT als Einheit also den Link zwischen zwei SpaceWire Nodes.
Zum Testbench Designkonzept gehören alle anfänglichen Überlegungen, auf welche Weise der SpaceWire Codec in der Testbench implementiert werden kann. Dies betrifft primär die Umsetzung der SpaceWire Link-FSM basierend auf deren Zustandsdiagramm als zentralem Block zur Steuerung der Link-Initialisierung, des Datentransfers bei bestehender Verbindung sowie zur Erkennung der im Standard definierten Fehler und die damit verbundenen Maßnahmen zur Re-Initialisierung des Links (Link Error Recovery Schema). Weiterhin ist der Receiver zu implementieren, der die Data-Strobe Signale eines IP-Cores decodiert, daraus SpaceWire Tokens bildet und schließlich in Pakete umwandelt. Die Aufgabe des Transmitters hingegen ist es, zuvor definierte Pakete und Systemzeitinformationen in Form von SpaceWire Tokens zusammen mit Tokens zur Verwaltung des Links (Flusssteuerung und Aufrechterhaltung des Links), gesteuert durch die Link-FSM, als Data-Strobe codierte Signale zum IP-Core zu senden. Für diese Operationen wird ein Data-Strobe Interface auf Low-Level Ebene zum Senden und Empfangen von Daten benötigt. Weiterhin ist ein FIFO-Interface notwendig, um über den SpaceWire Link durch den IP-Core zu sendende Daten/Pakete in dessen Tx-FIFO (RAM) zu schreiben und empfangene Daten/Pakete aus dem Rx-FIFO des IP-Cores zu lesen. Da beim SpaceWire Light für zu sendende sowie empfangene

Time-Codes separate Interface-Signale vorhanden sind, entstand die Idee, ein weiteres Time-Code Interface in der Testbench zu verwenden. Dadurch lassen sich Time-Codes und Daten/Pakete trennen. Diese Idee führte dann zur Implementierung eines dritten Agents (TC Agent) ausschließlich für Time-Codes mit zugehörigem Time-Code Interface.
Weitere Informationen und Anregungen zur Implementierung des SpaceWire Codec konnten im Datenblatt des GRSPW IP-Cores, Aeroflex Gaisler AB (2008) gefunden werden, der den SpaceWire Codec mit RMAP Unterstützung und einem AMBA Hostsystem-Interface implementiert. Diese betrafen hauptsächlich verwendete Ein- und Ausgangssignale des Transmitters und Receivers.

3.1.1 Task-basierter Aufbau

Der ursprüngliche Gedanke bei der Entwicklung der UVM-Testbench war ein grundsätzlicher Aufbau der Testbench in SystemVerilog Tasks. Dies bezieht sich sowohl auf die einzelnen Komponenten als auch auf die Umsetzung der Features von SpaceWire. Beispielsweise sollte es verschiedene Sende-Tasks (Tx-Tasks) für die einzelnen SpaceWire Tokens geben, deren Argument das zu sendende Paritätsbit selbst oder eine entsprechende Information zur Bestimmung des Bits ist. Die Umsetzung in Data-Strobe codierte Signale erfolgt dann innerhalb der Tasks. Basierend auf der Feststellung, welches Token als Nachfolger des jeweils vorherigen Tokens gültig ist, sollte dann durch die Vorgänger-Nachfolger Beziehung das zu sendende Paritätsbit ohne Berechnung bestimmt werden. Tatsächlich ist dies relativ unproblematisch und möglich, allerdings wäre dafür eine Art Lookup-Tabelle für Paritätsbits nötig gewesen, was einen zu großen Aufwand darstellt. Wie sich später herausstellte, ist die Berechnung des aktuell zu sendenden Paritätsbits über eine einfache Funktion, basierend auf der Vorgänger-Nachfolger Information, wesentlich eleganter. Nach dem Senden des aktuellen Tokens wird dieses als zuletzt gesendetes Token gespeichert und vor dem Senden des nächsten Tokens zur Berechnung des nächsten Paritätsbits genutzt. Statt mehrere Tasks einzusetzen, die mehr oder weniger die gleichen Aktionen durchführen, existiert in der aktuellen Version der Testbench nur ein einziger Tx-Tasks zum Senden eines Tokens, das als Argument übergeben wird. Der Tx-Task ruft an entsprechender Stelle die Funktion zur Berechnung des Paritätsbits mit dem Token als Argument auf. Aufgrund der besonderen Form des ersten zu sendenden NULL-Tokens im Rahmen der Link-Initialisierung (das zugehörige Paritätsbit ist 0 und es gibt kein Vorgänger-Token), existierte in einer früheren Version der Testbench noch ein separater Task zum Senden der ersten NULL. Dieser bzw. die darin durchgeführten Anweisungen wurden aber später noch in besagtem Tx-Task integriert.
Auf Empfängerseite war hingegen nur ein Task zur Decodierung der DS-Signale vorgesehen. Dieser sollte im DS-codierten Bitstream die Bitmuster der einzelnen Token erkennen und dann ein entsprechendes Flag (z.B. gotN-Char, gotFCT) zur Anzeige des Empfangs setzen. Diese Informationen des Rx- bzw. Decodier-Tasks sollten dann zur Link-FSM gelangen. Außerdem existierte die Idee, den Tx-Tasks über die Information des aktuellen Zustands der State Machine mitzuteilen, welche Tokens aktuell gesendet werden dürfen. Für das FIFO-Interface sollte es zwei einfache Tasks geben. Der Write-Task schreibt in Abhängigkeit der entsprechenden Interface-Signale der Spwstream Instanz zu sendende Daten in das Tx-FIFO des IP-Cores. Der Read-Task liest in Abhängigkeit der zugehörigen Signale über den Link empfangene Daten aus dem Rx-FIFO.

3.1.1.1 Prozessparallelisierung

Betrachtet man das Zustandsdiagramm der SpaceWire Link-FSM, so ist zu erkennen, dass es parallel ablaufende Prozesse beim Bewegen durch die States gibt. Der Übergang vom Zustand ErrorReset in den Zustand ErrorWait ist geradlinig. Nach Ablauf des 6,4 μs Timeouts, das in SystemVerilog z.B. einfach per Delay (#) umsetzbar ist, wird bedingungslos in den Folgezustand gewechselt. Im Zustand ErrorWait hat man bereits die ersten parallelen Prozesse. Einerseits wird nach dem 12,8 μs Timeout in den Folgezustand gewechselt, andererseits kann innerhalb dieser Zeit ein Fehler der Kategorie I auftreten. In diesem Fall wird zurück in den Ausgangszustand gewechselt. Im Zustand Ready hat man ein ähnliches Szenario. So lange das [Link Enabled] Flag nicht gesetzt ist, führt ein Fehler der Kategorie I zum Reset der State Machine. Sobald das Flag gesetzt ist, ohne dass währenddessen ein Fehler auftrat, wird mit der Link-Initialisierung im Zustand Started begonnen. Die Parallelisierung von Prozessen setzt sich auch bei weiteren Zustandswechseln fort. In den Zuständen Started und Connecting für das NULL/FCT-Handshake Protokoll, kann neben einem Fehler der Kategore I bzw. II auch das 12,8 μs Timeout zum Zurücksetzen der State Machine führen. Im Zustand Run werden Pakete gesendet und empfangen und Fehler der Kategorie III detektiert, die dann zum Verbindungsabbruch führen und alle laufenden Prozessen beenden.
Diese Tatsachen regen zur Verwendung von SystemVerilog fork-join Konstrukten zur Parallelisierung von Prozessen innerhalb der Link-FSM an. Sie stellen die einzige Möglichkeit der Prozessparallelisierung dar. Mit fork-join Blöcken lassen sich Blöcke innerhalb eines Tasks auf verschiedene Weise parallelisieren. Mit fork-join lässt sich ein Thread (z.B. ein Task) so lange blockieren, bis alle innerhalb des fork-join Konstrukts aufgerufenen Threads beendet sind. Erst dann wird mit weiteren Anweisungen innerhalb des Tasks fortgefahren. Mit fork-join_none wird der Thread nicht blockiert. Alle innerhalb des fork-join_none Konstrukts aufgerufenen Threads laufen parallel zu den darauf folgenden Anweisungen des Tasks, beginnend mit dem ersten blocking statement, ab. Am interessantesten hinsichtlich der Prozessparallelisierung innerhalb der Testbench ist das fork-join_any Konstrukt. Mit fork-join_any lässt sich ein Thread so lange blockieren, bis ein beliebiger innerhalb des Konstrukts aufgerufener Thread als erstes beendet wird. Die verbleibenden Threads laufen dann bis zum Ende parallel zu weiteren Anweisungen im Hauptthread ab. Mit der disable fork Anweisung, die normalerweise unmittelbar auf ein join_xxx folgt, lassen sich zu diesem Zeitpunkt noch nicht beendete Threads vorzeitig beenden. Dabei ist zu beachten, dass auch all diejenigen Threads beendet werden, die zuvor im Rahmen des zugehörigen Konstrukts gestartet wurden.
Durch ein fork-join_any gefolgt von einem disable fork im Zustand Started hat man beispielsweise die Möglichkeit, einen Timer für das 12,8 μs Timeout, einen Task zur Erkennung eines Fehlers der Kategorie I, sowie einen Task zur Erkennung des ersten NULLs zu parallelisieren. Je nachdem, ob zuerst der Timer abläuft (kein Fehler innerhalb der 12,8 μs), die Erkennung eines Fehlers den Task beendet (ein Fehler trat innerhalb von 12,8 μs auf) oder das gotNULL Flag gesetzt ist, wird in den Zustand ErrorReset oder Connecting gewechselt. Auf ähnliche Weise lassen sich per fork-join_any auch in allen weiteren Zuständen die entsprechenden Prozesse parallelisieren. Durch ein anschließendes disable fork wird dann das Beenden der noch laufenden Tasks vor dem Wechsel in den Folgezustand sichergestellt. Dies ist notwendig, da beispielsweise der Task zur Erkennung eines Fehlers der Kategorie I in mehreren Zuständen gestartet wird. Weiterhin ist es per fork-join_any möglich Tasks, die auf gewisse Ereignisse warten bzw. reagieren, zu parallelisieren.

3.1.1.2 Receiver: DS-Decodierung

Der Receiver, der zur Decodierung Data-Strobe codierter Signale dient, sollte einer ersten Idee nach mit einem separaten Receiver-Takt getaktet werden. Dieser Takt müsste dann folglich stets demjenigen Takt entsprechen, mit dem auf der gegenüber liegenden Seite des Links vom DUT aus über Data und Strobe gesendet wird. Außerdem war bekannt, dass die XOR-Verknüpfung von Data und Strobe zur Taktrückgewinnung nutzbar ist. Die Möglichkeit, diesen Takt dann zur Erkennung von Bitmustern zu verwenden, schien keine gute Lösung zu sein und wurde deshalb nicht weiter verfolgt. Stattdessen war die Idee, die Information über die Frequenz des Systemtakts (impl_generic) oder des separaten Takts (impl_fast) für den Transmitter des DUT in Form eines Konfigurationsparameters in der Testbench-Hierarchie nach unten bis zum Receiver durchzureichen. Der Sender des DUT und der Empfänger in der Testbench arbeitet so stets mit demselben Takt. In diesem Zusammenhang wurde anfangs allerdings nicht bedacht, das die Tx-Bitrate des DUT bei bestehender Verbindung (aktiver Link) jederzeit über den Taktteiler (txdivcnt Signal) einstellbar ist. Hinsichtlich der Verifikation eines SpaceWire IP-Cores ist es also auch notwendig, den Wert des Taktteilers in verschiedenen Tests zu variieren. In diesem Fall müsste sich der Receiver-Takt in der Testbench bei jeder Änderung des Teilerwerts entsprechend anpassen. Dies ist nur durch eine dynamische Anpassung von Parametern, die den Receiver-Takt bestimmen, oder durch die oben beschriebene Taktrückgewinnung machbar.
Die wesentlich einfachere Möglichkeit zur Lösung dieses Problems ist die Verwendung eines Takts mit sehr hoher Frequenz. Diese ist wesentlich höher als die Frequenz, mit der Bits auf der Data-Strobe Leitung ankommen. Zur Erkennung von Signalübergangen bzw. neuer Bits im Data-Strobe codierten Bitstream werden Data und Strobe mit einer hohen Frequenz abgetastet. Es handelt sich um das bereits zuvor erwähnte Prinzip des synchronen Oversamplings, also um eine Überabtastung der Signale. Dieses Verfahren wird auch im Receiver Front-End des SpaceWire Light genutzt. In der Dokumentation des IP-Cores wird erwähnt, dass dazu in der Praxis eine mindestens doppelt so große Abtastrate im Vergleich zur Rx-Bitrate ausreicht. Samples des Data- und Strobe-Signals sind demnach mit mindestens der doppelten Frequenz zu nehmen. Im SpaceWire Standard ist eine maximale Datenübertragungsrate von 400 MBit/s festgelegt. Eine Abtastfrequenz von 800 MHz ist also ausreichend, um der Definition entsprechend alle SpaceWire DS-Signale fehlerfrei zu decodieren. Das Verfahren des synchronen Oversamplings wird im Driver des Data-Strobe Agents angewendet.
Zur Decodierung des seriellen Bitstreams wird als Ausgangspunkt der Zustand ErrorReset angenommen. In diesem befindet sich die State Machine nach einem Reset oder nach der Erkennung eines Fehlers bei aktivem Link. Sobald der Receiver im Folgezustand eingeschaltet ist, wird mit der Decodierung begonnen. Als erstes Bit ist das erste Bit der ersten NULL zu erwartet, das per Definition im Standard 0 ist. Nach Data-Strobe Codierungsschema hat dies den ersten Signalwechsel auf der Strobe-Leitung zur Folge. Es folgt die Erkennung der NULL detection sequence. Wird diese erkannt, entspricht das zuletzt empfangene Bit dem Paritätsbit des darauf folgenden nächsten Tokens, das während der Link-Initialisierung entweder ein FCT oder ein NULL sein kann. Man hat somit einen festen Bezugspunkt für die weitere Decodierung des Streams. Die folgende Abbildung 22 zeigt das Schema, mit dem der Bitstream ab diesem Zeitpunkt in SpaceWire Tokens umgesetzt wird.

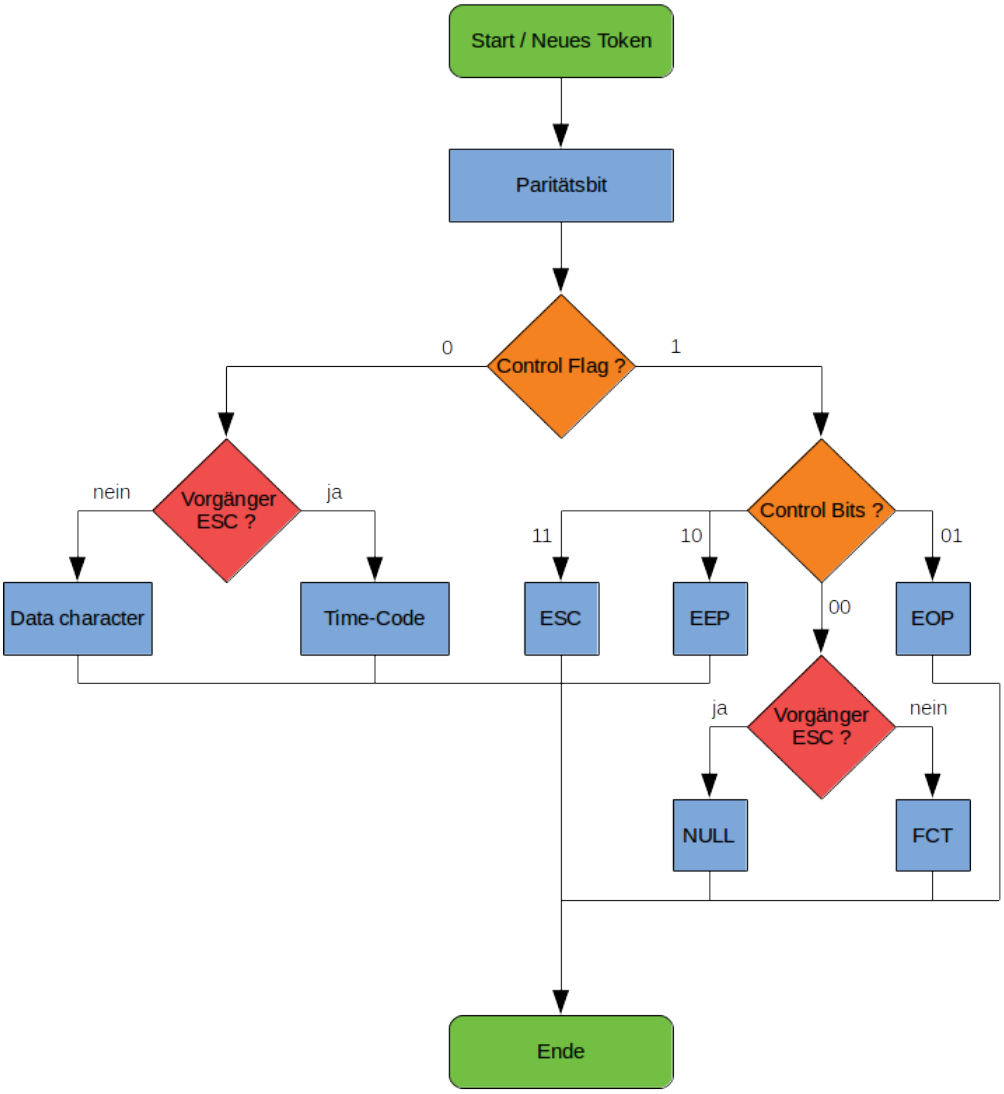

Abbildung 22: Data-Strobe Decodierungsschema

Der Ausgangspunkt ist ein zuletzt vollständig decodiertes Token. Es wird mit der Decodierung des Nachfolgertokens fortgefahren. Das nächste gültige Bit ist das zugehörige Paritätsbit. Danach folgt das data-control flag. Der parity coverage entsprechend kann jetzt mit den Bits des zuvor empfangenen Tokens sowie dem eben empfangenen Paritätsbit und dem data-control flag des aktuellen Tokens überprüft werden, ob ein Paritätsfehler vorliegt. Ist dies nicht der Fall, so ist das zuletzt empfangene Token gültig. Ist das soeben empfangene data-control flag 0, handelt es sich bei dem zu decodierenden Token um ein data character. Dieses Datenbyte ist entweder Bestandteil eines Time-Codes, falls das zuvor empfangene Token ein ESC war (Time-Code = ESC + data character) oder ein einfaches Datenbyte und daher Bestandteil eines Pakets. Nach dem Empfang der nächsten acht gültigen Bits wird das nächste Token erwartet. Ist das soeben empfangene data-control flag 1, handelt es sich bei dem zu decodierenden Token um ein control character. Die folgenden zwei gültigen Bits sind die control bits des control characters. Der Codierung entsprechend gehören sie zu einem FCT (00), einem EOP (01), einem EEP (10) oder zu einem ESC (11). Ein decodiertes FCT ist entweder Bestandteil eines NULL, wenn das zuvor empfangene

Token ein ESC war (NULL = ESC + FCT) oder ein einfaches FCT. Danach wird wieder das nächste Token erwartet. Zur Speicherung empfangener Tokens mit einer Breite von entweder 3, 7, oder 9 Bits (control characters, NULL, data characters oder Time-Codes, jeweils ohne Paritätsbit), die Bit für Bit decodiert werden, bieten sich SystemVerilog Queues an.

3.1.1.3 Transmitter: DS-Encodierung

Wie bereits erwähnt, wurde die ursprüngliche Idee, einzelne Tx-Tasks zum Senden eines SpaceWire Tokens zu implementieren, verworfen. Einzelne Tasks für die control characters sind deshalb nicht notwendig, da für alle nur das Paritätsbit die einzige Variable ist. Die restlichen 3 Bits stehen fest und können daher als FCT, EOP, EEP bzw. ESC in der Testbench hartcodiert werden. Bei den data characters ist das Paritätsbit und das eigentliche Datenbyte variabel. Das aktuell zu sendende Datenbyte inklusive data-control flag lässt sich einfach als Argument eines Tasks übergeben. Bei den aus zwei Tokens zusammen gesetzten Codes (control codes) sieht es ähnlich aus. Die 7 Bits eines NULL (exklusive Paritätsbit) stehen auch fest und lassen sich ebenso hartcodieren. Beim Time-Code sind das Paritätsbit und 6 Bits des Datenbytes, die den eigentlichen Wert des Time-Codes ausmachen, variabel. Die verbleibenden 2 Bits sind für eine zukünftige Verwendung reservierte control flags und per Definition im Standard auf 0 zu setzen. Auch das Datenbyte des Time-Codes inklusive data-control flag lässt sich als Argument eines Tasks übergeben. Diese Tatsachen führten zu dem Gedanken, einen einzelnen Tx-Task zum Senden von SpaceWire Tokens zu verwenden. Die Übergabeparameter des Tasks sind das zu sendende Token und eventuell das zugehörige Paritätsbit. Da das Token exklusive Paritätsbit aber entweder 3, 7 oder 9 Bit breit ist, muss auch die Breite des Übergabeparameters variabel sein. Zur Berechnung des zu sendenden Paritätsbits sind Informationen sowohl aus dem Vorgänger-Token als auch aus dem Nachfolger-Token nötig. Nach dem Senden eines Tokens muss dieses also als Vorgänger-Token zwischengespeichert werden. Dieses besitzt natürlich ebenfalls eine variable Breite.
Als Container für aktuell zu sendende und zuletzt gesendete Token kommen in SystemVerilog dynamische Arrays und Queues in Frage. Dies gilt ebenso für empfangene Tokens (DS-Decodierung). Bei den dynamischen Arrays handelt es sich um unpacked arrays, deren Größe und Dimension variabel ist. Dynamische Arrays sind außerdem für sämtliche Datentypen verwendbar. Für die dynamischen Arrays existieren außerdem Methoden, mit denen beispielsweise der Inhalt des Arrays gelöscht oder die aktuelle Größe bestimmt werden kann. Für die Tx- und Rx-Tokens sind daher jeweils zwei dynamische Arrays vorgesehen. Eine Queue ist sozusagen ein eindimensionales unpacked array. Sie wird genau so wie ein unpacked array deklariert. Die Größe einer Queue ist ebenfalls variabel, sie besitzt aber im Vergleich zu einem dynamischen Array weitere Methoden. Mit diesen lassen sich z.B. Elemente an beliebigen Stellen einfügen und löschen und Elemente am Anfang der Queue einfügen oder am Ende entnehmen. Diese Eigenschaften machen sie vor allem als Container für Bits, die während der Decodierung von SpaceWire Tokens empfangen werden, interessant.
Zur Berechnung eines zu sendenden Paritätsbits wird z.B. eine einfache Funktion im Tx-Task aufgerufen. Die Argumente sind dann die beiden dynamischen Arrays, die das aktuell zu sendende bzw. das zuletzt gesendete Token enthalten.

3.1.2 Implementierung der SpaceWire FSM

Nachdem feststand, dass zur Umsetzung des SpaceWire Codecs auch die Implementierung der Link-FSM notwendig ist, musste entschieden werden, welche Variante und welches Prozessmodell zur Umsetzung des endlichen Automaten dienen sollte. Zunächst wurde eine Liste mit allen Signalen und Zuständen der FSM, so wie sie im Standard definiert sind, erstellt. Die beiden Timeouts (12,8 μs und 6,4 μs), das Zeitfenster zur Erkennung eines Disconnect errors (0,85 μs) sowie die Zeit zwischen dem Reset des Data- und Strobe-Signals nach einem Link-Reset (max. 0,555 μs) sollten in Form von Taktzyklen dargestellt werden. Bezüglich der FSM-Variante gab es zwei Optionen: Mealy- oder Moore-Automat. Diese unterscheiden sich grundsätzlich in der Definition der Ausgangssignale der FSM. Beim Moore-Automat hängt der Ausgang lediglich vom aktuellen Zustand ab. Beim Mealy-Automat hängt der Ausgang sowohl vom aktuellen Zustand als auch vom Eingang ab. Allgemein existiert immer ein aktueller Zustand (CurrentState) und ein Folgezustand (NextState), der durch den aktuellen Zustand und die Eingänge bestimmt wird. Synchron zu einem Takt wird dann nach jeder Taktperiode der Folgezustand zum aktuellen Zustand. Das Modell mit zwei Prozessen kombiniert die Zustandsübergangs- und Ausgangslogik der FSM in einem einzigen Prozess. Der zweite Prozess enthält das rücksetzbare Zustandsregister. Das Modell mit drei Prozessen setzt die Zustandsübergangs- und die Ausgangslogik in einen separaten Prozess. Der dritte Prozess enthält dann wieder das Zustandsregister. Weiterhin besteht die Möglichkeit, beide FSM-Varianten mit oder ohne Ausgangsregister zu modellieren. Es gibt zwei Typen von Ausgangsregistern, die sich hinsichtlich der zeitlichen Verzögerung der Ausgangssignale unterscheiden.
Die vorgestellten Varianten und ihre zugrunde liegenden Prozessmodelle beschreiben taktsynchrone und damit synthesefähige Schaltwerke. Als Bestandteil der Testbench muss die Link-FSM jedoch nicht synthesefähig sein, da der Testbench-Code ebenfalls nicht synthetisierbar ist. Deshalb erübrigt sich die Frage nach der umzusetzenden Variante inklusive Prozessmodell. Stattdessen lässt sich die Link-FSM folgendermaßen implementieren. Zur Beschreibung der Zustände dient eine Enumeration, es wird also ein Aufzählungsdatentyp verwendet. Dies kann in SystemVerilog dann z.B. so aussehen:

```
// Definition der FSM states

typedef enum {State1, State2, ..., StateN} <enum_type_name>;

<enum_type_name> CurrentState;
<enum_type_name> NextState;

// Definition des aktuellen Zustands

if(<reset_signal>) begin
    @(<clock_edge>);
    CurrentState = State1;
end
else begin
    CurrentState = NextState;
end
```

Da es sich bei der Link-FSM in der Testbench nicht um ein synchrones Schaltwerk handelt, können die aktuellen Zustände und die Bedingungen für Zustandsübergänge z.B. innerhalb einer Case-Anweisung definiert werden. Die Zustandsübergangslogik für die Folgezustände ist somit hinfällig, und es genügt prinzipiell die Definition von CurrentState. NextState wird dann nicht benötigt.

```
// FSM states und Zustandsübergange

case (CurrentState)
    State1 :  begin
                  ...
                  CurrentState = State2;
              end
    State2 :  begin
                  ...
                  CurrentState = State3;
              end
    StateN  : begin
                  ...
                  CurrentState = State1;
              end
    default : begin
                  `uvm_fatal("CurrentState undefined.")
                  CurrentState = State1;
              end
endcase
```

Die Zustandsübergänge sind nicht taktabhängig. Sie geschehen nur durch die Zuweisung von CurrentState (CurrentState = NextState).
In Bezug auf das Link-Timing hat man in SystemVerilog zwei Optionen. Entweder man definiert die Timeouts über ein Delay (#), z.B. #6.4us oder man stellt sie in Form von Taktzyklen als Integer-Wert dar. Zur Berechnung des Werts wird als Grundlage natürlich eine Frequenz benötigt. Der Nachteil dieser Methode sind Ungenauigkeiten, die sich bei der Berechnung des entsprechenden Werts ergeben können, da eine Gleitkommazahl durch einen Integer-Wert repräsentiert wird. Je niedriger die verwendete Frequenz ist, desto ungenauer wird die abgebildete Zeit, falls der resultierende Werte nicht ganzzahlig ist. Wie bereits erwähnt, beträgt die Datenrate während der Link-Initialisierung bei SpaceWire 10 MBit/s. Dafür wird zur Berechnung des Werts eine Frequenz von 10 MHz zugrunde gelegt. Die beiden Timeouts (12,8 μs und 6,4 μs) werden nur für die Link-Initialisierung benötigt.

$$\mathrm{T} = 1 / \mathrm{f}$$

$$\mathrm{clk\,cnt} = \mathrm{Timeout} / \mathrm{T}$$

$$\Rightarrow \mathrm{clk\,cnt} = \mathrm{Timeout} \cdot \mathrm{f}$$

clk cnt entspricht dem Wert, der in einen Integer-Wert umzuwandeln ist. Durch eine Frequenz von 10 MHz ergibt sich daraus:

$$\mathrm{clk\,cnt}_{6{,}4\,\mu\mathrm{s}} = 6{,}4\,\mu\mathrm{s} \cdot 10\,\mathrm{MHz} = 64$$

$$\mathrm{clk\,cnt}_{12{,}8\,\mu\mathrm{s}} = 2 \cdot \mathrm{clk\,cnt}_{6{,}4\,\mu\mathrm{s}} = 128$$

Wie man sieht, lassen sich die beiden Timeouts exakt durch ein Vielfaches der Periodendauer eines Taktsignals mit einer Frequenz von 10 MHz darstellen. In der Testbench lassen sich die Timeouts auch exakt nachbilden, wenn man eine Frequenz von 800 MHz zugrunde legt, die der Abtastfrequenz beim synchronen Oversampling entsprechen soll. Gleiches gilt für die 0,85 μs, denn:

$$\mathrm{clk\,cnt}_{0{,}85\,\mu\mathrm{s}} = 0{,}85\,\mu\mathrm{s} \cdot 800\,\mathrm{MHz} = 680$$

Durch die hohe Frequenz besteht die Möglichkeit auch andere Zeiten mit hoher Genauigkeit durch Taktzyklen darzustellen.

3.1.3 Testbench-Signale

Die Kommunikation zwischen der UVM-Testbench und einem SpaceWire IP-Core (DUT), hier dem SpaceWire Light IP-Core, läuft über drei Interfaces, über die die entsprechend zugehörigen Ein- und Ausgangssignale des IP-Cores mit denen der Testbench verbunden sind. Dabei entsprechen die Ausgänge des DUT den Eingängen der Testbench und umgekehrt. Auf Low-Level Ebene befindet sich das Data-Strobe Interface (DS_IF). Darüber sind das System-Reset-Signal (rst) sowie die Data- und Strobe-Signale jeweils in Senderichtung (spw_do und spw_so) bzw. in Empfangsrichtung (spw_di und spw_si) des DUT mit der Testbench verbunden. Über das Data-Strobe Interface werden DS-codierte SpaceWire Token zum DUT gesendet (spw_di, spw_si) bzw. vom DUT empfangen (spw_do, spw_so) und ein Reset des DUT durchgeführt. Über das zweite Interface TC_IF (Time-Code Interface) sind die Signale zum Senden und Empfangen von Time-Codes verbunden. tick_in ist das Ausgangssignal der Testbench, über das dem DUT eine Anfrage zum Senden eines Time-Codes, bestehend aus ctrl_in und time_in, geschickt wird. tick_out ist das Eingangssignal der Testbench, über das der Testbench ein vom DUT empfangener Time-Code, bestehend aus ctrl_out und time_out, übermittelt wird. Über das dritte Interface FIFO_IF (FIFO-Interface) sind die Signale zum Schreiben und Lesen des Tx- bzw. Rx-FIFOs, die Statussignale der FIFOs und der Link-FSM, die Signale zur Steuerung des Links (autostart, linkstart, linkdis) und das Signal für den Taktteiler (txdivcnt) zur Bestimmung der Tx-Bitrate des DUT verbunden. Über das FIFO-Interface werden von der Testbench aus zu sendende Pakete mittels eines einfachen Handshake-Verfahrens in das Tx-FIFO des DUT geschrieben bzw. aus dem Rx-FIFO gelesen. Außerdem wird hierüber die Link-Initialisierung des DUT gesteuert bzw. das Signal zum Beenden der Verbindung seitens des DUT gegeben. Die Statussignale der Link-FSM, die Auskunft darüber geben, in welchem Zustand sie sich aktuell befindet, sind nur für Debugging-Zwecke interessant. Die verbleibenden Statussignale (errdisc, errpar, erresc, errcred) zur Anzeige eines Disconnect errors, Parity errors, Escape errors bzw. Credit Errors werden in der aktuellen Version der Testbench nicht berücksichtigt, da sie ausschließlich Verifikationszwecken dienen.

Auf Top-Level Ebene der Testbench sind die verbleibenden Signale direkt über die Spwstream-Instanz des DUT verbunden. Dabei handelt es sich um den Systemtakt (clk), den separaten Sendetakt (txclk) und den separaten Receiver-Sampling-Takt (rxclk), die beide für die schnellen Implementierungen verwendet werden.

Im Kapitel 4.3 des Buchs SystemVerilog for Verification, Spear u. a. (2012), wird beschrieben, wie sich Race conditions bei den Signalen der Testbench und des DUT durch kontrolliertes Timing

der taktsynchronen Signale mittels Clocking Blocks vermeiden lassen. Dazu ist jeweils ein Clocking Block in jedem der drei Interfaces vorhanden. In den Clocking Block sind dann alle Interface-Signale zu setzen, die einer Taktdomäne zugehörig sind. Falls mehrere Taktdomänen vorhanden sind, kann ein Clocking Block für jede Domäne verwendet werden. Durch den Einsatz von Clocking Blocks sind die Testbench und das DUT synchronisiert, da die Signale innerhalb der Clocking Blocks synchron zum Takt getrieben (drive) und abgetastet (sample) werden. Das Timing im Clocking Block lässt sich separat für als Ein- bzw. Ausgänge definierte Signale per Delay steuern. Per Default sind hier Standardwerte für Verzögerungen der Ein- und Ausgangssignale definierbar, um die Signale zum richtigen Zeitpunkt, d.h. nicht zu spät zu treiben bzw. nicht zu früh abzutasten. Im folgenden Beispiel aus dem Buch auf S. 109 ist für Eingänge eine Verzögerung von 15 ns und für Ausgänge eine Verzögerung von 10 ns festgelegt.

```
// Clocking Block mit Standardverzögerungen

clocking cb @(posedge clk);
    default input #15ns output #10ns;
    output request;
    input  grant;
endclocking
```

Das Eingangssignal grant wird 15 ns vor der positiven Taktflanke von clk abgetastet und das Ausgangssignal request genau 10 ns nach der positiven Flanke von clk getrieben. Ohne default Statement sind individuelle Verzögerungen der Signale definierbar. Für die Clocking Blocks der UVM-Testbench sind default Statements zum Abtasten von Eingangssignalen in der Postponed Region und zum Treiben von Ausgangssignalen mit einer Verzögerung von #0.5 oder #1, was 0.5 ns bzw. 1 ns entspricht, festgelegt. Zum Beispiel: default input #1step output #1.

3.1.4 Datenverarbeitung und Speicherung

Noch vor Beginn des Aufbaus der Testbench Grundstruktur wurde überlegt, wie über die Testbench zu sendende Daten/Pakete definierbar seien und zum IP-Core, bzw. vom IP-Core zur Testbench gelangen könnten, um sie dort zu speichern. Ausgangspunkt dafür sind Sequence Items und die darin definierten Variablen, die Daten oder Signalen entsprechen. Meistens bezeichnen diese Variablen ihrer Namensgebung nach Ein- und Ausgänge des DUT im zugehörigen Interface. Dies ist z.B. in einem Sequence Item der Fall, das über einen Sequencer zu einem Driver gelangt, der auf diesem Interface agiert. In Bezug auf das Tx-FIFO des SpaceWire Light lautete also die Frage, wie Pakete, bestehend aus einer variablen Anzahl von Datenbytes und einem EOP oder EEP, in SystemVerilog darstellbar sind. Die Antwort sind dynamische Arrays, bestend aus 8 Bit breiten Datenwörtern. Mit dynamischen Arrays hat man dann die Möglichkeit, eine beliebige Anzahl von Datenbytes per size() Methode zu randomisieren und als Paketinhalt festzulegen. Ein dynamisches Array ist ebenso für die aus dem Rx-FIFO gelesenen Pakete einsetzbar. Zur Darstellung eines EOP oder EEP dient einfach ein 8 Bit breites packed array, da sie in den FIFOs als Byte gespeichert werden. Die beiden Tokens sind dort durch ein gesetztes txflag im Tx-FIFO bzw. durch rxflag im Rx-FIFO identifizierbar. Es gab also die Idee, jeweils ein Sequence Item für jedes zu sendende bzw. empfangene Paket zu verwenden. Ein Sequence Item enthält dann ein vollständiges Paket mit

beliebiger Länge, das durch ein EOP oder EEP (nur bei empfangenen Paketen) abgeschlossen ist. Somit ist die Datenverarbeitung und Speicherung hinsichtlich des FIFO-Interfaces geregelt.
Betrachtet man nun als nächstes das Time-Code Interface, so sind darin die Signale tick_in und tick_out vorhanden. Im zugehörigen Sequence Item muss es also Variablen geben, die empfangene Time-Codes, bestehend aus ctrl_out und time_out, speichern, wenn tick_out den Empfang anzeigt. Diese Variablen sind zwei packed arrays mit einer Breite von 2 Bits (ctrl_out) oder 6 Bits (time_out). Das gleiche gilt entsprechend für zu sendende Time-Codes. Es sollte also jeweils ein Sequence Item für jeden zu sendenden bzw. empfangenen Time-Code geben. Im SpaceWire Standard wird vorgeschlagen, einen Time-Code beispielsweise jede Millisekunde zu senden. Selbst bei einer mittleren Datenübertragungsrate ist daher die Anzahl zu sendender Time-Codes im Vergleich zu Paketen zeitlich betrachtet relativ gering. Man könnte also theoretisch auch ein einziges Response Item für alle während eines Tests empfangenen Time-Codes verwenden. Am Ende der run_phase müsste dann das Sequence Item abgeschlossen werden. Die UVM-Testbench nutzt jedoch ein Sequence Item pro Time-Code.
Beim Data-Strobe Interface sieht es ähnlich wie beim FIFO-Interface aus. Der Unterschied ist, dass jedes Request Item nicht ein gesamtes zu sendendes Paket enthält, sondern nur ein data character bzw. das EOP eines Pakets. Dadurch hat man die Möglichkeit, mit jedem Request Item separat ein zu sendendes Token zu definieren. Über eine zugehörige Sequence lässt sich dann durch die Anzahl generierter Request Items und ein Flag zum Senden des EOP (send_eop) ein Paket definieren. Ein Response Item kann hier so wie beim FIFO-Interface ein komplettes Paket in einem dynamischen Array speichern. Das Item bzw. Paket wird dann nach dem Empfang des nächsten EOP oder EEP, angezeigt durch ein entsprechendes Flag, abgeschlossen.

3.2 Verifikationsplan

Zur Überprüfung der Funktionsweise und Verifikation eines SpaceWire IP-Cores wurde ein Verifikationsplan erstellt, der sowohl die im SpaceWire ECSS-E-ST-50-12C Standard definierten und relevanten Anforderungen, als auch das zu erwartende Verhalten des IP-Cores abdeckt. Der Verifikationsplan basiert auf einer erstellten tabellarischen Zusammenfassung des Standards, die alle verifikationsrelevanten Inhalte enthält. Der Verifikationsplan und die Zusammenfassung sind im Anhang enthalten. Ersterer enthält lediglich die Functional Coverage Sektion inklusive der Unterpunkte und deren Beschreibung ohne die Verbindungen zu Cover Points oder Assertions, da diese am Ende der Bearbeitungszeit noch nicht vorhanden waren. Der Verifikationsplan ist in der Functional Coverage Sektion in drei Hauptkategorien gegliedert. Diese beschreiben das zu erwartende Verhalten des IP-Cores im Normalfall bzw. im normalen Betrieb (Normal operation behavior), bei einem im Standard definierten Fehler bzw. im Fehlerfall (Error operation behavior) und bei speziellen Randbedingungen bzw. in Ausnahmefällen (Exception operation behavior).
Der erste Punkt des zu erwartenden Verhaltens des IP-Cores im Normalfall beschreibt die Einhaltung des Data-Strobe Codierungsschemas. Damit ist zu überprüfen, ob der IP-Core auf Low-Level Ebene Daten auf der Data- und Strobe-Leitung korrekt codiert. Der zweite Punkt kann als Erweiterung des ersten Punkts zum Testen der DS-Codierung angesehen werden und betrifft die Rückgewinnung des Takts, mit dem das DUT aktuell sendet. Der folgende Punkt 1.1.3 soll sicherstellen, dass der IP-Core Datenbytes (data characters) stets LSB first über Data und Strobe

sendet (Vergleich der Bitpositionen in gesendeten und später empfangenen Bytes). Als nächstes ist das korrekte Verhalten während der Link-Initialisierung zu überprüfen. Dies betrifft die Einhaltung des NULL/FCT Handshake-Protokolls und die damit verbundenen Timeouts. Punkt 1.1.5 dient zur Überprüfung, ob nach einem Reset (Reset, Beenden des Links per linkdis oder Link-Fehler) das Data- und Strobe-Signal in der richtigen Reihenfolge, d.h. kontrolliert zurück gesetzt wird. Der Standard sieht vor, zuerst Strobe und kurze Zeit später Data zurück zu setzen. Weiterhin ist zu prüfen, ob der IP-Core die festgelegte Übertragungsrate von 10 MBit/s während der Link-Initialisierungsphase einhält. Der Punkt 1.1.7 beschreibt, dass nach einem Reset des Links das erste zu übertragende Bit (entspricht dem mit der ersten NULL gesendeten Paritätsbit) logisch 0 ist. Dies hat den ersten Signalübergang auf der Strobe-Leitung zur Folge. Weiterhin gibt der SpaceWire Standard vor, die control flags eines Time-Codes, die für zukünftige Anwendungen reserviert sind, auf 0 zu setzen. Dies ist durch den Punkt 1.1.8 abgedeckt. Die Punkte 1.1.9 bis 1.1.11 betreffen die Flusssteuerung (flow control). Damit lässt sich das Verhalten des IP-Cores hinsichtlich des Tx- und Rx credit counts bzw. des Füllstands der FIFOs überprüfen. Der IP-Core muss aufhören, FCTs zu senden, sobald das Rx-FIFO voll ist. Dies wird erreicht, wenn keine Daten mehr aus dem Rx-FIFO gelesen werden. Auf der anderen Seite darf der IP-Core nur so viele data characters schicken, wie durch das Senden von FCTs angefordert wurden (8 data characters pro gesendetem FCT). So lange seitens der Testbench keine neuen FCTs zum IP-Core gelangen, sind keine neuen data characters zu erwarten. Der nächste Punkt des Verifikationsplans betrifft die maximale Anzahl initial gesendeter FCTs. Diese ist prinzipiell von der Größe des Rx-Buffers (RX-FIFOs) am anderen Ende des Links abhängig, darf aber nicht größer als 7 sein, falls der Buffer bzw. das FIFO größer als 56 Bytes ist. Punkt 1.1.13 beschreibt die Priorität beim Senden von SpaceWire Tokens. Der Time-Code besitzt die höchste Priorität. Dementsprechend ist beispielsweise zu erwarten, dass der IP-Core nach einer Anfrage zum Senden eines Time-Codes per tick_in Signal nach dem aktuell gesendeten Token einen Time-Code schickt. Bei bestehender Verbindung in beide Richtungen wird ständig ein Token gesendet, auch dann, wenn aktuell keine zu sendenden Daten/Pakete vorhanden sind. In diesem Fall soll der IP-Core nur NULLs, Time-Codes oder FCTs senden. Dies trifft z.B. zu, wenn das Tx-FIFO leer ist, da keine neuen Pakete in das Tx-FIFO geschrieben wurden. Diese dauerhafte Aktivität auf der Data- und Strobe-Leitung stellt sicher, dass ein Disconnect error erkannt werden kann. Die Punkte 1.1.15 und 1.1.16 betreffen Signale des FIFO-Interfaces. Hier ist zu prüfen, wie der IP-Core auf das tick_in bzw. tx_write Signal reagiert (Time-Code bestehend aus ctrl_in und time_in wird gesendet, data character, EOP or EEP per txwrite in das Tx-FIFO geschrieben, falls txrdy = 1 ist). Außerdem ist zu erwarten, dass der seitens der Testbench gesendete Time-Code aus ctrl_out und time_out besteht, wenn der IP-Core den Empfang per tick_out anzeigt. Der folgende Punkt beschreibt das Verhalten des DUT vor dem Empfang des ersten NULL. Der IP-Core soll dann alle N-Chars und L-Chars ignorieren, also nur auf ein NULL reagieren und sendet als nächstes oder spätestens übernächstes Token ein FCT. Punkt 1.1.18 deckt die Erkennung der NULL detection sequence ab. Mit dem vorletzten Punkt zur Beschreibung des Verhaltens im Normalfall wird die Funktion zur automatischen („einseitigen“) Link-Initialisierung geprüft, bei der der IP-Core automatisch ein NULL nach dem Empfang des ersten NULLs sendet (Bedingung: linkdis = 0, linkstart = 0, autostart = 1). Die beiden letzten Punkte (1.1.20 und 1.1.21) behandeln die korrekte Verteilung der Systemzeit per Time-Code. Auf der einen Seite wird erwartet, dass der IP-Core nach

der Anfrage zum Senden eines Time-Codes (tick_in) einen Time-Code sendet, dessen Wert um 1 größer ist als der des vorherigen Codes. Auf der anderen Seite ist zu erwarten, dass der IP-Core nur dann den Empfang eines gültigen Time-Codes per tick_out Signal anzeigt, wenn der Wert dieses Time-Codes um 1 größer ist als der des zuvor empfangenen. Beides ist beim SpaceWire Light nicht prüfbar, da einerseits ein zu sendender Time-Code per ctrl_in und time_in definiert wird und andererseits das tick_out Signal (laut Dokumentation) jeden empfangenen Time-Code, unanbhängig von dessen Wert, anzeigt.
Die zweite Kategorie beschreibt das Verhalten des IP-Cores im Fehlerfall. Mit dem ersten Punkt soll die Einhaltung des Link Error Recovery Schemas geprüft werden (Fehlererkennung, korrektes FIFO/Buffer Handling, Link Re-Initialisierung). Mit den Punkten 1.2.2 bis 1.2.6 wird die Erkennung eines Link-Fehlers (auch während der Initialisierung) beschrieben. Dazu gehören der Escape error, Parity error, Credit Error und Disconnect error. Punkt 1.2.7 bezieht sich ausschließlich auf einen Link-Fehler bei bestehendem Link. Der Character sequence error, also ein Fehler bei der Reihenfolge empfangener Tokens während der Initialisierung (siehe Fehlerkategorie I), wird mit dem nächsten Punkt abgedeckt. Mit dem vorletzten Punkt 1.2.9 soll das Verhalten des IP-Cores in Bezug auf das „Exchange of Silence“ Protokoll geprüft werden. Der Unterschied zum ersten Punkt ist, dass hier der Fokus auf dem Timing liegen soll. Der letzte Punkt bezieht sich spezifisch auf den SpaceWire Light, der Flags zur Anzeige eines Link-Fehlers besitzt.
Die dritte Kategorie soll Randbedingungen und Ausnahmefälle abdecken, die im SpaceWire Standard definiert sind. Sie entsprechen nicht dem normalen Betrieb des IP-Cores. Der erste Punkt bezieht sich auf gleichzeitige Signalübergange auf der Data- und Strobe-Leitung, was in der Praxis durchaus vorkommen kann. In diesem Fall soll sich der Receiver des IP-Cores nicht aufhängen, also der Fehler toleriert werden. Bei den Punkten 1.3.2, 1.3.4 und 1.3.5 geht es um die Erkennung eines Paritätsfehlers (nach dem Empfang des ersten NULLs) bzw. eines Disconnect errors während der Link-Initialisierung. Der Punkt 1.3.3 bezieht sich auf das Handling leerer Pakete. Der letzte Punkt beschreibt Ausnahmefehler, bei denen z.B. Fehler in empfangenen EOPs oder EEPs oder in gesendeten NULLs oder FCTs auftreten. Diese führen in den ersten beiden Fällen zum Verlust empfangener Pakete und in den letzten beiden Fällen zum Verwurf von teilweise gesendeten Paketen (für Details siehe Zusammenfassung im Anhang).

3.3 UVM-Testbench Grundstruktur

Die folgenden Abschnitte sollen einen Überblick über die Grundstruktur der UVM-Testbench geben. Im ersten Abschnitt wird die Entwicklungsrichtlinie erläutert, womit die Grundlage für die Entwicklung der Testbench gemeint ist. Der Abschnitt 3.3.2 beschreibt die Instanziierung des SpaceWire Light auf Top-Level Ebene und dessen Konfiguration. Die darauf folgenden Abschnitte 3.3.3 und 3.3.4 enthalten Informationen zu den einzelnen Komponenten und zu Möglichkeiten der Konfiguration der Testbench.

3.3.1 Entwicklungsrichtlinie

Das UVM-Cookbook bietet eine Vielzahl an Code-Beispielen, die dem Leser zum aktuellen Thema an geeigneter Stelle direkt im Text präsentiert werden und das Verständnis von UVM erleichtern

sollen. Die Beispiele sind bis auf wenige Ausnahmen recht allgemein gehalten und manchmal auch lückenhaft. Sie vermitteln aber dennoch eine Idee, wie der Code für eigene Zwecke aufzubauen ist. Das Cookbook in PDF-Form ist letztendlich eine Darstellungsform des online zur Verfügung stehenden UVM-Cookbooks auf der Seite der Mentor Verification Academy. Dort lassen sich auch diverse Code-Beispiele zu den einzelnen Kapiteln, beispielsweise zu Sequences, herunterladen. Zum Aufbau einer Testbench Grundstruktur ist auch ein (mehr oder weniger) vollständiges Block-Level Testbench Beispiel vorhanden. Dieses Beispiel und online Code-Beispiele sowie einige Beispiele aus dem (alten) UVM-Cookbook bildeten die Grundlage zur Entwicklung der Grundstruktur der Testbench für SpaceWire IP-Cores.
Einige Zeit nach dessen Fertigstellung erschien eine neue Version des offline Cookbooks, da sich die Online-Inhalte im Laufe der Zeit geändert hatten. Aus diesem Grund unterscheiden sich diese Inhalte von den Inhalten der Vorgängerversion des offline Cookbooks an diversen Stellen. Das neue UVM-Cookbook verwendet in den Beispielen zum Aufbau der Testbench-Grundstruktur als virtuelle Interfaces implementierte BFMs zur Kommunikation zwischen Testbench und DUT . Dadurch sind Komponenten wie Driver und Monitor portabler, wodurch sie leichter in anderen Verifikationsumgebungen integrierbar sind. Da das neue UVM-Cookbook wegen seiner späten Verfügbarkeit nicht als Grundlage für die Entwicklung der Grundstruktur diente, verwendet die Testbench keine BFMs.
Vor Beginn der eigentlichen Programmierung wurde eine Ordnerstruktur für die einzelnen Dateien, aus denen die Testbench bestehen sollte, erstellt. Diese orientiert sich an Vorgaben aus dem UVM-Cookbook und an der gegebenen Ordnerstruktur des oben genannten Block-Level Testbench Beispiels. Danach wurden Dummys für die einzelnen Testbench-Komponenten in Form leerer Dateien in den jeweils zugehörigen Ordnern erstellt. Für jede Klasse wird eine eigene Datei angelegt. Für jede Komponente und deren Sub-Komponenten existiert ein separater Ordner. Auch die Dateinamen und Dateiendungen folgen gewissen Vorgaben. Interfaces, Module und Packages, die zum Zusammenfassen von Komponenten auf einer hierarchischen Ebene dienen, erhalten stets die Endung .sv und Komponenten bzw. Sub-Komponenten die Endung .svh. Namen von Packages enden stets mit „_pkg“, usw. Wenn die Ordnerstruktur sowie die ersten Dateien vorhanden sind, wird mit dem Erstellen der Simulationsscripts zum Kompilieren des Testbench-Designs, zur Definition der Simulationsparameter für den Simulator und zum Start der Tests fortgefahren. Falls im Laufe der Zeit noch weitere Dateien bzw. Komponenten dazu kommen, sind diese an entsprechender Stelle in die Struktur der UVM-Testbench zu integrieren und Packages zu erweitern.
Danach folgt die schrittweise Entwicklung der Testbench-Grundstruktur, beginnend mit dem Entwurf des Top-Level Testbench Moduls auf oberster Ebene und der drei Interfaces sowie der Instanziierung des SpaceWire Light. Der hierarchischen Struktur von UVM entsprechend wird als nächstes der Basistest zur Konfiguration der Testumgebung und zum Verbinden der einzelnen Komponenten untereinander aufgebaut. Es folgt der Aufbau der Env und des Env-Konfigurationsobjekts sowie der Grundstruktur des Scoreboards. Der nächste Schritt liegt in der Entwicklung der drei Agents. Da sowohl der FIFO-Agent als auch der Time-Code Agent weniger umfangreich als der Data-Strobe Agent ist, beginnt man z.B. mit der Entwicklung des FIFO-Agents. Hier fängt man am besten mit dem Driver des Agents an. Erst danach sollte mit der Entwicklung des zugehörigen Monitors fortgefahren werden, da dieser oft ähnlich wie der Driver aufgebaut ist. Sind

beide Komponenten eines Agents fertig, folgt das entsprechende Sequence Item. Mit den Sequence Items lassen sich dann die einzelnen Sequences (FIFO Sequences, Time-Code Sequences und Data-Strobe Sequences) zur Erzeugung von Stimuli aufbauen. Darauf folgt schließlich die Definition der abgeleiteten Tests zum Starten der Sequences und zur Verifikation des SpaceWire IP-Cores.

3.3.2 SpaceWire Light Instanziierung und Konfiguration

Der SpaceWire Light IP-Core bzw. die Spwstream Instanz wird im Top-Level Testbench Modul der UVM-Testbench instanziiert. Die Konfiguration des SpaceWire Light IP-Cores erfolgt über Parameter im Testparameter-Package (spw_test_params_pkg). Diese sind über die Spwsteam Instanz mit den Generics des SpaceWire Light verbunden. Die folgende Tabelle gibt eine Übersicht über die Konfigurationsparameter und die entsprechenden Generics.

Tabelle 7: Konfiguration des SpaceWire Light IP-Cores über das Testparameter-Package

Parameter	SpaceWire Light Generic	Beschreibung
S_CLK_FREQ	sysfreq	Systemfrequenz in Hz
TX_CLK_FREQ	txclkfreq	Sendefrequenz in Hz (impl_fast)
RX_CLK_FREQ	/	Samplingfrequenz in Hz (impl_fast)
RX_CHUNK	rxchunk	Bit Einheiten (Receiver Front-End)
RX_IMPL	rximpl	Receiver Front-End Implementierung
TX_IMPL	tximpl	Transmitter Implementierung
RX_FIFO_SIZE	rxfifosize_bits	Größe des Rx-FIFOs
TX_FIFO_SIZE	txfifosize_bits	Größe des Tx-FIFOs

Für RX_CHUNK, RX_FIFO_SIZE und TX_FIFO_SIZE existieren Grenzwerte (siehe dazu Abschnitt 2.3.3.2). Der Parameter S_CLK_FREQ wird auch für die Clocking Blocks im FIFO- und Time-Code Interface verwendet. Die Tabellen 8, 9 und 10 zeigen die Verbindung der Signale der Spwstream Instanz des SpaceWire Light IP-Cores über die gleichnamigen Signale der Interfaces der Testbench. Das Signal des Systemtakts sowie die Signale der beiden separaten Takte für die schnellen Implementierungen (impl_fast) des Receiver Front-Ends (rxclk) und des Transmitters (txclk) des SpaceWire Light sind auf oberster Testbenchebene direkt verbunden.

Tabelle 8: Signale des Time-Code Interfaces (spw_tc_if) der UVM-Testbench

Signal	Beschreibung	Input/Output
clk	Systemtakt (TC_IF)	In
tick_in	Signal zum Senden eines Time-Codes	Out
ctrl_in	Control flags des zu sendenden Time-Codes	Out
time_in	Wert des zu sendenden Time-Codes	Out
tick_out	Signal zur Anzeige eines empfangenen Time-Codes	In
ctrl_out	Control flags eines empfangenen Time-Codes	In
time_out	Wert eines empfangenen Time-Codes	In

Tabelle 9: Signale des Data-Strobe Interfaces (spw_ds_if) der UVM-Testbench

Signal	Beschreibung	Input/Output
dsclk	Data-Strobe Sampling-Takt (DS_IF)	In
rst	System-Reset des IP-Cores	In
spw_di	Data-In Signal	Out
spw_si	Strobe-In Signal	Out
spw_do	Data-Out Signal	In
spw_so	Strobe-Out Signal	In

Tabelle 10: Signale des FIFO Interfaces (spw_fifo_if) der UVM-Testbench

Signal	Beschreibung	Input/Output
clk	Systemtakt (FIFO_IF)	In
autostart	Option zum automatischen Link-Aufbau	Out
linkstart	Anfrage zum Aufbau eines Links	Out
linkdis	Link trennen / keinen neuen Link aufbauen	Out
txdivcnt	Taktteilerwert (Transmitter)	Out
txwrite	N-Char in das Tx-FIFO schreiben	Out
txflag	Definiert txdata als Datenbyte, EOP oder EEP	Out
txdata	Zu sendendes N-Char	Out
rxread	N-Char aus dem Rx-FIFO lesen	Out
txrdy	Tx-FIFO des IP-Cores ist nicht voll	In
txhalff	Tx-FIFO des IP-Cores ist min. halb voll	In
rxvalid	Rx-FIFO des IP-Cores enthält Daten	In
rxhalff	Rx-FIFO des IP-Cores ist min. halb voll	In
rxflag	Definiert rxdata als Datenbyte, EOP oder EEP	In
rxdata	Empfangenes N-Char	In
started	Link-FSM des IP-Cores im Zustand Started	In
connecting	Link-FSM des IP-Cores im Zustand Connecting	In
running	Link-FSM des IP-Cores im Zustand Run	In
errdisc	Disconnect error (RxErr)	In
errpar	Parity error (RxErr)	In
erresc	Escape error (RxErr)	In
errcred	Credit error	In

Aktuell besteht nur über das Data-Strobe Interface die Möglichkeit, einen Reset des IP-Cores durchzuführen.

3.3.3 Testbench-Komponenten

Dieser Abschnitt soll eine Übersicht über die entwickelten Testbench-Komponenten geben. Die Entwicklung der Agents, der Sequences und der Tests ist ab dem Abschnitt 3.4 im Detail beschrieben. Die zugrunde liegenden Dateien der Testbench beginnen im Namen stets mit dem Präfix „spw_“. In der Datei „tb_filelist.txt“ im Root-Verzeichnis der Testbench sind die Dateien mit ihrem Namen und der bestehenden Ordnerstruktur entsprechend aufgelistet.

Das **Top-Level Testbench Modul** befindet sich in der Datei „spw_tb_top.sv“. Der Hierarchie einer Block-Level Testbench entsprechend wird darin das UVM-Package, das Test-Library Package mit den erweiterten Tests und das Testparameter-Package importiert. Außerdem sind hier die **Macros** („spw_macros“) und Definitionen bezüglich des Timings im Simulator enthalten („timescale.v“). Zur Generierung der Takte sind Bit-Variablen definiert. Das event tests_finished dient als Indikator für das Ende der Tests und löst die Deaktivierung der Taktgeneratoren aus. Neben dem DUT sind auch die Interfaces auf dieser Ebene instanziiert. Über einen Initial-Block wird ein einmaliger Reset des IP-Cores ausgelöst. Ein weiterer UVM Initial-Block setzt die virtuellen Interface-Handles in die uvm_config_db und startet dann die Tests. Der dritte Initial-Block enthält die Taktgeneratoren.
Der **Test** oder Basistest (Datei „spw_test_base.svh“) wird zur Konfiguration der Env und der drei Agents (Data-Strobe Agent, FIFO Agent und Time-Code Agent) verwendet und dient als Grundlage für die später folgenden abgeleiteten Tests. Vor der Konfiguration werden alle vier Konfigurationsobjekte angelegt und danach in die uvm_config_db gesetzt. Das **Test-Library Package** („spw_test_lib_pkg“) importiert alle Packages von Sub-Komponenten sowie das **Testparameter-Package** („spw_test_params_pkg“) mit Konfigurationsparametern für den IP-Core und die Testbench und enthält die abgeleiteten Tests (per ’include).
Die **abgeleiteten Tests** der Testbench („spw_norm_op_test“ und „spw_err_op_test“) erweitern den Basistest. Mit diesen Tests wird später die Funktion des SpaceWire Light IP-Cores überprüft.
Die **Env** der Testbench liegt in der Datei „spw_env“. In Abhängigkeit der Konfiguration der Testumgebung werden in der Env das verschachtelte Konfigurationsobjekt aufgebaut und Handles für die Sub-Komponenten (die drei Agents und das Scoreboard) angelegt. Das **Env-Konfigurationsobjekt** („spw_env_config“) besitzt Handles für die Konfigurationsobjekte der Agents und für das Scoreboard sowie Variablen die festlegen, welche Sub-Komponenten im Testfall aufzubauen sind. Das **Env-Package** („spw_env_pkg“) dient zur Zusammenfassung der Sub-Komponenten auf einer Ebene.
Das **Scoreboard** der Testbench („spw_scoreboard“) ist bislang nur als eine Art Dummy vorhanden. Darin sind lediglich die benötigten Variablen zur Kommunikation des Scoreboard mit den Monitoren der Agents auf TLM-Ebene definiert. Dies geschieht in Form von Request- und Response-Items, die über die UVM Analyse-Ports der Agents zum Analyse-FIFO des Scoreboards gelangen. Während der Connect-Phase wird in der Env die Verbindung zwischen den Analyse-Ports der Agents und des Scoreboards geschaffen.
Zur Erzeugung von Stimuli sind **Sequences** („xxx_seq“) erstellt worden. Diese Sequences generieren zu sendende Pakete und Time-Codes in Form von Sequence Items, die über die Sequencer der Agents an die jeweiligen Driver gesendet werden. Diese setzen die Sequence Items dann in Pin-Level Transaktionen um. Es sind Sequences für alle drei Interfaces vorhanden. Das **Sequence-Library Package** („spw_seq_lib_pkg“) importiert per Default die Packages der Agents und der Env und enthält die Sequences (per ’include).
Die **Sequence Items** („spw_xxx_seq_item“), von denen es jeweils eines pro Agent gibt, dienen dem Transfer von Daten sowohl in Sende- als auch Empfangsrichtung (request and response).
Auf der untersten Ebene der Hierachie befinden sich dann die drei **Agents** („spw_xxx_agent“). Die Agent-Klassen selbst legen lediglich Handles für die Sub-Komponenten des jeweiligen Agents an (Build-Phase), verbinden die Monitore mit den Analyse-Ports der Agents und mit den virtuellen

Interfaces und verbinden die Driver mit den Sequencern sowie ebenfalls mit den Interfaces (Connect-Phase). Jeder Agent besitzt ein zugehöriges **Interface** („spw_xxx_if“), über das die Port-Signale des IP-Cores mit der Testbench verbunden sind. Außerdem verfügt jeder Agent über einen **Sequencer** („spw_xxx_sequencer“) zum Transfer von Sequence Items zum entsprechenden Driver. Die **Monitore** der Agents („spw_xxx_monitor“) beobachten die Pin-Level Aktivitäten auf dem jeweiligen Interface und geben Informationen über gesendete und empfangene Daten in Form von Sequence Items zu Analysezwecken an das Scoreboard weiter. Die **Driver** der Agents („spw_xxx_driver“) setzen die über die Sequences generierten Sequence Items in Pin-Level Transaktionen um und senden dadurch Daten in Form von Paketen und Time-Codes an den IP-Core. Die **Agent-Konfigurationsobjekte** („spw_xxx_agent_config“) stellen virtuelle Interface-Handles zur Verfügung. Weiterhin sind Variablen zur Konfiguration von Sub-Komponenten vorhanden. Über das Agent-Konfigurationsobjekt lassen sich auch Parameter zu Sub-Komponenten eines Agents transportieren, die im Testparameter-Package definiert sind und beispielsweise vom Driver benötigt werden. Dies ist beim DS Driver der Fall. Die **Agent-Packages** („spw_xxx_agent_pkg“) vereinen schließlich alle Klassen der Sub-Komponenten, das zugehörige Sequence Item und das Konfigurationsobjekt auf einer Ebene. Im Verzeichnis „Docs“ der beigelegten DVD liegt die htm-Datei „spw_testbench_components“, die die oben beschriebenen Testbench-Komponenten in tabellarischer Form enthält.

3.3.4 Testbench-Konfiguration

Neben den Optionen zur Konfiguration des SpaceWire Light IP-Cores, beschrieben im Abschnitt 3.3.2, gibt es auch einige wenige Optionen, die Testbench selbst zu konfigurieren.
Das Data-Strobe Interface besitzt einen separaten Takt (dsclk), der für das synchrone Oversampling im Receiver der Link-FSM der Testbench verwendet wird. Die Samplingfrequenz beträgt aktuell 800 MHz. Dadurch ist es möglich, den Data-Strobe codierten Bitstream mit Übertragungsraten bis 400 MBit/s zu decodieren. Sollte in einer zukünftigen Version des Standards eine höhere Maximalrate definiert sein, ist die Samplingfrequenz entsprechend anzupassen. Dazu ist im Testparameter-Package der Parameter DS_CLK_FREQ vorhanden, über den sich die Samplingfrequenz (in Hz) einstellen lässt. Falls der IP-Cores aufgrund seines Designs hinsichtlich des maximalen Takts limitiert ist und daher nur mit geringerer Rate (weniger als 400 MBit/s) senden kann oder das Verhalten des Cores nur bis zu einem bestimmten Takt getestet werden soll, ist natürlich eine geringere Samplingfrequenz nötig. Die restlichen Parameter des Package sind durch die eingestellten Taktfrequenzen definiert und entsprechen der Dauer einer halben Taktperiode der Takte innerhalb der Testbench. Diese Parameter müssen unverändert bleiben. Die Taktgeneratoren, die diese Parameter benötigen, befinden sich im Top-Level Testbench Modul.
Für das Timing innerhalb der Link-FSM im DS Driver existieren Parameter, die im Basistest bei der Konfiguration des DS Agents im zugehörigen Konfigurationsobjekt abgelegt werden. Über das Konfigurationsobjekt hat der DS Driver dann Zugriff auf diese Variablen. Deren Werte repräsentieren eine bestimmte Anzahl an Taktzyklen, abgeleitet vom Parameter DS_CLK_FREQ, die zur Darstellung der drei Timeouts dienen. Diese sind das Timeout zur Erkennung eines Disconnect errors (0,85 μs) sowie die beiden Timeouts (6,4 μs und 12,8 μs) für Zustandsübergänge. Außerdem wird über das Konfigurationsobjekt der Wert des Taktteilers zur Bestimmung der

Übertragungsrate während der Link-Initialisierung (10 MBit/s) weitergereicht.
Zur Konfiguration der Testumgebung während der Build- und Connect-Phasen der Testbench sind ein paar Macros programmiert worden (Datei „spw_macros.svh“). Dabei handelt es sich um Konfigurationsmacros die dazu dienen, Handles für Konfigurationsobjekte und die virtuellen Interfaces aus der UVM-Konfigurationsdatenbank zu beziehen. Für den Fall, dass diese nicht ordnungsgemäß in der config_db abgelegt sind, wird automatisch eine Fehlermeldung ausgegeben. Ein weiteres Macro dient zum Erstellen von Sequences während der Run-Phase der Tests.

3.4 FIFO Agent Design

3.4.1 FIFO Driver

Der FIFO Driver der UVM-Testbench steuert hauptsächlich die Schreib- und Lesevorgänge auf dem Tx- bzw. Rx-FIFO des SpaceWire Light IP-Cores. Weiterhin agiert er auf den Signalen des FIFO-Interfaces, die die Link-Initialisierung und das Beenden einer bestehenden Verbindung seitens des IP-Cores kontrollieren sowie den Taktteilerwert des Transmitters bei der schnellen Implementierung des SpaceWire Light festlegen. Dazu gehören die Signale autostart, linkstart, linkdis und txdivcnt. Die als Request Items (req_item) bezeichneten FIFO Sequence Items definieren Pakete mit variabler Länge bestehend aus data characters, EOPs oder EEPs , die zum Senden in das Tx-FIFO des IP-Cores geschrieben werden sollen. Das FIFO Sequence Item besitzt ein weiteres Flag namens fifo_write. Mit diesem wird grundsätzlich entschieden, ob mit einem aktuell vorhandenen Sequence Item ein Paket zu senden ist und/oder ob sich die Signale autostart, linkstart, linkdis oder txdivcnt ändern. Zur Erinnerung: Jedes FIFO Sequence Item definiert stets ein komplettes Paket. Außerdem enthält das Sequence Item ein 8 Bit breites Array (eop_eep_tx), dessen Wert entweder 0x00 zum Schreiben eines EOPs oder 0x01 zum Schreiben eines EEPs ist. Im Normalfall ist der Wert 0x00, da keine fehlerhaften Pakete in das Tx-FIFO geschrieben werden.
Der FIFO Driver besteht im Kern aus zwei Tasks. Der spw_fifo_write()-Task steuert dem Namen entsprechend die Schreibvorgänge auf dem Tx-FIFO und der spw_fifo_read()-Task die Lesevorgänge auf dem Rx-FIFO. Zu Beginn der Run-Phase setzt der Driver die Interface-Signale auf Default-Werte (kein Schreibvorgang, Link-Initialisierung nicht starten, etc.) und führt dann nach dem initialen System-Reset des IP-Cores die beiden Tasks zeitgleich per fork-join aus. Der Read-Task ist aktuell so aufgebaut, dass Daten aus dem Rx-FIFO gelesen werden, sobald welche vorhanden sind. Dies hat zur Folge, dass sich das Rx-FIFO niemals füllt. Optional könnte man z.B. mit den Lesevorgängen so lange warten, bis das Rx-FIFO mindestens halb voll ist (rxhalff Signal). Oder man randomisiert den Mechanismus und liest nur alle paar Takte. Im Laufe der weiteren Entwicklung der Testbench im Hinblick auf die vollständige Verifikation des SpaceWire IP-Cores sind hier ein paar Änderungen notwendig, um das Verhalten des IP-Cores bei vollem Rx-FIFO überprüfen zu können. Innerhalb des Read-Tasks wird momentan mit jeder steigenden Taktflanke geprüft, ob rxvalid gesetzt ist und somit Daten im Rx-FIFO liegen. Ist dies der Fall, wird rxread gesetzt, um die Daten zu lesen. Im Write-Task wird innerhalb einer forever-Schleife mit jeder steigenden Taktflanke zunächst per try_next_item() versucht, ein neues Sequence Item aus dem Request-FIFO des Sequencers zu beziehen. Ist keines vorhanden, wird der Versuch im nächsten Takt wiederholt. Falls eines vorhanden ist, werden die Signale autostart, linkstart, linkdis und txdivcnt den Feldern des Sequence

Items entsprechend geändert und das Item beendet, falls fifo_write nicht gesetzt ist. Sollte fifo_write = 1 zum Schreiben eines Pakets sein, wird so lange gewartet, bis txrdy = 1 und das Tx-FIFO somit bereit ist, Daten aufzunehmen (das Tx-FIFO ist nicht voll). Danach sorgt eine foreach-Schleife dafür, dass alle Datenbytes des Pakets mit jedem zweiten Takt per txwrite in das FIFO geschrieben werden, sofern das Tx-FIFO aktuell nicht voll ist. Ansonsten wird mit dem Schreiben des nächsten Bytes gewartet. txwrite wird jeweils nur eine Taktperiode lang gesetzt, da im nächsten Takt das Tx-FIFO bereits voll sein kann, wodurch txwrite keinen Effekt mehr hat und zu sendende Daten verloren gehen. Aus diesem Grund wird ein nicht volles FIFO nur mit jedem zweiten Takt beschrieben. Das ist absolut kein Problem, da zu sendende Daten ohnehin wesentlich schneller in das Tx-FIFO gelangen, als sie dieses verlassen können (serielles Senden der Daten).
Befinden sich alle Bytes eines Pakets im FIFO, ist es im nächsten Schritt mit einem EOP (oder EEP) abzuschließen. Dazu wird im nächsten Takt neben txwrite auch txflag eine Taktperiode lang gesetzt und gleichzeitig der Wert von eop_eep_tx in das Tx-FIFO geschrieben, sofern es aktuell nicht voll ist. Ansonsten wird wieder gewartet. Das Handshake-Verfahren für Sequence Items wird danach per item_done()-Methode im FIFO Driver beendet und im nächsten Takt versucht, ein neues Sequence Item zu lesen bzw. Paket in das Tx-FIFO zu schreiben.

3.4.2 FIFO Monitor

Der FIFO Monitor der UVM-Testbench ist sozusagen das Gegenstück zum FIFO Driver. Zu den Aufgaben dieses Monitors gehören die Beobachtung der Aktivitäten auf dem FIFO-Interface hinsichtlich der Schreib- und Lesevorgänge auf den beiden FIFOs sowie der Aufbau von Request- und Response Items. Diese gelangen dann zum Vergleich von gesendeten und empfangenen Daten auf TLM-Ebene zum Scoreboard. Die aufgebauten Request Items speichern die Daten/Pakete, die der Driver in das Tx-FIFO des IP-Cores schreibt. Dazu muss der Monitor prinzipiell diejenigen Signale beobachten, die der Driver zum Schreiben der Pakete in das FIFO verwendet. Die Response Items speichern die Daten/Pakete, die aufgrund des in Abhängigkeit des rxvalid Signals gesetzen rxread Signals im Driver aus dem Rx-FIFO gelesen werden. Da rxread im FIFO Driver immer dann gesetzt wird, wenn rxvalid =1 ist, muss der Monitor prinzipiell nur die Signale rxread und rxflag beobachten. Die Grundlage zum Speichern der Daten/Pakete sind alle Felder bzw. Variablen des FIFO Sequence Items, die den Interface-Signalen für Schreib- und Lesevorgänge auf den beiden FIFOs zuzuordnen sind.
Der FIFO Monitor besteht im Kern aus zwei Tasks und einer Funktion, die Request- und Response Items über zwei UVM Analyse-Ports des Monitors in das Analyse-FIFO des Scoreboards schreibt. Die Funktion notify_transaction() besitzt zwei Übergabeparameter, das Sequence Item und einen String, der das Item als Request- oder Response Item identifiziert. Die Art des Items entscheidet dann, über welchen der beiden Analyse-Ports geschrieben wird. Der String ist entweder „in“ für ein Request Item oder „out“ für ein Response Item. Entsprechend wird ein Request Item über den Port „ap_i“ und ein Response Item über den Port „ap_o“ gesendet. Dadurch lassen sich Request- und Response Items voneinander trennen. Die Richtungen „in“ und „out“ beziehen sich auf die Ein- bzw. Ausgänge des IP-Cores. Der spw_monitor_req()-Task dient seinem Namen entsprechend zum Aufbau von Request Items und der spw_monitor_rsp()-Task zum Aufbau von Response Items. Zu Beginn der Run-Phase wartet der Monitor, bis der initiale System-Reset des IP-Cores abgeschlossen

ist. Danach werden die beiden Tasks zeitgleich per fork-join ausgeführt. Im Request-Task wird innerhalb einer forever-Schleife zunächst ein Request Item per create()-Methode erstellt. Der Monitor wartet jetzt so lange, bis das Signal txwrite zum Schreiben von Daten in das Tx-FIFO gesetzt ist. Dies ist nur dann der Fall, wenn seitens des Drivers über ein Request Item der Befehl zum Schreiben (fifo_write) gegeben wird und das Tx-FIFO aktuell nicht voll ist (txrdy = 1). Durch den ersten Schreibvorgang gelangt also das erste Datenbyte eines Pakets in das FIFO (txdata). Der Monitor setzt dieses Byte dann in ein dynamisches Byte-Array (txdata) im Request Item. Über eine while-Schleife wird dann das Array des Items mit jedem Takt in dem txwrite = 1 ist, um ein weiteres in das FIFO geschriebene Datenbyte erweitert. Dies geschieht so lange, bis txflag gesetzt ist und das Ende des Pakets anzeigt. In diesem Fall gelangt als nächstes ein EOP oder EEP in Form eines Bytes in das Tx-FIFO. Dieses wird in der Variable eop_eep_tx gespeichert, wodurch das komplette Paket jetzt im Request Item liegt. Anschließend setzt die Funktion notify_transaction() das Request Item in das Analyse-FIFO des Scoreboards, bevor ein neues Item für das nächste Paket erzeugt wird.
Der Response-Task enthält ebenfalls eine forever-Schleife und beginnt mit dem Erstellen eines Response Items für das nächste seitens des IP-Cores empfangenen Pakets. Da durch den FIFO Driver gesteuert Paketdaten aus dem Rx-FIFO gelesen werden, sobald welche vorhanden sind, wird im Monitor stets auf die positive Flanke des rxvalid Signals gewartet (das FIFO enthält dann wieder ein empfangenes Byte). Soll später im Hinblick auf eine vollständige Verifikation des IP-Cores das Verhalten bei vollem Rx-FIFO überprüft werden, sind an dieser Stelle einige Änderungen im Code notwendig, da rxvalid dann über einen längeren Zeitraum logisch 1 ist. Das Signal rxflag zeigt an, ob als nächstes ein Datenbyte oder ein EOP bzw. EEP im Rx-FIFO liegt. Dem Flag entsprechend wird danach entweder das dynamische Byte-Array (rxdata) im Response Item um ein gelesenes Datenbyte erweitert (rxflag = 0) oder ein EOP oder EEP in Form eines Bytes in der Variable eop_eep_rx des Items gespeichert (rxflag = 1). Dies geschieht für alle Bytes eines Pakets innerhalb einer fußgesteuerten Schleife, die das Flag rx_flag abfragt. rx_flag wird genau dann gesetzt, sobald innerhalb eines Schleifendurchlaufs rxflag (ohne Unterstrich) das Ende des Pakets anzeigt und folglich das gelesene EOP oder EEP in eop_eep_rx vorhanden ist. Zu diesem Zeitpunkt liegt also ein komplett empfangenes Paket im Response Item. Die Schleife wird danach verlassen und abschließend der Funktion notify_transaction() das Response Item übergeben, das somit in das Analyse-FIFO des Scoreboards gelangt. Zum Lesen des nächsten empfangenen Pakets ist dann im neuen Durchlauf der forever-Schleife rx_flag zurück zu setzen und ein neues Response Item zu erstellen.

3.4.3 FIFO Sequence Item

Das FIFO Sequence Item dient zum Speichern von kompletten Paketen mit variabler Größe, die entweder in das Tx-FIFO des IP-Cores geschrieben oder aus dem Rx-FIFO gelesen werden. Das Sequence Item wird entweder vom FIFO Driver in Form von Request Items oder vom FIFO Monitor in Form von Request- und Response Items verwendet. Die im Item enthaltenen Variablen für Request Items besitzen stets ein voran gestelltes „rand“ und können somit per randomize()-Methode erzeugte pseudozufällige Daten speichern. Dies ermöglicht eine randomisierte Stimuli-Erzeugung über den Driver, wenn dieser die zufälligen Daten aus den Request Items in Pin-Level Transaktionen

umsetzt. Zu den randomisierbaren Variablen gehören all diejenigen Variablen des Items, die den Signalen des FIFO-Interfaces entsprechen, welche darin aus der Sicht der Testbench als Ausgänge definiert sind. Es handelt sich dabei also um die Eingänge des IP-Cores. Diese Variablen sind dieselben, die im Monitor in den Request Items für gesendete Pakete genutzt werden. Zu den nicht randomisierbaren Variablen gehören analog alle Variablen des Items, die denjenigen Signalen des FIFO-Interfaces zugehörig sind, welche darin aus der Sicht der Testbench als Eingänge definiert sind. Sie entsprechen daher den Ausgängen des IP-Cores. Diese Variablen nutzt lediglich der Monitor zum Speichern der Pakete aus dem Rx-FIFO.
Die folgende Tabelle bietet eine Übersicht über die Felder des FIFO Sequence Items.

Tabelle 11: Variablen des FIFO Sequence Items

Variable	Datentyp	Randomisierung	Richtung (Testbench)
autostart	Bit	rand	Out
linkstart	Bit	rand	Out
linkdis	Bit	rand	Out
[7:0] txdivcnt	Packed Bit-Array	rand	Out
txwrite	Bit	rand	Out
fifo_write	Bit	rand	Out
txflag	Bit	rand	Out
[7:0] txdata []	Dyn. Bit-Array	rand	Out
rxread	Bit	rand	Out
[7:0] eop_eep_tx	Packed Bit-Array	rand	Out
txrdy	Bit	non-rand	In
txhalff	Bit	non-rand	In
rxvalid	Bit	non-rand	In
rxhalff	Bit	non-rand	In
rxflag	Bit	non-rand	In
[7:0] rxdata []	Dyn. Bit-Array	non-rand	In
[7:0] eop_eep_rx	Packed Bit-Array	non-rand	In
started	Bit	non-rand	In
connecting	Bit	non-rand	In
running	Bit	non-rand	In
errdisc	Bit	non-rand	In
errpar	Bit	non-rand	In
erresc	Bit	non-rand	In
errcred	Bit	non-rand	In

Der Vollständigkeit halber enthält das FIFO Sequence Item auch Variablen für diejenigen Signale des FIFO-Interfaces, die aktuell aus Sicht des Drivers und Monitors keine Relevanz haben und daher weder gelesen noch geschrieben werden. Dazu gehören beispielsweise die Flags zur Anzeige der Zustände der Link-FSM im SpaceWire Light.

3.5 TC Agent Design

3.5.1 Erweiterung um TC Agent und TC-IF

Das Spwstream-Interface des SpaceWire Light IP-Cores besitzt separate Signale zum Senden von Time-Codes und zur Anzeige bzw. Darstellung empfangener Time-Codes. Zu Beginn der Testbench-Entwicklung waren diese Signale noch über das FIFO-Interface mit der Testbench verbunden. Es stellte sich jedoch nach einiger Zeit heraus, dass sich die Koordination beim gleichzeitigen Senden und Empfangen von Paketen und Time-Codes über nur einen Driver bzw. einen Monitor in Bezug auf die Sequence Items schwierig gestaltete. Um diese Vorgänge einfacher zu steuern und Pakete und Time-Codes getrennt voneinander betrachten zu können, wurde ein dritter Agent inklusive zugehörigem Interface ausschließlich für Time-Codes hinzugefügt. Das Time-Code Interface besitzt daher lediglich die sechs Signale zum Senden/Empfangen der Time-Codes und das Signal für den Systemtakt.
Gemäß der Definition zur Verteilung der Systemzeit im SpaceWire Standard ist es eigentlich nicht nötig, Time-Codes durch den Benutzer festlegen zu lassen, da sich der Wert eines gültigen Time-Codes mit jedem gesendeten Time-Code automatisch um 1 erhöht. Angenommen, der zuletzt gesendete Time-Code habe den Wert 6 (0b000110). Sollte danach ein Time-Code mit dem Wert 7 (0b000111) gesendet werden, gilt nur dieser auf Empfängerseite als gültig, wodurch das tick_out Signal gesetzt wird.

3.5.2 TC Driver

Der TC Driver der UVM-Testbench hat die Aufgabe, die über den SpaceWire Light IP-Core zu sendenden Time-Codes durch die beiden Signale ctrl_in und time_in zu definieren. Mit dem tick_in Signal wird das Senden eines Time-Codes über den Link, bestehend aus dem aktuellen Wert von ctrl_in und time_in, unmittelbar nach dem aktuell zu sendenden Token ausgelöst.
Zu Beginn der Run-Phase werden im Driver die drei Signale auf Default-Werte gesetzt (keinen Time-Code senden). Nach dem initialen System-Reset des IP-Cores ist hier aktuell eine Zeitverzögerung von 25 μs eingebaut. Diese soll sicherstellen, dass der erste Versuch einen Time-Code zu senden erst zum frühstmöglichen Zeitpunkt nach dem initialen Reset unternommen wird, an dem die Link-Initialisierung abgeschlossen sein kann (knapp 20 μs). Erst danach ist es überhaupt möglich, Time-Codes über den Link zu senden. Im Driver ist dann zum Senden der Time-Codes eine forever-Schleife vorhanden. In dieser wird mit der nächsten positiven Taktflanke das nächste Time-Code Sequence Item per get_next_item()-Methode bezogen. Da mit jedem Request Item ein Time-Code gesendet werden soll, setzt der Driver das tick_in Signal auf 1 und weist die Werte von ctrl_in und time_in aus dem Request Item den beiden entsprechenden Interface-Signalen zu. Mit der nächsten positiven Taktflanke ist tick_in dann wieder zurück zu setzen und das Handshake-Verfahren für Sequence Items per item_done()-Methode abzuschließen. Bevor das nächste Sequence Item für den darauf folgenden Time-Code bezogen wird, sorgt eine über das Request Item definierbare Verzögerung für gleichmäßige zeitliche Abstände. Die Zeitabstände zwischen den Anfragen zum Senden von Time-Codes sind ein Vielfaches einer Millisekunde, also 1 ms, 2 ms, usw.. Der Wert eines gesendeten Time-Codes ist stets um 1 größer als der des vorherigen Time-Codes.

3.5.3 TC Monitor

Der Time-Code Monitor ähnelt in seiner Struktur dem FIFO Monitor. Seine Aufgaben sind die Beobachtung der Aktivitäten auf dem Time-Code Interface, die Speicherung der per Driver über den IP-Core gesendeten Time-Codes in Form von Request Items und die Speicherung der vom IP-Core während eines bestehenden Links empfangenen Time-Codes in Form von Response Items. Auch der Time-Code Monitor besitzt wieder zwei UVM Analyse-Ports („ap_i“ und „ap_o“) zum separaten Senden der Sequence Items unterschiedlichen Typs in Richtung Scoreboard über die notify_transaction()-Funktion. Der Monitor besteht im Kern wieder aus den beiden Tasks spw_monitor_req() und spw_monitor_rsp(). Der spw_monitor_req()-Task beobachtet das tick_in Signal, das zum Senden eines Time-codes im Driver gesetzt wird und der spw_monitor_rsp()-Task das tick_out Signal, das einen seitens des IP-Cores empfangenen Time-Code anzeigt.
Zu Beginn der Run-Phase wartet der Monitor zunächst auf die negative Flank des rst Signals. In diesem Fall ist der initiale System-Reset abgeschlossen und der IP-Core beginnt nach ca. 20 μs erstmals mit der Link-Initalisierung, sofern das linkstart und/oder das autostart Signal gesetzt sind. Per fork-join werden jetzt wieder die beiden Tasks zeitgleich gestartet. Im Request-Task wird dann innerhalb einer forever-Schleife zu jeder positiven Taktflanke das tick_in Signal des Interfaces gelesen. Ist tick_in nicht gesetzt, existiert aktuell keine Anfrage zum Senden eines Time-Codes. In diesem Fall wird tick_in im nächsten Takt erneut geprüft. Sollte tick_in = 1 sein, ist im aktuellen Takt ein Time-Code zu senden. Der Monitor erstellt dann per create()-Methode ein neues Request Item und weist die Werte von tick_in, ctrl_in und time_in den entsprechenden Variablen im Sequence Item zu. Danach schreibt die Funktion notify_transaction() das Request Item in das Analyse-FIFO des Scoreboards.
Im Response-Task wird innerhalb einer forever-Schleife zu jeder positiven Taktflanke das tick_out Signal des Time-Code Interfaces beobachtet. Ist tick_out nicht gesetzt, hat der IP-Core im aktuellen Takt keinen Time-Code empfangen. In diesem Fall wird tick_out wieder bei der nächsten steigenden Taktflanke überprüft. Ist tick_out jedoch 1, hat der IP-Core im aktuellen Takt einen Time-Code empfangen und die Signale ctrl_out und time_out dem neuen Wert entsprechend aktualisiert. Der Monitor erstellt dann per create()-Methode das nächste Response Item und weist die Werte von ctrl_out und time_out den entsprechenden Variablen im Response Item zu. Abschließend schreibt die notify_transaction()-Funktion das Response Item über den ap_o-Port per write()-Methode in das Analyse-FIFO des Scoreboards.

3.5.4 TC Sequence Item

Das Time-Code Sequence Item dient zur Speicherung eines einzelnen Time-Codes. Dabei handelt es sich entweder um einen über den SpaceWire IP-Core zu sendenden Time-Code in Form eines Request Items oder um einen vom IP-Core empfangenen Time-Code in Form eines Response Items. Per Request Item gelangen einzelne Time-Codes durch den Driver in den IP-Core. Denselben Itemtyp nutzt der Monitor zum Speichern dieses gesendeten Time-Codes. Das Response Item enthält jeweils einen empfangenen Time-Code. Die Variablen des Items, die für ein Request Item verwendet werden, sind randomisierbar. Dazu gehört zunächst eine Integer-Variable, die den zeitlichen Abstand zwischen zwei zu sendenden Time-Codes in Millisekunden festlegt. Die

verbleibenden Variablen entsprechen dem Namen nach denjenigen Signalen, die im Time-Code Interface zum Senden eines Time-Codes dienen. Für ein Response Item existieren lediglich Variablen entsprechend der Interface-Signale, die als Eingänge definiert sind und empfangene Time-Codes anzeigen bzw. darstellen.
Die folgende Tabelle zeigt eine Übersicht über die Felder des TC Sequence Items.

Tabelle 12: Variablen des TC Sequence Items

Variable	Datentyp	Randomisierung	Richtung (Testbench)
delay	Integer	rand	Out
tick_in	Bit	rand	Out
[1:0] ctrl_in	Packed Bit-Array	rand	Out
[5:0] time_in	Packed Bit-Array	rand	Out
tick_out	Bit	non-rand	In
[1:0] ctrl_out	Packed Bit-Array	non-rand	In
[5:0] time_out	Packed Bit-Array	non-rand	In

Die Variable ctrl_in eines zu sendenden Time-Codes ist laut Standard auf 0 zu setzen. Es ist daher zu erwarten, dass ctrl_out eines empfangenen Time-Codes ebenfalls 0 ist. Obwohl sich der Wert des zu sendenden Time-Codes mit jedem gesendeten Time-Code um 1 erhöhen soll, ist time_in standardmäßig randomisierbar.

3.6 DS Agent Design

Die Entwicklung des Data-Strobe Agents sowie das anschließende Debugging nahm mit Abstand am meisten Zeit in Anspruch, da im Driver der komplette SpaceWire Codec zu implementieren war. Aufgrund seiner Komplexität und wegen der Koordination zwischen der Link-FSM, dem Transmitter und dem Receiver, stellte die Speicherung der über den Link zu sendenden und empfangenen Daten (Pakete und Time-Codes) in Form von Sequence Items eine Herausforderung dar. Dies betraf vor allem das Timing beim Erstellen und Beenden von Request- und Response Items im Monitor, da hier während der Decodierung des Bitstreams detektierte Link-Fehler zum frühzeitigen Beenden eines Items führen (Stichwort Link Error Recovery Schema). Diesbezüglich musste sorgfältig darauf geachtet werden, das Handshake-Protokoll für Sequence Items nicht zu verletzen, was zu Fehlern bei der Kommunikation zwischen Sequencer und Driver bzw. Monitor und Scoreboard geführt hätte. Weiterhin müssen Variablen in den Request- und Response Items vorhanden sein, die ein gesendetes bzw. empfanges Paket als fehlerfrei und damit vollständig (abgeschlossen durch ein EOP) oder als fehlerhaft und daher unvollständig (abgeschlossen durch ein EEP) identifizieren. Letzteres ist zum Beispiel dann der Fall, wenn ein Sequence Item im Monitor frühzeitig beendet wird. Im Driver ist hinsichtlich des Error Recovery Schemas darauf zu achten, im Fehlerfall den verbleibenden Teil eines zu sendenden Pakets, also die verbleibenden Request Items, die ein Paket definieren, zu verwerfen.
Ein weiterer Punkt hinsichtlich des Designs ist die Decodierung der Data- und Strobe-Signale. Zu Beginn der Entwicklung des DS Driver wurde nicht bedacht, dass sich aus Sicht der Testbench die Rx-Datenrate bei bestehender Verbindung in beide Richtungen jederzeit ändern kann, wenn

der SpaceWire IP-Core den Wert des Taktteilers im Transmitter ändert. Anfangs gab es die Idee, den Takt des Receivers dynamisch an den Sendetakt des IP-Cores anzupassen. Dazu sollte die Information über den aktuell eingestellten Taktteilerwert (Signal txdivcnt) an den Driver weitergegeben werden, um dort den entsprechenden Takt für den Receiver in Abhängigkeit von der Sendefrequenz und des Taktteilerwerts zu generieren. Diese Idee wurde jedoch verworfen und das Verfahren des synchronen Oversamplings umgesetzt. Dadurch ist die dynamische Anpassung des Takts nicht mehr nötig. Stattdessen hat man nur einen festen Sampling-Takt zur Decodierung der Signale.
Während der Entwicklung des Drivers konnte außerdem ein grundsätzlicher Fehler bei der Decodierung der Tokens gefunden und korrigiert werden. Dieser bezieht sich auf die Entscheidung, wann ein empfangenes Token gültig ist und wann somit das entsprechende Flag zur Anzeige des Empfangs (z.B. got_FCT) zu setzen ist. Anfangs galt ein Token während der Decodierung gemäß Abbildung 22 bereits als gültig, sobald das letzte zur Identifikation des Tokens notwendige Bit empfangen wurde. In Bezug auf die Paritätsabdeckung (siehe Abbildung 19) ist die Entscheidung, ob ein Token gültig oder ungültig ist, jedoch erst nach dem Empfang des Paritätsbits im nächsten Token möglich, da dieses das vorherige Token abdeckt. Die Fehlerkorrektur erforderte Änderungen in der Vorgehensweise bei der Decodierung sowohl im Driver als auch im Monitor.

3.6.1 DS Driver

3.6.1.1 SpaceWire FSM

Die SpaceWire Link-FSM zur Steuerung der Link-Initialisierung und des Receivers zur Decodierung empfangener Daten und zur Erkennung von Link-Fehlern sowie des Transmitters zum Senden von Tokens seitens der UVM-Testbench ist im DS Driver implementiert. Sie besitzt die gleiche Anzahl an Zuständen mit denselben Namen wie die im Standard definierte FSM. Die diversen Flags, von denen die Link-FSM Gebrauch macht, sind am Anfang der Klasse des Drivers definiert. Zu Beginn der Run-Phase des DS Drivers wird anfangs auf das Ende des initialen System-Resets des IP-Cores gewartet und danach die Link-FSM (spw_link_fsm() Task) zeitgleich mit dem spw_ds_decode() Task zur Decodierung von Data und Strobe per fork-join gestartet. Innerhalb des spw_link_fsm() Tasks werden dann zuerst die Zustände definiert, bevor die Link-FSM innerhalb einer forever-Schleife und einer Case-Anweisung im Zustand ErrorReset beginnt.
Im Zustand **ErrorReset** setzt die FSM zunächst alle zugehörigen Flags bzw. Variablen zurück. Dazu gehören auch der Rx- und Tx credit count für die Flusssteuerung. Per fork-join wird dann der spw_ds_reset() und der spw_timer_6p4() Task ausgeführt. Ersterer setzt das Data- und Strobe-Signal (spw_di, spw_si) kontrolliert zurück; Strobe zuerst, gefolgt von Data mit kurzer Verzögerung. Diese entspricht anfangs einer Taktperiode des Data-Strobe Sampling-Takts und danach der vor einem Reset per txdivcnt eingestellten Tx-Bitrate. Der spw_timer_6p4() Task setzt die 6,4 μs in Form von Taktperioden des Data-Strobe Sampling-Takts als Wartezeit um, bevor der bedingungslose Übergang in den Zustand ErrorWait erfolgt.
Im Zustand **ErrorWait** aktiviert die FSM den Receiver und startet danach die beiden Tasks detect_error_l() und spw_timer_12p8() per fork-join_any. detect_error_l() dient der Erkennung eines

Fehlers der Kategorie I und wird beendet, wenn ein Fehler seitens des Receivers innerhalb von 12,8 μs erkannt wird. Der spw_timer_12p8() Task setzt die 12,8 μs so wie der spw_timer_6p4() Task um. Nach Ablauf dieser Zeit setzt er das timer_12p8_done Flag und wird beendet. Je nachdem, ob zuerst der Timer abläuft oder ein Fehler der Kategorie I erkannt wurde, folgt der Übergang in den Zustand Ready (timer_12p8_done gesetzt) oder ErrorReset (timer_12p8_done nicht gesetzt). Im Zweig für den Übergang nach ErrorReset wird im Fehlerfall vorher geprüft, ob es sich hierbei um einen Character sequence error handelt. Falls ja, gibt es eine Fehlermeldung.
Im Zustand **Ready** werden die beiden Tasks wait_link_en() und detect_error_I() parallel per fork-join_any gestartet. detect_error_I() wird wieder nach der Erkennung eines Fehlers der Kategorie I beendet. wait_link_en() wartet, bis die Bedingung **LE (LinkEnable)** (siehe Abschnitt 2.2.4) zum Starten der Link-Initialisierung erfüllt ist. Das Flag gotNULL entspricht hier dem Flag got_null_i zur Anzeige des Empfangs des ersten NULLs während der Link-Initialisierung. Ist die Bedingung erfüllt, wird das link_en Flag gesetzt und der Task beendet. Je nachdem, welcher der beiden Tasks zuerst beendet wurde, folgt der Übergang in den Zustand Started (link_en = 1) oder ErrorReset (link_en = 0). Im Zweig für den Übergang nach ErrorReset erfolgt wieder eine Fehlermeldung im Falle eines Character sequence errors.
Im Zustand **Started** werden vier Tasks spw_send_char(), wait_got_null(), detect_error_I() und spw_timer_12p8() zeitgleich per fork-join_any gestartet. Die Funktion der beiden zuletzt genannten Tasks ist bekannt. Der Task wait_got_null() wartet seinem Namen nach auf got_null_i = 1. Dieses Flag wird im Receiver nach Erkennung der NULL detection sequence gesetzt. spw_send_char() zum Senden von NULLs wird wiederholt aufgerufen, so lange got_null_i = 0 ist. Nach dem Senden des ersten NULLs ist fnull_sent = 1. Einer der drei Tasks wait_got_null(), detect_error_I() oder spw_timer_12p8() wird jetzt zuerst beendet. Im Fall von wait_got_null() kann der Wechsel in den Zustand Connecting erfolgen, nachdem mindestens ein NULL gesendet (fnull_sent = 1) und ein eventuell aktuell laufender spw_send_char() Task beendet wurde (null_done = 1). Im Fall von detect_error_I() oder spw_timer_12p8() ist entweder ein Fehler innerhalb der 12,8 μs aufgetreten oder der Timer abgelaufen. In jedem Fall folgt dann der Wechsel in den Zustand ErrorReset. Im entsprechenden Zweig erfolgt wieder eine Fehlermeldung bei einem Character sequence error.
Im Zustand **Connecting** starten per fork-join_any die vier Tasks spw_send_char(), wait_got_fct(), detect_error_II() und spw_timer_12p8(). detect_error_II() dient der Erkennung eines Fehlers der Kategorie II und wird beendet, wenn ein Fehler innerhalb von 12,8 μs auftritt. wait_got_fct() wartet seinem Namen nach auf got_fct_i = 1. Dieses Flag ist gesetzt, sobald im Receiver das erste FCT decodiert wurde. Der vierte Task spw_send_char() sendet im Zustand Connecting entweder FCTs oder NULLs. Nach dem Senden des ersten FCTs ist ffct_sent = 1. So lange weder got_fct_i = 1 ist (erstes FCT noch nicht empfangen) noch ein Fehler der Kategorie II auftritt oder die 12,8 μs abgelaufen sind, sendet der Driver in Abhängigkeit des Rx credit counts maximal 7 FCTs und danach nur noch NULLs. Einer der drei Tasks wait_got_fct(), detect_error_II() oder spw_timer_12p8() wird zuerst beendet. Im Fall von wait_got_fct() kann der Wechsel in den Zustand Run erfolgen, nachdem mindestens ein FCT gesendet (ffct_sent = 1) und ein eventuell laufender spw_send_char() Task zum Senden eines FCTs oder eines NULLs beendet wurde (fctnull_done = 1). Im Fall von detect_error_II() oder spw_timer_12p8() ist entweder wieder ein Fehler aufgetreten oder der Timer abgelaufen. In beiden Fällen folgt der Wechsel nach ErrorReset. Falls der Fehler ein Character

sequence error war (N-Char oder Time-Code im Zustand Connecting empfangen), gibt es zuvor eine Fehlermeldung.
Im Zustand **Run** besteht der Link zwischen der Testbench und dem IP-Core. Der Driver setzt link_running = 1 und startet die beiden Tasks detect_error_III() und spw_link_com(). Letzterer Task dient zur Kommunikation mit dem IP-Core bei bestehender Verbindung. Der detect_error_III() Task dient zur Erkennung eines Fehler der Kategorie III. Im Falle eines Fehlers wird der Task beendet und in den Zustand ErrorReset gewechselt. Zuvor finden jedoch an dieser Stelle noch einige Aktionen statt, die sich auf das vorzeitige Beenden von Sequence Items im Zusammenhang mit dem Link Error Recovery Schema bzw. dem Erzeugen von Fehlern beziehen.
Für den Fall, dass mit dem aktuellen Request Item ein Fehler generiert wurde (Disconnect error, Credit error, Parity error oder Escape error) wird das Handshake-Verfahren für das Item per item_done() abgeschlossen. Dies ist durch ein gesetztes err_gen Flag erkennbar. Trat ein Fehler der Kategorie III auf oder wurde der Link explizit per linkdis Signal beendet, ist zu prüfen, ob aktuell ein Request Item vorhanden ist, dieses bereits in Bearbeitung war und ob es Teil eines zu sendenden Pakets ist. Trifft all dies zu, hängen weitere Aktionen vom zuletzt mit diesem Request Item gesendeten Token ab. Handelte es sich bei dem Token um ein EOP oder EEP, ist zuletzt ein komplettes Paket oder ein fehlerhaftes gesendet worden. Das Request Item wird dann einfach per item_done() abgeschlossen. War es weder ein EOP noch EEP, diente das aktuelle Item zum Senden eines Datenbytes innerhalb eines Pakets. Die verbleibenden Request Items, die den Rest des Pakets repräsentieren, sind dann zu verwerfen (bis zum Request Item mit dem das EOP oder EEP gesendet werden sollte).

3.6.1.2 Receiver: Decoder

Der Receiver ist im Task spw_decode_char() untergebracht und wird innerhalb des spw_ds_decode() Tasks gestartet, sobald nach einem Link-Reset rx_en =1 ist (Zustand ErrorWait). Die Aufgabe des spw_decode_char() Tasks ist die Decodierung der vom IP-Core über Data und Strobe gesendeten Token nach dem Verfahren des synchronen Oversamplings, das Setzen entsprechender Flags, die den Empfang des jeweiligen Tokens anzeigen und die Erkennung von Übertragungsfehlern (Parity error, Escape error und Credit error). Ein Disconnect error wird in einem separaten Task mit dem Namen spw_detect_dc(), parallel zum spw_decode_char() Task laufend, erkannt.
Der Receiver-Task besitzt eine SystemVerilog Queue (in_reg) zum Speichern von genau zwei Data-Strobe Sample-Paaren. Eine zweite Queue (actual_char_rx) dient zur Aufnahme empfangener Bits zur Erkennung der NULL detection sequence während der Link-Initialisierung und zum Speichern von Bits bei der anschließenden Decodierung von Tokens. Auf oberster Ebene des spw_decode_char() Tasks befindet sich eine forever-Schleife, innerhalb der zu jeder positiven Taktflanke des Data-Strobe Sampling-Takts ein neues Data-Strobe Sample-Paar (aktuelle Werte von Data und Strobe) in die Queue in_reg geschoben wird. Der Inhalt von in_reg ist dadurch stets der folgende:

in_reg = [Data_old, Strobe_old, Data_new, Strobe_new]

Der Wert der bitweisen XOR-Verknüpfung der beiden Sample-Paare (alt und neu) entscheidet dann, ob mit dem neuen Sample-Paar ein neues bzw. gültiges Bit (in_reg[2]) empfangen wurde. Warum

das so ist, soll im nächsten Abschnitt erklärt werden. Ist in diesem Takt kein neues Bit vorhanden, wird der eben beschriebene Vorgang im nächsten Takt wiederholt, ansonsten wird es im aktuellen Takt verarbeitet. Zur Erkennung der NULL detection sequence im Rahmen der Link-Initialisierung werden anfangs so lange neue Bits in die Queue actual_char_rx geschoben, bis sich darin die NULL detection sequence abbildet. In diesem Fall ist got_null_i = 1 zu setzen. Die darauf folgende Decodierung der Tokens erfolgt prinzipiell nach dem in Abschnitt 3.1.1.2 beschriebenen Schema. Die einzelnen Schritte die dazu nötig sind, werden mit Hilfe einer Case-Anweisung durchlaufen. Eine Enumeration bildet diese Schritte zur Abfrage innerhalb der Case-Anweisung ab. Die folgende Tabelle gibt Auskunft über die für die Case-Anweisung definierten Schritte.

Tabelle 13: Schritte zur Token-Decodierung innerhalb der Case-Anweisung im Receiver

Schritt (next_step)	Beschreibung
read_par	Neues Bit ist das Paritätsbit eines Tokens
read_ctrl	Neues Bit ist das data-control Flag eines Tokens
read_cc_null	Neues Bit ist Bestandteil eines control characters oder NULLs
read_tc_dchar	Neues Bit ist Bestandteil eines Time-Codes oder data characters

Das Flag new_token entscheidet, ob ein neues Bit im aktuellen Takt das erste Bit des nächsten Tokens ist. In diesem Fall ist das neue Bit das Paritätsbit des Tokens. Dementsprechend ist als nächster Schritt (next_step) read_par zu setzen, wodurch in der Case-Anweisung durch die Abfrage von next_step in den Zweig read_par gesprungen wird. Beim ersten Durchlauf der Case-Anweisung ist next_step = read_ctrl, da das zuletzt empfangene Bit der NULL detection sequence bereits das Paritätsbit des nächsten Tokens war. Bei jedem neuen empfangenen Bit springt die Case-Anweisung dann per Abfrage von next_step in den aktuellen Zweig, um das Bit in die Queue actual_char_rx zu schieben.

Am Ende des Schritts read_par wird stets next_step = read_ctrl gesetzt, da das nächste gültige Bit das data-control flag ist. Im read_ctrl-Zweig erfolgt die Prüfung auf Paritätsfehler, da bereits das Paritätsbit sowie das data-control flag des neuen Tokens decodiert wurde. Im dynamischen Array last_char_rx liegt zu diesem Zeitpunkt das zuletzt decodierte Token, dessen Gültigkeit jetzt geprüft werden kann. Falls ein Paritätsfehler auftrat, ist es ungültig. Ist es gültig, wird in diesem Zweig dem Decodierschema nach als nächstes bestimmt, um welches Token es sich handelt und das entsprechende Flag zur Anzeige des Empfangs gesetzt (z.B. got_fct = 1). Durch den Empfang eines ESC, EOP, EEP oder FCT erfolgt in diesem Zweig auch die Überprüfung, ob ein Escape error bzw. Credit error vorliegt. Am Ende des read_ctrl-Zweigs entscheidet das data-control flag des aktuell zu decodierenden Tokens, ob die nächsten gültigen Bits entweder Bestandteil eines control characters/NULLs oder eines Time-Codes/data characters sind. Dementsprechend wird next_step = read_cc_null oder next_step = read_tc_dchar gesetzt. In den folgenden Durchläufen der Case-Anweisung werden dann die neuen Bits im jeweiligen Zweig in die Queue actual_char_rx geschoben.

Im Zweig read_cc_null erfolgt dies zwei Mal, da dann in actual_char_rx das data-control flag sowie die beiden control bits des control characters (ESC, EOP, EEP oder FCT) liegen. Das Paritätsbit wird zuvor immer in einer eigenen Variable (par_rx) gespeichert. Mit den control bits in actual_char_rx

wird jetzt per Case-Anweisung entschieden, um welches der vier Tokens es sich handelt. Falls das zuletzt empfangene Token ein ESC war, ist das Flag pre_esc zu setzen. Dadurch kann später entschieden werden, ob das aktuell empfangene Token ein NULL oder ein FCT ist. In Abhängigkeit der control bits erfolgt danach die Neuzuweisung des entsprechenden Tokens in last_char_rx für die spätere Überprüfung der Gültigkeit. Am Ende des read_cc_null-Zweigs ist new_token = 1 zu setzen, da das nächste gültige Bit das Paritätsbit eines neuen Tokens ist.
Der read_tc_dchar-Zweig wird innerhalb der Case-Anweisung genau acht Mal durchlaufen, da dann in der Queue actual_char_rx genau neun Bits liegen. Diese sind das data-control Flag sowie die acht Bits des Time-Codes bzw. data characters. Im achten Durchlauf ist vor der Zuweisung last_char_rx = actual_char_rx (Update) wieder pre_esc zu setzen, falls last_char_rx ein ESC repräsentiert. Dadurch lässt sich später wieder entschieden, ob das aktuell empfangene Token ein Time-Code oder ein data character ist. Am Ende des read_tc_dchar-Zweigs ist wieder new_token = 1 zu setzen.

3.6.1.3 Synchrones Oversampling

Das Prinzip des synchronen Oversamplings wurde bereits zuvor erläutert. Dieser Abschnitt dient zur Erklärung, warum die XOR-Verknüpfung von zwei nacheinander genommenen Data-Strobe Sampling-Paaren, gespeichert in der Queue in_reg im Receiver, den Empfang eines neuen bzw. gültigen Bits anzeigt. Dazu muss man das Data-Strobe Codierungsschema beachten. Für die Vorgänger-Nachfolger Beziehung empfangener Bits gibt es vier Möglichkeiten. Erstens: Das zuletzt empfangene Bit war 0, das neue Bit ist ebenfalls 0. Zweitens: Das zuletzt empfangene Bit war 0, das neue Bit ist aber 1. Drittens: Das zuletzt empfangene Bit war 1, das neue Bit ist aber 0. Und viertens: Das zuletzt empfangene Bit war 1, das neue Bit ist ebenfalls 1. Es existieren also vier mögliche Signalübergange auf der Data-Leitung:

I. $0 \rightarrow 0$

II. $0 \rightarrow 1$

III. $1 \rightarrow 0$

IV. $1 \rightarrow 1$

Da sich das Strobe-Signal immer genau dann ändert, wenn sich Data von einem zum nächsten Bit (Takt) nicht ändert, sind entsprechend folgende Übergänge auf der Strobe-Leitung zu erwarten:

I. $1 \rightarrow 0$ oder $0 \rightarrow 1$

II. $1 \rightarrow 1$ oder $0 \rightarrow 0$

III. $1 \rightarrow 1$ oder $0 \rightarrow 0$

IV. $1 \rightarrow 0$ oder $0 \rightarrow 1$

Für jeden möglichen Signalübergang von Data existieren zwei mögliche Signalübergänge von Strobe, die von vorherigen Übergängen abhängig sind. Diese acht möglichen Kombinationen mit zwei Data-Strobe Paaren stellen die gültigen Signalübergänge für den Empfang eines neuen

Bits dar. Die verbleibenden acht Kombinationen von zwei Data-Strobe Paaren stellen ungültige Signalübergänge dar. In diesem Fall ist kein neues bzw. gültiges Bit vorhanden. Die gleichzeitige Änderung von Data und Strobe von einem zum nächsten Takt verletzt die Codierungsvorschrift. Zu diesen Signalübergängen gehören die folgenden vier Kombinationen:

I. Data: 0 → 1 Strobe: 0 → 1

II. Data: 0 → 1 Strobe: 1 → 0

III. Data: 1 → 0 Strobe: 0 → 1

IV. Data: 1 → 0 Strobe: 1 → 0

Die restlichen vier Kombinationen von zwei Data-Strobe Paaren bedeuten keine Aktivität auf der Data- und Strobe-Leitung:

I. Data: 0 → 0 Strobe: 0 → 0

II. Data: 1 → 1 Strobe: 0 → 0

III. Data: 0 → 0 Strobe: 1 → 1

IV. Data: 1 → 1 Strobe: 1 → 1

Die folgende Abbildung zeigt alle möglichen Kombinationen der zwei Data-Strobe Paare.

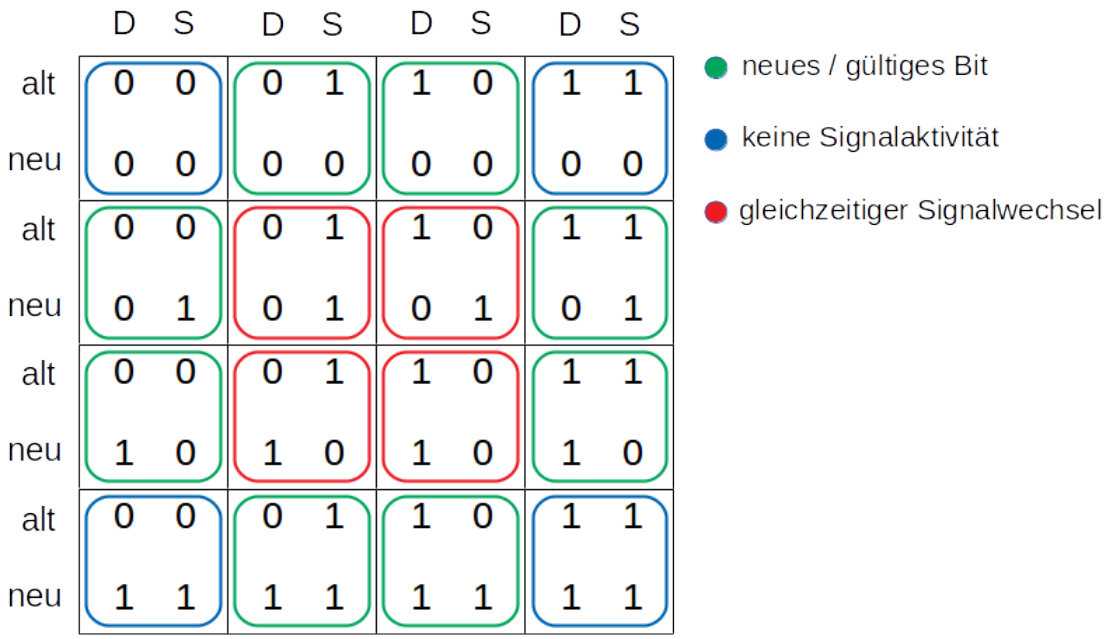

Abbildung 23: Synchrones Oversampling - Kombinationen von zwei Data-Strobe Paaren

Es ist zu erkennen, dass nur diejenigen Kombinationen von zwei Data-Strobe Sample-Paaren eine ungerade Anzahl an Einsen besitzen, die gültige Signalübergänge darstellen und somit ein neues bzw. gültiges Bit repräsentieren. Folglich ergibt die bitweise XOR-Verknüpfung der beiden Sample-Paare genau dann eine logische 1, wenn es sich um ein neues Bit handelt. In diesem Fall entspricht Data im neuen Sample-Paar (untere Reihe der grünen Kästen) dem neuen Bit. Dieses befindet sich innerhalb der Queue in_reg im Receiver stets an der Position in_reg[2].

3.6.1.4 Transmitter: Encoder

Der Transmitter (Encoder) ist im Task spw_link_com() untergebracht und wird gestartet, sobald sich die Link-FSM im Zustand Run befindet. Der spw_link_com()-Task dient zur Kommunikation mit dem IP-Core bei bestehender Verbindung. Er steuert das Senden von N-Chars (Paketen) und Time-Codes auf der Grundlage von Informationen in den Request Items und ist außerdem für die Flusssteuerung (Aktualisierung des Tx credit counts beim Senden von N-Chars und FCTs) sowie die Aufrechterhaltung des Links durch das Senden von NULLs verantwortlich. Zusätzlich können über die Request Items hier Disconnect errors, Parity errors, Credit errors und Escape errors generiert werden. Zur Codierung der einzelnen Tokens in Data- und Strobe-Signale wird innerhalb einer forever-Schleife der spw_send_char() Task aufgerufen, der als Übergabeparameter das jeweils zu sendende Token hat. Die im SpaceWire Standard definierte Priorität der Tokens wird beim Senden stets eingehalten. Die höchste Priorität haben die Time-Codes, gefolgt von FCTs, data characters, EOPs bzw. EEPs und NULLs mit der niedrigsten Priorität.
Mit dem Start des spw_link_com()-Tasks werden einige Flags definiert. Dazu gehören zunächst fünf Flags die anzeigen, ob im aktuellen Schleifendurchlauf entweder ein Time-Code, ein FCT, ein data character, ein EOP oder ein EEP gesendet wurde. Ist am Ende des Durchlaufs keines dieser Flags gesetzt, wird ein NULL gesendet um den Link aufrecht zu erhalten. Drei weitere Flags zeigen die Anfrage zum Senden eines N-Chars an (dchar_r, eop_r und eep_r). Bei einem Durchlauf kann es vorkommen, dass ein über ein Request Item definiertes N-Char aktuell nicht gesendet werden kann, da der Tx credit count 0 ist. Dies ist dann der Fall, wenn der IP-Core keine FCTs mehr sendet, die den Tx credit count erhöhen würden und alle N-Chars in Abhängigkeit zuvor empfangener FCTs gesendet sind. Das entsprechende Flag wird daraufhin gesetzt und verhindert den Abschluss des Handshake-Verfahrens für das Request Item per item_done(). Im nächsten Schleifendurchlauf ist der Aufruf von try_next_item() daher blockiert. Erst wenn in einem der folgenden Durchläufe das N-Char gesendet werden konnte, führt das Rücksetzen des jeweiligen Flags zum Aufruf von item_done() gefolgt von einem try_next_item() für das nächste Item im nächsten Durchlauf.
Mit try_next_item() kommt bewusst die nicht blockierende Variante der Methode zum Bezug des nächsten Request Items zum Einsatz. Wenn im aktuellen Durchlauf kein Request Item vorhanden ist, dürfte ein get_next_item() nicht den Task blockieren, was das Senden weiterer Tokens verhindert und zu einem ungewollten Disconnect error seitens des IP-Cores führt. Die Überprüfung, ob aktuell ein Request Item vorhanden ist, erfolgt über den Null-Pointer des Items. Steht aktuell kein Item über eine zugehörige Sequence zur Verfügung, d.h. es sind keine Daten zu senden oder ein Fehler zu generieren, dann ist anhand des aktuellen Werts des Rx credit counts zu prüfen, ob ein FCT gesendet werden darf. Falls ja, ist das FCT und ansonsten ein NULL zu senden. Steht im aktuellen Durchlauf ein Item zur Verfügung, wird das item_started Flag gesetzt, um zu signalisieren, dass ein Item in Bearbeitung ist. Im weiteren Verlauf des Tasks erfolgt dann die Abfrage der Variablen bzw. Flags des Items in der Reihenfolge, die auf der Priorität der Tokens basiert. Den Anfang macht das Flag zur Generierung eines Disconnect errors (gen_linkdis). Falls es nicht gesetzt ist, folgt danach die Abfrage zum Senden eines Time-Codes (send_tc) und zur Generierung eines Escape errors (gen_erresc_xxx). xxx steht für esc, eop oder eep, da ein Escape error bekanntlich auf drei verschiedene Arten erzeugt werden kann. Als nächstes wird die Anfrage zur Erzeugung eines Credit errors geprüft. Ein gesetztes gen_errcred Flag bewirkt das Senden von insgesamt acht FCTs, was

unabhängig vom aktuellen Wert des Rx credit counts seitens des IP-Cores zu einem Credit error führt. Danach folgen der Priorität nach die Abfragen zum Senden eines FCTs (rx_cred_cnt), eines data characters (send_dchar), eines EOPs (send_eop) und eines EEPs (send_eep).

3.6.2 DS Monitor

Der Monitor des Data-Strobe Agents ist das Gegenstück zum DS Driver und ist in Bezug auf den Aufbau der beiden Tasks spw_decode_char() und spw_detect_dc() des Drivers diesem sehr ähnlich. Die Aufgaben des DS Monitors sind die Beobachtung der Aktivitäten auf dem Data-Strobe Interface zur Kommunikation zwischen dem IP-Core und der Testbench auf Low-Level Ebene sowie der Aufbau von Request- und Response Items. Die aufgebauten Request Items dienen der Speicherung von Paketen und Time-Codes, die der Driver im spw_link_com()-Task über den Link an den IP-Core sendet. Die Response Items hingegen speichern diejenigen Pakete und Time-Codes, die im Receiver-Task des Drivers decodiert werden. Jedes Request- bzw. Response Item speichert stets ein komplettes gesendetes bzw. empfangenes Paket und alle während der Decodierung dieses Pakets gesendeten bzw. empfangenen Time-Codes. Um Request- und Response Items aufbauen zu können, benötigt der Monitor im Prinzip zwei Decoder, einen zur Decodierung der Data-Strobe Signale in Senderichtung (TB $\Rightarrow$ DUT) und den anderen zur Decodierung der Signale in Empfangsrichtung (TB $\Leftarrow$ DUT). Weiterhin muss der Monitor einen Disconnect error in beide Richtungen erkennen können, der auf die Erkennung eines Receiver-Fehlers (Parity error, Escape error oder Disconnect error) auf einer Seite des Links folgt. Wie bereits bekannt ist, führt die Erkennung dieses Receiver-Fehlers auf einer Seite des Links ca. 0,85 μs später zum Disconnect error auf der anderen Seite.
Aufgrund der oben genannten Gegebenheiten besteht der Monitor im Kern aus vier Tasks. Die zwei Tasks spw_detect_dc_tx() und spw_detect_dc_rx() dienen zur Erkennung eines Disconnect errors in Sende- bzw. Empfangsrichtung. Die Erkennung dieses Fehlers ist maßgeblich dafür, wann und auf welche Weise ein Request- bzw. Response Item abgeschlossen werden muss. Die anderen beiden Tasks spw_monitor_req() und spw_monitor_rsp() enthalten jeweils eine modifizierte Version des Inhalts des spw_decode_char()-Tasks (also des Receivers) im Driver, dessen Funktion in Abschnitt 3.6.1.2 erläutert wurde. Die beiden Tasks repräsentieren also die Decoder, einen in Senderichtung (spw_monitor_req()) zum Aufbau von Request Items und einen in Empfangsrichtung (spw_monitor_rsp()) zum Aufbau von Response Items. Zur Anzeige der Erkennung eines Paritätsfehlers über den Monitor sind außerdem zwei Funktionen zur Berechnung des Paritätsbits vorhanden: spw_pargen_tx() und spw_pargen_rx(). Sowohl der Request- als auch der Response-Task durchläuft die Phase der Link-Initialisierung durch die Erkennung der NULL detection sequence, denn nur so existiert ein fester Einstiegspunkt für die weitere Decodierung der Tokens und damit den Aufbau der Sequence Items. Auch der Data-Strobe Monitor besitzt, so wie der FIFO- und der Time-Code Monitor, zwei UVM Analyse-Ports („ap_i“ und „ap_o“) zum separaten Senden der Sequence Items unterschiedlichen Typs (req_item bzw. rsp_item) in Richtung Scoreboard über die notify_transaction()-Funktion.
Im Folgenden soll die Funktion des DS Monitors nur anhand eines der beiden Tasks beschrieben werden, da der andere Task prinzipiell dieselben Aktionen nur auf dem anderen Typ Sequence Item ausführt. Die grundsätzlichen Unterschiede zwischen dem spw_decode_char()-Task im Driver

und den beiden modifizierten Versionen im Monitor sind folgende: Die Monitor-Tasks enthalten ein eigenes got_null_i-Flag zur Anzeige der abgeschlossenen Link-Initialisierung. Dadurch wird bei der weiteren Token-Decodierung der Inititalisierungspart innerhalb der forever-Schleife übersprungen. Da die beiden Monitor-Tasks während der gesamten Simulationszeit laufen und nicht wie der spw_decode_char()-Task nach einem Fehler über die Link-FSM neu gestartet wird, enthalten beide Tasks zu Beginn einen separaten Zweig, der im Falle eines Disconnect errors zu durchlaufen ist. Dieser Zweig setzt einige lokale Flags im Task zurück und schließt ein Request- bzw. Response Item mit einem gesetzten EEP-Flag (eep_req = 1 bzw. eep_rsp = 1) im entsprechenden Item ab, falls dies nicht bereits an anderer Stelle während der Decodierung geschah (siehe rx_valid Flag im Quellcode). Diese Verriegelung soll sicherstellen, dass kein Sequence Item verloren geht bzw. das Handshake-Verfahren für Sequence Items nicht verletzt wird. Ein weiterer Unterschied besteht darin, dass die Flags zur Anzeige des Empfangs eines Tokens im spw_decode_char()-Task in den Monitor-Tasks fehlen, da sie nicht benötigt werden. Außerdem existiert weder ein Rx- noch Tx credit count zur Flusssteuerung. Diese ist ausschließlich Bestandteil des Drivers, der aktiv mit dem IP-Core kommuniziert. Abgesehen von diesen Unterschieden funktioniert die Decodierung der Tokens in den Monitor-Tasks genauso wie im Receiver-Task des Drivers.
Im spw_monitor_req()-Task wird nach der Erkennung der NULL detection sequence im ersten Durchlauf oder nach der Erkennung eines Disconnect errors ein Request Item erstellt. Sobald während der weiteren Decodierung der gesendeten Daten ein gültiges data character decodiert wird, wird es dem dynamischen Array dchar_req im Item hinzugefügt. Das gleiche gilt für Time-Codes, die dem dynamischen Array tcode_req im Item hinzugefügt werden. Im Normalfall geschieht dies so lange, bis ein gesendetes EOP oder EEP das Ende des Pakets markiert. Im Falle eines EOPs wird im Request Item das Flag eop_req und im Falle eines EEPs das eep_req Flag gesetzt und daraufhin das Item per notify_transaction() abgeschickt. Im Request Item sind also ein vollständiges Paket und alle bis zu diesem Zeitpunkt gesendeten Time-Codes vorhanden. Per create()-Methode ist dann vor der weiteren Decodierung ein neues Request Item anzulegen. Der oben beschriebene Vorgang wiederholt sich so lange, bis ein Fehler während des Sendens (Fehler wird generiert) auftritt. Dabei bewirkt die Erkennung eines Parity errors oder Escape errors an entsprechender Stelle das vorzeitige Beenden des Request Items mit gesetztem eep_req-Flag. In beiden Fällen ist daraufhin kein neues Request Item zu erstellen. Dies geschieht erst wieder nach der nächsten Link (Re)-Initialisierung. Die Erkennung eines Disconnect errors bewirkt den Durchlauf des oben erwähnten Zweigs und somit den vorzeitigen Abschluss des Request Items mit gesetztem eep_req-Flag.
Wie bereits erwähnt, führt der spw_monitor_rsp()-Tasks die gleichen Aktionen aus. Der Unterschied ist die Verwendung von Response- anstatt Request Items zum Speichern der durch den IP-Core zur Testbench gesendeten Pakete und Time-Codes.

3.6.3 DS Sequence Item

Das Data-Strobe Sequence Item dient aus der Sicht des DS Drivers zum Senden einzelner N-Chars (data characters, EOPs und EEPs) und Time-Codes in Form von Request Items. Dazu enthält das Sequence Item vier standardmäßig randomisierbare Flags für die Anfrage zum Senden eines dieser Tokens über den Driver. Diese lauten send_tc, send_dchar, send_eop und send_eep und sind

selbsterklärend. EOP und EEP sind, so wie das FCT und das ESC, im Driver hartcodiert. Der Wert eines zu sendenden Datenbytes (data characters) bzw. Time-Codes wird über die randomisierbaren Variablen dchar_tx bzw. tcode_tx bestimmt. Dabei handelt es sich um zwei 9 Bit breite unpacked arrays. Das neunte Bit entspricht dem data-control flag der beiden Tokens und ist stets 0. Die zwei control bits eines Time-Codes sind ebenfalls 0. Dies wird durch zwei constraints im Sequence Item (c_dchar_tx und c_tcode_tx) bei der Randomisierung berücksichtigt. Das Item enthält außerdem Flags zur Erzeugung eines Disconnect errors (gen_linkdis), Parity errors (gen_errpar), Credit errors (gen_errcred) und eines Escape errors (gen_erresc_esc, gen_erresc_eop und gen_erresc_eep). Ein Escape error kann bekanntlich auf drei verschiedene Arten generiert werden, daher die drei Flags. Ein weiteres Flag (packet) dient zur Anzeige, ob ein aktuell verarbeitetes Request Item Bestandteil eines zu sendenden Pakets ist. Sollte mit dem aktuellen Request Item ein Fehler generiert worden sein, wird dem Driver über das Flag mitgeteilt, dass weitere Request Items, die sonst Bestandteil eines zu sendenden Pakets wären, hinsichtlich des Error Recovery Schemas nicht zu verwerfen sind. Die letzte randomisierbare Variable für Request Items lautet tx_divcnt. Damit lässt sich der Taktteilerwert zur Einstellung der Tx-Bitrate definieren. Diese berechnet sich wie folgt:

$$\text{Tx-Bitrate} = \text{DS\_CLK\_FREQ} / (\text{tx\_divcnt} + 1)$$

DS_CLK_FREQ entspricht der Samplingfrequenz für das synchrone Oversampling (aktuell 800 MHz). Für den Wert tx_divcnt = 39 ergibt sich also eine Tx-Bitrate von 20 MBit/s.
Aus der Sicht des DS Monitors dient das Sequence Item zum Speichern von gesendeten (TB $\Rightarrow$ DUT) bzw. empfangenen (TB $\Leftarrow$ DUT) Paketen mit variabler Größe und Time-Codes in Form von Request- bzw. Response Items. Ein Item eines der beiden Typen enthält immer ein komplettes gesendetes bzw. empfangenes Paket und alle währenddessen gesendeten bzw. empfangenen Time-Codes. Ein Request Item verwendet zwei dynamische Arrays (dchar_req und tcode_req) als Container für gesendete Datenbytes eines Pakets bzw. gesendete Time-Codes. Die beiden Flags eop_req und eep_req zeigen an, ob ein gesendetes Paket fehlerfrei und damit vollständig oder aufgrund eines Fehlers unvollständig übertragen wurde. Das eep_req-Flag markiert also ein frühzeitig beendetes Paket. Auf der Seite des Receivers verwendet ein Response Item ebenfalls zwei dynamische Arrays (dchar_rsp und tcode_rsp) als Container für empfangene Datenbytes eines Pakets bzw. empfangene Time-Codes. Die beiden Flags eop_rsp und eep_rsp zeigen an, ob ein empfangenes Paket fehlerfrei und somit vollständig oder aufgrund eines Link-Fehlers unvollständig empfangen wurde.
Das aktuelle Design des DS Drivers sieht vor, dass in einem Request Item, welches der Driver verarbeitet, nur eines der Flags zur Erzeugung eines Fehlers gesetzt wird. Dieselbe Vorgabe gilt generell auch für die Flags für zu sendende N-Chars oder Time-Codes. Das heißt nur ein erzeugter Fehler, ein zu sendendes N-Char und ein zu sendender Time-Code pro Request Item. Man könnte daher behaupten, es sei sinnlos, diese Variablen zu randomisieren, denn sobald ein Flag randomisiert wird, dürfen die anderen Flags nicht gesetzt werden. Nach UVM sind jedoch alle Variablen eines Request Items standardmäßig randomisierbar.
Die folgende Tabelle zeigt eine Übersicht über die Felder des Data-Strobe Sequence Items.

Tabelle 14: Variablen des Data-Strobe Sequence Items

Variable	Datentyp	Randomisierung	Richtung (Testbench)
send_tc	Bit	rand	Out
send_dchar	Bit	rand	Out
send_eop	Bit	rand	Out
send_eep	Bit	rand	Out
gen_linkdis	Bit	rand	Out
gen_errpar	Bit	rand	Out
gen_errcred	Bit	rand	Out
gen_erresc_esc	Bit	rand	Out
gen_erresc_eop	Bit	rand	Out
gen_erresc_eep	Bit	rand	Out
tcode_tx [8:0]	Unpacked Bit-Array	rand	Out
dchar_tx [8:0]	Unpacked Bit-Array	rand	Out
packet	Bit	rand	Out
tx_divcnt	Integer	rand	Out
tcode_req []	Dyn. Byte-Array	non-rand	Out
tcode_rsp []	Dyn. Byte-Array	non-rand	In
dchar_req []	Dyn. Byte-Array	non-rand	Out
dchar_rsp []	Dyn. Byte-Array	non-rand	In
eop_req	Bit	non-rand	Out
eep_req	Bit	non-rand	Out
eop_rsp	Bit	non-rand	In
eep_rsp	Bit	non-rand	In

Bei den nicht randomisierbaren Variablen, die mit der Richtung „Out“ gekennzeichnet sind, handelt es sich um Variablen, die in einem Request Item nur vom Monitor verwendet werden.

3.7 Sequences

Für die UVM-Testbench wurden diverse Sequences für die in Abschnitt 3.8 beschriebenen Funktionstests hinsichtlich der Verifikation des IP-Cores erstellt. Diese sollen im Folgenden erläutert werden. Eine Sequence generiert jeweils nur eines der drei Sequence Items (FIFO Sequence Item, TC Sequence Item oder DS Sequence Item) in Form von Request Items zur Erzeugung von Stimuli und wird nur einem Sequencer zugeordnet. Daher sind die Sequences in drei Kategorien eingeteilt: FIFO Sequences, Time-Code Sequences und Data-Strobe Sequences. Die Data-Strobe Sequences beginnen im Namen stets mit einem voran gestellten „spw_ds“, die FIFO Sequences mit „spw_dut“ oder „spw_fifo“ und die TC Sequence (Einzahl) mit „spw_tc“.

3.7.1 FIFO Sequences

Die FIFO Sequences, von denen es aktuell drei gibt, dienen zur Erzeugung von Paketen mit variabler Anzahl und Länge in Form von FIFO Sequence Items, sowie zum Starten und Beenden eines Links seitens des IP-Cores über das FIFO-Interface.

Bei der **FIFO payload sequence** (spw_fifo_pl_seq) handelt es sich um eine Sequence zur Generierung von Paketen, die in das Tx-FIFO des IP-Cores geschrieben werden sollen. Um die Anzahl bzw. die Länge der Pakete variieren zu können, besitzt die Sequence zwei randomisierbare Integer-Variablen (packet_cnt bzw. packet_len). Da über ein Request Item und den FIFO Driver die Signale autostart, linkstart, linkdis und txdivcnt des FIFO-Interfaces zu definieren sind, bietet die Sequence mit einer weiteren Variable (ls_delay) die Möglichkeit, die Link-Initialisierung seitens des IP-Cores zu verzögern. Der Wert von ls_delay entspricht einer Verzögerung in Millisekunden. Der Link wird hier standardmäßig über die Autostart-Funktion aufgebaut. Die Sequence enthält außerdem noch benutzerdefinierbare constraints für die drei Variablen.
Bei der **DUT link start sequence** (spw_dut_ls_seq) handelt es sich um eine sehr einfache FIFO Sequence, die lediglich ein einzelnes Sequence Item generiert. Mit der Sequence lässt sich der Zeitpunkt der Link-Initialisierung per Autostart-Funktion seitens des IP-Cores festlegen. Sie ist unabhängig von der FIFO payload sequence einsetzbar und enthält ebenfalls die Variable ls_delay zur Verzögerung der Link-Initialisierung (Wert in Millisekunden).
Die **DUT link disable sequence** (spw_dut_ldis_seq) ist eine einfache Sequence zur Erzeugung eines einzelnen Sequence Items, mit dem das linkdis-Signal zum expliziten Beenden einer bestehenden Verbindung zwischen der Testbench und dem IP-Core über das FIFO-Interface gesetzt wird. Die Sequence besitzt aktuell keine weiteren Möglichkeiten für das Timing zum Beenden des Links, außer die Sequence selbst an geeigneter Stelle innerhalb eines Tests auszuführen. Die Ausführung der Sequence führt kurze Zeit später zur Erkennung eines Disconnect errors seitens der Testbench, die danach versucht den Link wieder neu aufzubauen.

3.7.2 Time-Code Sequences

Die **Time-Code sequence** (spw_tc_seq) wird zur Generierung von Time-Codes mit variabler Anzahl und variablem zeitlichen Abstand zwischen den Time-Codes verwendet. Sie gelangen dann über das Time-Code Interface per tick_in-Signal in den IP-Core, der diese jeweils nach dem nächsten Token sendet. Die Variable tc_cnt der Sequence bestimmt die Anzahl zu generierender Time-Codes bzw. Sequence Items. Mit tc_delay lässt sich der Zeitabstand zwischen zwei Anfragen zum Senden eines Time-Codes in Millisekunden einstellen. Für die beiden Variablen enthält die Sequence benutzerdefinierbare constraints. Der Wert eines zu sendenden Time-Codes wird automatisch inkrementiert. Die beiden control flags eines Time-Codes sind entsprechend der Definition im Standard stets 0. Der Wert des ersten gesendeten Time-Codes ist 0b000001. Sobald ein erzeugter Time-Code den Wert 63 hat (0b111111), wird als nächstes wieder ein Time-Code mit dem Wert 0b000001 gesendet. Sollen mit der Sequence zufällige Time-Code Werte generiert werden, sind einige wenige Änderungen im Code vorzunehmen.

3.7.3 Data-Strobe Sequences

Die Data-Strobe Sequences werden zur Erzeugung von vier verschiedenen Fehlern (Disconnect error, Parity error, Escape error und Credit error) und zur Generierung von N-Chars (Paketen) und Time-Codes zum Senden auf Low-Level Ebene über das Data-Strobe Interface verwendet. Das Ziel bei der Entwicklung dieser acht Sequences war eine möglichst vielseitige Einsetzbarkeit. Aus

diesem Grund existieren zwei Sequences, mit denen sich jeweils ausschließlich Time-Codes bzw. Pakete senden lassen. Eine weitere Sequence stellt eine Kombination der beiden zuletzt genannten Sequences dar. Eine Sequence zur Erzeugung eines Character sequence errors existiert in der aktuellen Version der Testbench noch nicht, da hierzu wahrscheinlich größere Änderungen im Code des DS Drivers bezüglich der Link-FSM nötig gewesen wären. Der Character sequence error tritt nur während der Link-Initialisierung auf, es müsste daher schon vor einer bestehenden Verbindung möglich sein, Informationen aus Sequence Items abzurufen.

Die **DS NULL sequence** (spw_ds_null_seq) ist eine einfache Sequence und dient, wie der Name bereits vermuten lässt, zum Erzeugen von NULLs. Bei einem aktiven Link werden NULLs immer dann gesendet, wenn aktuell keine N-Chars, Time-Codes oder FCT zu senden sind. Aus diesem Grund sind bei der Erzeugung der Request Items die Flags zum Senden von Tokens stets zurück gesetzt. In diesem Fall wird über den spw_link_com()-Task im Driver automatisch ein NULL gesendet (wenn aktuell kein FCT gesendet werden kann). Die Sequence besitzt zwei randomisierbare Variablen. null_cnt legt die Anzahl zu sendender NULLs fest. Die tatsächlich gesendete Anzahl kann leicht von diesem Wert abweichen, falls zwischenzeitlich das eine oder andere FCT anstatt eines NULLs gesendet wird. Über die zweite Variable txdivcnt lässt sich, wie zuvor beschrieben, der Taktteilerwert zur Festlegung der Tx-Bitrate einstellen (falls benötigt). Weiterhin lässt sich per constraint die Anzahl zu sendender NULLs einschränken.

Bei der **DS payload sequence** (spw_ds_pl_seq) handelt es sich um eine Sequence ausschließlich zur Generierung von Paketen mit variabler Anzahl und Länge. Diese ist der FIFO payload sequence sehr ähnlich und besitzt ebenfalls zwei Integer-Variablen (packet_cnt und packet_len) zur Bestimmung von Paketanzahl bzw. Paketlänge. Auch txdivcnt zur Bestimmung der Tx-Bitrate ist hier vorhanden. Zwei einstellbare constraints beschränken die Werte von packet_cnt bzw. packet_len. Bei der Erzeugung der Request Items über zwei verschachtelte Schleifen wird zunächst in der inneren Schleife über packet_len iteriert. Die darin generierten Items setzen stets das send_dchar-Flag. Nach packet_len Durchläufen ist das Flag zum Senden des EOPs (send_eop) im nächsten Sequence Item zu setzen. Danach werden über die äußere Schleife die verbleibenden packet_cnt - 1 Pakete generiert. Das packet-Flag ist für alle Request Items gesetzt.

Die **DS time-code sequence** (spw_ds_tc_seq) dient ausschließlich zur Erzeugung von Time-Codes mit variabler Anzahl (tc_cnt) und variablem Zeitabstand (tc_delay Millisekunden) zwischen den gesendeten Time-Codes. Die Sequence ist der Time-Code Sequence sehr ähnlich. constraints für die beiden Variablen sind vorhanden. Über txdivcnt lässt sich der aktuelle Taktteilerwert bestimmen. Die DS time-code sequence generiert und startet alle tc_delay Millisekunden ein neues Request Item, in dem das send_tc-Flag gesetzt ist und in dem der Wert von tcode_tx dem aktuellen Wert der lokale Variable tc_val entspricht. tc_val wird vor dem Start jedes Items inkrementiert und wieder auf 0 gesetzt, sobald der Maximalwert (63) erreicht ist.

Die **DS payload/time-code sequence** (spw_ds_pltc_seq) stellt eine Kombination der beiden Sequences DS payload sequence und DS time-code sequence dar und generiert sowohl Pakete als auch Time-Codes. Die Variablen der Sequence lauten packet_cnt, packet_len, tc_delay und txdivcnt. Deren Funktion wurde bereits erläutert. Die Variable tc_val hat die gleiche Funktion wie in der DS time-code sequence und definiert den aktuellen Wert eines Time-Codes, der mit jedem gesendeten Time-Code inkrementiert wird. Zur Erzeugung der Request Items werden zwei verschachtelte

Schleifen, so wie in der DS payload sequence, durchlaufen. Am Anfang der inneren Schleife erfolgt jedoch alle tc_delay Millisekunden zusätzlich der Start eines Request Items mit gesetztem send_tc-Flag und dem aktuellen Wert von tc_val in tcode_tx.
Mit der **DS parity error sequence** (spw_ds_errpar_seq) wird ein einzelnes Sequence Item zur Erzeugung eines Paritätsfehlers generiert. Die einzige Variable der Sequence ist txdivcnt. Das generierte Request Item besitzt ein gesetztes gen_errpar-Flag. Alle anderen Flags sind zurück gesetzt. Die Abfrage von gen_errpar führt dann innerhalb des spw_send_char()-Tasks zum Senden eines falschen Paritätsbits.
Mit der **DS escape error sequence** (spw_ds_erresc_seq) wird ein einzelnes Request Item zur Erzeugung eines Escape errors generiert. Die Sequence besitzt vier Variablen (Flags). Das Flag erresc_esc löst einen Escape error durch das Senden eines ESC, gefolgt von einem weiteren ESC, aus. Das Flag erresc_eop löst den Fehler durch das Senden eines ESC, gefolgt von einem EOP, aus. Durch erresc_eep wird nach einem ESC ein EEP gesendet. Das vierte Flag ist wieder txdivcnt. Welches der drei Flags der Sequence im Request Item zum Setzen eines der drei entsprechenden Flags gen_erresc_esc, gen_erresc_eop oder gen_erresc_eep führt, ist auf höherer Ebene im Test zu definieren, der die Sequence startet.
Bei der **DS disconnect error sequence** (spw_ds_errdisc_seq) handelt es sich um eine Sequence zur Erzeugung eines Disconnect errors seitens der Testbench. Die einzige Variable der Sequence ist txdivcnt. Das generierte Request Item besitzt ein gesetztes gen_linkdis-Flag. Alle anderen Flags sind wieder zurück gesetzt. Die Abfrage von gen_linkdis erfolgt im spw_link_com()-Task des DS Drivers. Das gesetzte Flag bewirkt dann wiederum das Setzen des linkdis-Flags, wodurch ein bestehender Link beendet wird (Fehlerkategorie III).
Die letzte Data-Strobe Sequence heißt **DS credit error sequence** (spw_ds_errcred_seq) und ermöglicht die Erzeugung eines Credit errors durch das Senden mehrerer aufeinander folgender FCTs. Die einzige Variable der Sequence ist txdivcnt. Im generierten Request Item ist lediglich das gen_errcred-Flag gesetzt, das im spw_link_com()-Task abgefragt wird und zum Senden von 8 FCTs führt. Dies verursacht selbst dann einen Credit error, wenn der Tx credit count im Transmitter des IP-Cores aktuell 0 ist.
Diese Variante zur Erzeugung eines Credit errors stellt nur eine von zwei Möglichkeiten dar. Die durch die Sequence beschriebene Variante ignoriert den Rx credit count der Testbench, wonach FCTs nur dann zu senden sind, wenn dieser es zulässt. Die zweite Möglichkeit, die Flusssteuerung zu umgehen, besteht darin, den Tx credit count der Testbench zu ignorieren und selbst dann weiter N-Chars zu senden, wenn der Tx credit count bereits 0 ist. Der IP-Core empfängt dann N-Chars, obwohl er keine mehr erwartet, was einen Credit error verursacht (Rx credit count ist 0 während des Empfangs eines N-Chars). Um den Vorgang zu beschleunigen, können zusätzlich seitens der Testbench empfangene FCTs ignoriert werden, wodurch der Tx credit count nicht weiter erhöht wird. Hinsichtlich der vollständigen Verifikation eines IP-Cores ist die zweite Möglichkeit noch zu implementieren.

3.8 Abgeleitete Tests

Zur Überprüfung des korrekten Verhaltens des SpaceWire Light IP-Cores wurden zwei Basistests erstellt. Beim ersten Test handelt es sich um einen normalen Funktionstest, mit dem die korrekte Funktionsweise des IP-Cores im normalen Betrieb geprüft werden soll. Der zweite Test ist ein Fehler-Funktionstest zur Prüfung des Verhaltens des IP-Cores im Fehlerfall. Für weitere Tests lassen sich die im vorherigen Abschnitt beschriebenen Sequences noch auf andere Weise kombinieren. Denkbar wäre z.B. ein dritter Funktionstest, der die beiden hier separat durchzuführenden Tests kombiniert und während des normalen Betriebs zwischenzeitlich per Zufall einen Fehler generiert. Mit dem separaten Fehler-Funktionstest soll jedoch ausschließlich die Fähigkeit zur Erkennung der im SpaceWire Standard definierten Fehler (außer wie bereits erwähnt ein Character sequence error) überprüft werden.

3.8.1 Testauswahl

Um eine Simulation des IP-Cores auf der Grundlage eines der zwei Tests starten zu können, ist der jeweilige Test über den Namen der Test-Klasse in der Datei „spw_vsim_cmd_args.f“ auszuwählen:

```
# Normal operation test
# =====================
#+UVM_TESTNAME="spw_norm_op_test"
# Error operation test
# ====================
#+UVM_TESTNAME="spw_err_op_test"
```

Dabei ist stets nur ein Test per UVM_TESTNAME für die Simulation zu definieren. Die Simulation wird dann per „run_spw_sim.do“ Script gestartet. In welchem Ordner sich die Dateien befinden ist der Datei „tb_filelist.txt“ im Root-Verzeichnis der Testbench zu entnehmen.
Zur Anzeige bestimmter Informationen während der Simulation (UVM Reporting) ist das Verbosity Level per +UVM_VERBOSITY=<Verbosity Level> in der oben genannten .f-Datei entsprechend anzupassen. Durch die Auswahl des Verbosity Levels UVM_NONE wird gar nichts angezeigt. Per UVM_LOW erfolgt die Ausgabe von grundlegenden UVM-Informationen und Informationen bezüglich des Beginns sowie des Endes der Simulation. UVM_MEDIUM gibt zusätzlich Fehlermeldungen des Data-Strobe Agents und weitere UVM-Informationen aus. UVM_HIGH zeigt darüber hinaus den Inhalt aller Request- und Response Items der drei Monitore an. Per UVM_FULL erhält man ergänzende Informationen zu den randomisierten Werten aller Request Items der Sequences. Mit UVM_DEBUG werden schließlich weitere UVM Debug-Informationen ausgegeben.

3.8.2 Normaler Funktionstest

Der normale Funktionstest schafft ein Szenario, in dem der Link zur Kommunikation mit dem IP-Core einmalig aufgebaut und am Ende der Simulation per linkdis-Signal wieder beendet wird. Innerhalb dieses Zeitraums werden Pakete zum Senden in das Tx-FIFO des IP-Cores geschrieben und über den Link empfangen, zu sendende Time-Codes über das Time-Code Interface für den IP-Core bereit gestellt, Pakete und Time-Codes über Sequences generiert und zum IP-Core gesendet (Data-Strobe Interface) und zwischendurch mehrere NULLs eingeschoben.

In der Klasse des normalen Funktionstests sind zu Beginn der Run-Phase benötigte Sequences zu erstellen (z.B. per create_seq-Macro). Dazu gehören die **Data-Strobe Sequences** zum Senden von NULLs, Paketen und Time-Codes, die **FIFO payload sequence**, die **Time-Code sequence** und die **DUT link disable sequence**. Danach folgt die Randomisierung der Sequences bzw. innerhalb der Sequences die Randomisierung der zu generierenden Request Items. Durch die Festlegung der Sequence-Variablen bei der Randomisierung der Sequences wird z.B. die Anzahl und die Länge der Pakete bestimmt. Anschließend werden die Sequences in der gewünschten Reihenfolge über die zugehörigen Sequencer der Agents gestartet.
Der normale Funktionstest beginnt mit der Ausführung der FIFO payload sequence sowie der Time-Code sequence, die im weiteren Verlauf des Tests parallel zu den nachfolgenden Sequences läuft. Die FIFO payload sequence wird zuerst ausgeführt, damit dem IP-Core bereits zu sendende Pakete zur Verfügung stehen, wenn der Link besteht. Als nächstes werden ausschließlich Pakete per DS payload sequence zum IP-Core gesendet. Danach folgt eine kurze Leerlaufphase durch eine DS NULL sequence. Im Folgenden wird die DS payload/time-code sequence gestartet, wodurch Pakete mit eingeschobenen Time-Codes zum IP-Core gelangen. Nach der nächsten Leerlaufphase seitens der Testbench durch das Senden von NULLs und eventuell einiger eingeschobener FCTs, sorgt die DS time-code sequence dafür, dass ausschließlich Time-Codes zum IP-Core gesendet werden. Abschließend bewirkt die Ausführung der DUT link disable sequence nach einer weiteren Leerlaufphase den Abbruch der Verbindung.
Die normalen Funktionstests zeigten, dass der SpaceWire Light IP-Core den Link während der gesamten Simulationszeit bei verschiedenen Einstellungen der Sequences aufrecht erhielt und fehlerfrei mit der Testbench kommunizierte. Dabei stützt sich die Behauptung der fehlerfreien Kommunikation auf stichprobenartige Vergleiche von gesendeten und empfangenen Daten über den Vergleich der Inhalte von Request- und Response Items. Dies ist die einzige Möglichkeit der Verifikation, da die automatisierte Verifikation zu diesem Zeitpunkt noch nicht möglich ist.

3.8.3 Fehler-Funktionstest

Der Fehler-Funktionstest schafft ein Szenario, in dem der Link zur Kommunikation mit dem IP-Core initial nach einer bestimmten Wartezeit automatisch aufgebaut und dann durch generierte Fehler zwischenzeitlich wieder beendet wird. Die erzeugten Fehler sollen danach stets zur Re-Initialisierung des Links führen. Zwischen dem Start zweier Sequences, die jeweils den Fehler bewirken, werden entweder NULLs (Leerlauf) und/oder Pakete zum IP-Core gesendet. Am Ende des Tests wird der Link per Disconnect error abgebrochen. Weitere erzeugte Fehler sind der Parity error, der Credit error und der Escape error mit seinen drei Varianten.
In der Klasse des Fehler-Funktionstests sind zu Beginn der Run-Phase die benötigten Sequences zu erstellen. Dazu gehören die vier **DS xxx error sequences** zur Generierung eines der vier Fehler, die **DS NULL sequence** für die Leerlaufphasen, die **DS payload sequence** zum Senden einiger Pakete und die **DUT link start sequence** zur Verzögerung der Link-Initialisierung. Es folgt wieder die Randomisierung der Sequences bzw. der Request Items. Die Parameter der Sequences sind wieder im Zuge ihrer Randomisierung festzulegen. Dazu gehören beispielsweise die Anzahl bzw. die Länge der Pakete und die Flags zur Bestimmung der Art des Escape errors. Als nächstes werden die Sequences der Reihe nach über die entsprechenden Sequencer der Agents gestartet.

Zu Beginn des Fehler-Funktionstests wird per ls_delay in der DUT link start sequence die Link-Initialisierung verzögert. Nach der Ausführung einer DS payload sequence erfolgt zunächst die Erzeugung eines Credit errors durch die DS credit error sequence (Senden von 8 FCTs). Es ist zu erwarten, dass der IP-Core den Link daraufhin beendet und versucht ihn neu aufzubauen. Danach wird in der oben genannten Reihenfolge fortgefahren, wobei sich eine DS NULL sequence und/oder eine DS payload sequence zwischen zwei Sequences zur Generierung eines weiteren Fehlers befindet, um den Link eine Zeit lang aufrecht zu erhalten. Bei den nun folgenden Fehlern handelt es sich der Reihe nach um einen Parity error, Escape error per ESC, Escape error per EOP, Escape error per EEP und schließlich um einen Disconnect error. Erwartungsgemäß reagiert der IP-Core bei der Erkennung eines dieser Fehler mit dem Abbruch des Links, was seitens der Testbench zum Disconnect error führt.
Die durchgeführten Fehler-Funktionstests haben gezeigt, dass der SpaceWire Light IP-Core so reagierte, wie es zu erwarten war. Die Generierung eines der vier Fehler führte in jedem Fall zur Erkennung dieses Fehlers, was anhand des jeweiligen Signals im FIFO-Interface (Signal errdisc, errpar, erresc oder errcred) überprüft werden konnte. Darauf folgte stets der Verbindungsabbruch seitens des IP-Cores. Die Beobachtung der Signale ist aktuell die einzige Möglichkeit die Fehlererkennung zu verifizieren, da die automatisierte Verifikation zu diesem Zeitpunkt noch nicht möglich ist.

4 Zusammenfassung

Die Aufgabe dieser Abschlussarbeit war die Entwicklung einer UVM-Testbench zur automatisierten Verifikation beliebiger SpaceWire IP-Cores, die nach dem ECSS-E-ST-50-12C Standard (SpaceWire - Links, nodes, routers and networks) entwickelt wurden. Als bereits geprüftes und daher als zuverlässig geltendes DUT kam der SpaceWire Light IP-Core von OpenCores zum Einsatz, um die vorgesehene bzw. korrekte Funktionsweise der Testbench selbst prüfen zu können.
Im Theorieteil der Arbeit wurden zunächst die zur Entwicklung der Testbench benötigten Grundlagen von UVM erläutert. Nach einer kurzen Begriffserklärung und der Beschreibung der sich ergebenden Vorteile durch die Verwendung von UVM bei der Hardware-Verifikation im Gegensatz zur Verifikation mittels klassischer Verilog/VHDL Testbenches, folgte die Vorstellung der beiden grundlegenden Architekturen einer UVM-Testbench. Die in diesem Abschnitt beschriebene Blockstruktur bildete die Grundlage der in der Arbeit entwickelten Block-Level Testbench. Die zum Aufbau dieser Grundstruktur benötigten Basiskomponenten wurden im Folgenden erläutert, vom Top-Level Testbench Modul zur Instanziierung des DUT über den Basistest zum Aufbau der Environment (Env) und benötigter Subkomponenten, bis hin zu den UVM Agents auf der untersten Hierarchieebene. Zum weiteren Verständnis des Ablaufs bei der Verifikation per UVM beschreibt der nächste Abschnitt die UVM-Phasen, die während einer Simulation zu durchlaufen sind, beginnend mit den Build-Phasen zum Aufbau und zur Verbindung der Testbench-Komponenten, gefolgt von den Run-Phasen zur Durchführung der eigentlichen Simulation und den abschließenden Cleanup-Phasen. Im letzten Abschnitt zur Architektur der Testbench wurde die Verwendung der UVM-Factory erläutert, wobei im Sinne der Wiederverwendbarkeit von Komponenten Regeln für die Registrierung von Komponenten, bei der Verwendung des class constructors und beim Erstellen von Komponenten

und Objekten zu beachten sind. Der darauf folgende Abschnitt „TB-DUT Kommunikation“ beschreibt die Art und Weise, wie eine UVM-Testbench mit dem DUT kommuniziert und zeigt diesbezüglich die Verwendung der UVM-Konfigurationsdatenbank (uvm_config_db) zur Kommunikation über virtuelle Interfaces anhand von Code-Beispielen. Als nächstes wird die Konfiguration und der Aufbau der Testumgebung erläutert. Hier geht es im Kern darum, wie Informationen zum Aufbau von Subkomponenten und deren Konfiguration innerhalb der Testbench-Hierarchie mit Hilfe eines verschachtelten Konfigurationsobjekts weiter gereicht werden. Außerdem wird der Vorgang zur Verbindung der einzelnen Komponenten nach deren Aufbau (Build-Phase) erklärt. Der letzte UVM-Grundlagen-Abschnitt befasst sich mit dem Thema Sequences und Sequence Items und gibt einen Überblick über deren Eigenschaften und Funktion. Er beschreibt im Detail, wie Sequences bzw. Sequence Items bei UVM zur Erzeugung von Stimuli einzusetzen sind, welche Methoden (Funktionen) für Sequence Items existieren und wofür sie stehen, wie Sequence Items zwischen einem Driver und einem Sequencer transferiert werden und nach welchen Regeln das Handshake-Protokoll für Sequence Items abläuft.

Im folgenden Abschnitt des Theorieteils geht es um die für die Entwicklung der Testbench relevanten Inhalte des SpaceWire Standards. Zunächst wurde dessen Inhalt durch die Beschreibung der darin enthaltenen Kapitel knapp zusammen gefasst. Diesbezüglich erfolgte auch die Abgrenzung nicht relevanter Inhalte. Dazu gehört z.B. das Kapitel Network level, in dem SpaceWire Netzwerke und das Routing von Paketen definiert sind. Ein weiterer Unterabschnitt erklärt das Data-Strobe Codierungsschema, das bei der Kommunikation über SpaceWire Links per LVDS verwendet wird. Es folgen Erläuterungen zu den im Standard definierten Zeichen (SpaceWire Tokens) und deren Funktion bei der Übertragung von Paketen und Zeitinformationen sowie zur Flusssteuerung (Flow Control), die den Verlust von Daten verhindern sollen. Für alle weiteren relvanten Inhalte des Standards sind jeweils Unterabschnitte vorhanden. Darin werden die zur Steuerung des Link-Interfaces verwendete FSM und ihre Zustände, der Ablauf der Link-Initialisierung und zugrunde liegende Protokolle, die per Standard definierten Fehlerarten und die Fehlererkennung sowie das Link Error Recovery Schema im Fehlerfall beschrieben.

Der dritte und letzte Abschnitt des Theorieteils befasst sich mit dem als Testobjekt eingesetzten SpaceWire Light IP-Core und enthält zunächst einige Informationen zu Bezugsquellen und zur Entwicklung des IP-Cores. Danach wird das Konzept des Transmitters und des Receivers des IP-Cores und deren Implementierung erläutert und gezeigt, nach welchem Prinzip Daten gesendet und empfangen werden (Stichwort Oversampling). Es folgt eine ausführliche Beschreibung der Funktionsweise der Instanzen des IP-Core Interfaces anhand eines Blockdiagramms, eine Übersicht über die Konfigurationsmöglichkeiten des IP-Cores über VDHL Generics sowie eine abschließende detailierte Beschreibung aller Port-Signale (Ein- und Ausgänge) des SpaceWire Light inklusive deren Bedeutung.

Das dritte Kapitel dieser Arbeit beschreibt die einzelnen Phasen der Entwicklung der UVM-Testbench im Detail, vom ursprünglichen Designkonzept über den Entwurf eines Verifikationsplans für SpaceWire IP-Cores, den Aufbau des Grundgerüsts, die Entwicklung der einzelenen Agents und ihrer Subkomponenten, den Sequences und Sequence Items für ausreichende Stimuli-Variationen bis hin zu abschließenden Funktionstests des SpaceWire Light IP-Cores. Im ersten Abschnitt wird das Designkonzept der Testbench präsentiert. Dazu gehören anfängliche Überlegungen, welche

Position die Testbench innerhalb eines SpaceWire Netzwerks einnimmt, welche Aufgaben sie daher zu erfüllen hat und wie der SpaceWire Codec umzusetzen ist. Da die Entscheidung auf einen task-basierten Aufbau fiel, folgen in Unterabschnitten Erklärungen, wie z.B. zentrale Bestandteile des SpaceWire Interfaces wie die Link-FSM, der Transmitter und der Receiver in Form von Tasks implementiert wurden. In Bezug auf die Link-FSM betrifft dies beispielsweise die Parallelisierung gewisser Prozesse (Tasks) zur Bestimmung von Zustandsübergängen. Bezüglich des Receivers wird das Konzept zur Decodierung der SpaceWire Tokens anhand eines Ablaufplans vorgestellt. Weiterhin folgen Erläuterungen, warum und wie das synchrone Oversampling implementiert wurde. Der Unterabschnitt „Transmitter: DS-Encodierung“ beschreibt die Entwicklung des Tasks, der in der Testbench die Funktion des Transmitters besitzt und daher zu sendende Pakete in Form von Sequence Items verarbeitet. Im nächsten Unterabschnitt wird die Implementierung der Link-FSM in Form einer case-Anweisung beschrieben und erläutert, wie die einzelnen Link-Timeouts umgesetzt wurden. Der vorletzte Abschnitt in Bezug auf das Designkonzept befasst sich mit den Signalen der Testbench. Hier geht es um die Festlegung der grundsätzlichen Kommunikation zwischen dem IP-Core und der Testbench über die Interfaces der Agents und wie sich Race conditions durch den Einsatz von Clocking Blocks vermeiden lassen. Anschließend wird darauf eingegangen, auf welche Art und Weise Sequence Items zur Verarbeitung und Speicherung von Daten verwendet werden.
Der zweite Abschnitt erklärt die Inhalte des erstellten Verifikationsplans für SpaceWire IP-Cores, der auf einer tabellarischen Zusammenfassung verifikationsrelavanter Inhalte des SpaceWire Standards basiert. Sowohl der Verifikationsplan als auch die Zusammenfassung befinden sich im Anhang dieser Arbeit.
Der dritte Abschnitt gibt anfangs eine Übersicht über die Grundstruktur der Testbench und ihre Hauptkomponenten und beschreibt, wie diese auf der Basis eines Beispiels einer Block-Level Testbench aus dem UVM-Cookbook aufgebaut wurden. Es folgen Informationen zur Instanziierung des SpaceWire Light IP-Cores und wie sich dieser konfigurieren lässt. Die letzten beiden Unterabschnitte geben Auskunft über die entwickelten Basiskomponenten sowie Möglichkeiten der Konfiguration der Testbench.
Die nächsten drei Abschnitte enthalten detaillierte Erläuterungen zur Entwicklung der drei Agents (FIFO Agent, Time-Code Agent und Data-Strobe Agent). Da die wichtigsten Bestandteile der Agents jeweils der Driver, der Monitor sowie das zugehörige Sequence Item sind, existieren dafür separate Unterabschnitte, in denen ihr Aufbau und ihre Funktion innerhalb der Testbench erklärt werden.
Der Abschnitt „Sequences“ beschreibt die für abschließende Funktionstests des SpaceWire Light IP-Cores entwickelten Sequences. Dazu gehören FIFO Sequences, Time-Code Sequences und Data-Strobe Sequences. Mit den FIFO Sequences lassen sich entweder Pakete in das Tx-FIFO des IP-Cores schreiben, oder ein Link seitens des IP-Cores starten oder beenden. Die Time-Code Sequence erzeugt in frei definierbaren Zeitabständen über den IP-Core zu sendende Time-Codes. Bei den Data-Strobe Sequences handelt es sich um Sequences, mit denen über die Testbench zu sendende Pakete/Time-Codes generiert und Link-Fehler erzeugt werden können.
Der letzte Abschnitt des dritten Kapitels beschreibt die Konstruktion und die Durchführung von Tests, mit denen die korrekte Funktionsweise des SpaceWire Light IP-Cores nachgewiesen wurde. Normale Funktionstests mit verschiedenen Einstellungen der Parameter in den zugrunde liegenden Sequences zeigten die fehlerfreie Kommunikation zwischen dem IP-Core und der Testbench. Die

Fehler-Funktionstests belegten, dass der SpaceWire Light IP-Core in der Lage ist, einen Disconnect error, Parity error, Escape error und Credit error zu erkennen.
Nach etwa vier Monaten der Entwicklungszeit stellte sich heraus, dass der anfangs festgelegte Zeitplan für die Entwicklung der UVM-Testbench nicht eingehalten werden konnte. Dies ist dadurch begründet, dass die Entwicklung des Data-Strobe Agents, der den kompletten SpaceWire Codec im Driver implementiert, wesentlich mehr Zeit in Anspruch nahm, als ursprünglich geplant war. Weiterhin tauchten aufgrund der Komplexität des Drivers in Testläufen bislang unentdeckte Fehler auf, deren Beseitigung die Fertigstellung zusätzlich verzögerte. Aus diesen Gründen fehlte die Zeit für die Entwicklung des UVM-Scoreboards sowie die Beschreibung des Functional Coverage zur Erfüllung des Verifikationsplans. Eine automatisierte IP-Core Verifikation ist daher nach aktuellem Entwicklungsstand der Testbench noch nicht möglich. Dennoch konnte die korrekte Funktionsweise der Testbench durch die abschließenden Tests des SpaceWire Light IP-Cores gezeigt werden. Das Ergebnis dieser Arbeit ist somit eine voll funktionsfähige parametrisierbare UVM-Testbench zur Verifikation von SpaceWire IP-Cores, für die zur automatisierten Verifikation noch ein Scoreboard und die Beschreibung des Functional Coverage benötigt wird.

5 Ausblick

Zum Schluss soll noch ein kurzer Ausblick auf die Entwicklung des Scoreboards, sowie die Beschreibung des Functional Coverage gegeben werden. Hierbei handelt es sich um die noch benötigten Komponenten zur automatisierten Verifikation.

5.1 UVM-Scoreboard und Functional Coverage

In der aktuellen Version der Testbench existiert das UVM-Scoreboard als Template (Datei spw_scoreboard.svh im Ordner der Env). Die benötigten Verbindungen zwischen den Analyse-FIFOs des Scoreboards und den Analyse-Ports der Agents werden bereits während der Connect-Phase in der Env hergestellt. Für spätere Funktionstests während der weiteren Entwicklung des Scoreboards stehen somit Request- und Response Items, aufgebaut durch die drei Monitore, für einen Vergleich auf TLM-Ebene über zwei Ports (spw_xxx_i bzw. spw_xxx_o) zur Verfügung.
Als Grundlage für die Beschreibung des Functional Coverage ist bereits ein Verifikationsplan für SpaceWire IP-Cores vorhanden, der alle verifikationsrelavanten Inhalte des SpaceWire Standards abdeckt. Die Functional Coverage ist dann entweder durch Cover Groups (Bins, Cross Coverage) oder durch Assertions zu beschreiben.

Literaturverzeichnis

AEROFLEX GAISLER AB, 2008. *SpaceWire CODEC with RMAP*. Actel Corporation. Version 1.0.2. Auch verfügbar unter: https://www.actel.com/ipdocs/Spacewire_DS.pdf.

AYNSLEY, John u. a., 2016. *SystemVerilog Golden Reference Guide*. Version 7.0. Doulos. ISBN 978-1-910062-08-1.

ECSS-E-ST-50-12C, 2008. *Space engineering - SpaceWire - Links, nodes, routers and networks*. Noordwijk, The Netherlands. Standard. European Cooperation for Space Standardization.

ECSS-S-ST-00-01C, 2012. *ECSS system - Glossary of terms*. Noordwijk, The Netherlands. Glossar. European Cooperation for Space Standardization.

IEEE STD 1394-1995, 1996. *IEEE Standard for a High Performance Serial Bus*. New York, NY. Standard. Institute of Electrical und Electronics Engineers, Inc.

IEEE STD 1596.3-1996, 1996. *IEEE Standard for Low-Voltage Differential Signals (LVDS) for Scalable Coherent Interface (SCI)*. New York, NY. Standard. Institute of Electrical und Electronics Engineers, Inc.

IEEE STD 1800-2012, 2013. *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. New York, NY. Standard. Institute of Electrical und Electronics Engineers, Inc.

IEEE STD 1800.2-2017, 2017. *IEEE Standard for Universal Verification Methodology Language Reference Manual*. New York, NY. Standard. Institute of Electrical und Electronics Engineers, Inc.

ISO/IEC 14575:2000, 2001. *IEEE Standard for Heterogeneous InterConnect (HIC) (Low-Cost, Low-Latency Scalable Serial Interconnect for Parallel System Construction)*. New York, NY. Standard. Institute of Electrical und Electronics Engineers, Inc.

MENTOR GRAPHICS, 2018a. *Questa SIM Command Reference Manual*. Mentor Graphics Corporation. Version 10.6d.

MENTOR GRAPHICS, 2018b. *Questa SIM User's Manual*. Mentor Graphics Corporation. Version 10.6d.

MENTOR GRAPHICS, Verification Academy Team, 2013. *UVM Cookbook*. Mentor Graphics Corporation. Version 2013. Auch verfügbar unter: https://verificationacademy.com/cookbook/uvm.

SPEAR, Chris; TUMBUSH, Greg, 2012. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Third Edition. Springer. ISBN 978-1-4614-0714-0.

UVM Cookbook: Code-Beispiele. Auch verfügbar unter: https://verificationacademy.com/cookbook/code-examples.

VAN RANTWIJK, Joris, 2013. *SpaceWire Light manual*. Version 20130504.

VAN RANTWIJK, Joris. *SpaceWire Light source code*. Auch verfügbar unter: https://opencores.org/projects/spacewire_light.

WOLTER, Stefan, 2018a. *Lecture Notes: Hardware Verification with SystemVerilog*.

WOLTER, Stefan, 2018b. *Skriptum: Hardware-Entwurf mit VHDL*.

Anhang

I. SpaceWire IP-Core Verifikationsplan

II. SpaceWire ECSS-E-ST-50-12C Standard Zusammenfassung

VERIFICATION PLAN FOR SPW IP-CORE

Section	Title	Description	Link	Type	Weight	Goal	Verifies
1	**Functional Coverage**						
1.1	**Normal operation behavior**						
1.1.1	SpaceWire coding scheme	Data - Strobe (DS) signal encoding					
1.1.2	Transmit clock recovery	Transmit clock recovery from Data and Strobe signals					
1.1.3	Data char transmission order	Bit transmission order of byte data characters					
1.1.4	Link initialization	NULL / FCT handshake protocol, consider NULL / FCT handshake rules (e.g. NULL / FCT timeouts)					
1.1.5	Data - Strobe (DS) signal reset	Controlled reset after transmitter reset or transmission stop (link error), signal reset order					
1.1.6	Initial data signalling rate	After reset / disconnect, const. during link initialization					
1.1.7	First Bit	First bit to send (Data signal) after reset					
1.1.8	Time-code control flags	Set value of the two MSBits of time-codes					
1.1.9	Flow control - transmit / receive FCTs	Sending FCTs based on free Rx buffer space, sending N-Chars based on no. of FCTs received					
1.1.10	Flow control - transmitter credit count	Inc. for every FCT received, dec. for every N-Char sent, only send L-Chars					

		while count is 0					
1.1.11	Flow control - receiver (outstanding) credit count	Inc. for every FCT sent, dec. for every N-Char received, do not send FCTs while count too high					
1.1.12	Initial FCT transmission / reception	Initial no. of FCTs to transmit based on Rx buffer size					
1.1.13	Character / control-code transmission priority	Based on specified priority, time-codes and FCTs first and sent next immediately if req.					
1.1.14	Link maintenance	Transmission of NULLs on an idle link (keep link active)					
1.1.15	FIFO agent to DUT transmitter interface	Expected DUT response on TICK_IN and TX_WRITE input signals					
1.1.16	DUT receiver to FIFO agent interface	Expected DUT response via time-code output signals (TIME_OUT, CONTROL-FLAGS_OUT)					
1.1.17	DUT receiver before first NULL	Receiver shall ignore all incoming N-Chars, L-Chars, parity- and escape errors					
1.1.18	Receiver NULL (first NULL) detection sequence	Sequence to be detected as first NULL during NULL / FCT handshake					
1.1.19	AutoStart during link initialization	Start sending NULLs automatically after first NULL detected (if link is not disabled)					
1.1.20	System time distribution	Send time-code if TICK_IN asserts, inc. a time-counter on each tick, valid on active (running) link					
1.1.21	System time validation	Check validity of received time-codes, update with invalid codes but do not assert TICK_OUT					

1.2	**Error operation behavior**						
1.2.1	Error recovery scheme (link error)	Error detection, reset of error detecting end (disconnect), FIFO buffers handling, link re-initialization					
1.2.2	Escape error (link error / RxErr)	ESC followed by ESC, EOP or EEP					
1.2.3	Parity error (link error / RxErr)	Parity Bit causing even parity (odd expected), parity bit value based on coverage specification					
1.2.4	Credit error (link error)	Tx credit count too high, Rx buffer not ready to receive N-Chars (no more N-Chars expected)					
1.2.5	Disconnect error (link error / RxErr)	Based on standardized timeout window (ns), missing transition on Data or Strobe (receiver)					
1.2.6	Link error during link initialization	Defined as RxErr OR { gotNULL AND [(gotFCT) OR gotN-Char OR gotTime-Code] }					
1.2.7	Link error on active (running) link	Defined as RxErr OR Credit error (RxErr = Receiver error)					
1.2.8	Character sequence error (link error)	During link initialization, unexpected reception of characters / codes (violation of handshake protocol)					
1.2.9	“Exchange of silence” protocol	Bilateral and mutual disconnection detection (link disabled or link error at either end)					
1.2.10	DUT error flags	Check detected errors flagged at DUT output pins (Disconnect, parity, escape and credit)					

1.3	**Exception operation behavior**						
1.3.1	Simultaneous signal transitions	On Data and Strobe lines (due to data corruption)					
1.3.2	Parity error during link initialization	Parity error detected while waiting to start (after first NULL received)					
1.3.3	Empty packets handling	EOP or EEP followed directly by EOP or EEP (e.g. due to a link error)					
1.3.4	Disconnect error while “waiting to start”	Timeouts (disconnect, waiting for NULL) used to force a reset + disconnect error at both link ends					
1.3.5	Parity error while “waiting to start”	Parity error during link initialization is ignored, identical to disconnect error					
1.3.6	Exception link errors	Data corruption / error in a EOP or EEP, corruption / error in a NULL or FCT sent for a packet					
2	**Assertions**						

Item No.	Category	Description	Explanation / Comment	Page [ECSS-E-ST-50-12C]	Verification
1	Signalling	SpaceWire link speed - data signalling rate	Min. 2 Mbit/s - Max. 400 Mbit/s (full-duplex, serial, p2p)	13	
2	Coding	SpaceWire coding scheme	DS-DE - data-strobe, differentially ended (IEEE 1355-1995)	24	
3	Signalling	SpaceWire signalling technique	LVDS - Low Voltage Differential Signalling (ANSI/TIA/EIA-644)	26	
4	Signalling	LVDS failsafe operation	Receiver outputs go to high state (“1” - inactive) when: 1. Receiver powered, driver not powered 2. Inputs short circuited 3. Input wires disconnected	27	
5	Coding	Transmission clock recovery	DS encoded clock is recovered by XORing data (D) and strobe (S) signals	28	
6	Transmission	Data character transmission order	Byte data values are transmitted least significant bit (LSB) first	28	
7	Link	Link initialization	NULL / FCT handshake protocol	29	
8	Link	Flow control - general	Receive buffer management via flow control tokens (FCTs)	29	
9	Link	Link disconnection detection	No new data bit within a 850 ns timeout window after last bit received	30, 64, 75	
10	Link	Parity error detection	Parity error detected → link restart (re-initialization) → see item no. 21	30	
11	Link	Link error recovery	Perform a link re-initialization	30, 75, 107	
12	Encapsulation	Data encapsulation - packet format	<dest. id’s><cargo> <EOP/EEP>; 0 dest. id’s = p2p link; a dest. id is one data character; number of dest. id’s not delimited; cargo = 1 or more data chars	31, 87	
13	Encapsulation	Next packet detection	First data character after EOP/EEP marks start of the next packet	31, 88	
14	Coding	Strobe signal generation	Strobe signal toggles whenever Data signal does not change from one bit to the next bit	45	
15	Transmission	Simultaneous transitions on Data and Strobe	Simultaneous transitions (data corruption) shall be tolerated by the receiver	45	
16	Transmission	Transmitter reset / transmission stop	First reset Strobe signal (0), then reset Data signal (0); delay between resets is between 555 ns to period of transmission clock	45, 83	

Color key	Importance
	Needs to be verified
	No need to be verified
	Can’t be verified / already covered

17	Signalling	Different data signalling rates	Different data signalling rates per link direction / different links are allowed	46	
18	Signalling	Initial data signalling rate (after reset / disconnect)	(10±1) Mbit/s initially, unchanged unitl full connection (Run State)	47	
19	Link	Escape error (Link error)	ESC followed by ESC, EOP or EEP (FCT or single data character expected); activated after first NULL received	53	
20	Link	Parity bit coverage	Previous 8 bits of a data character or previous 2 bits of a control character + current parity bit + current data-control flag → see item no. 21	54	
21	Link	Parity error (Link error)	Parity bit produces even parity (odd parity expected); activated after first NULL received	54, 76	
22	Transmission	Transmitter reset / Link error	Set Strobe signal and Data signal to 0 (Strobe first to avoid simultaneous transition of Data and Strobe) → see item no. 16	54, 71	
23	Transmission	First bit after reset	First bit sent after reset is a parity bit set to 0 → first transition on Strobe line	54	
24	Transmission	EOP / EEP encoding (in case 8 data bits / 1 control flag)	Control flag = 1 → EOP = 0b00000000, EEP = 0b00000001 (Data bits)	55	
25	Transmission	Time-code control flags	Control flags of a time-code (two most significant bits) are set to 0	56	
26	Transmission	Flow control - transmit / receive FCTs	Send FCTs to indicate free space in receive buffer (one FCT = space for 8 more N-Chars) → for each FCT received 8 more N-Chars can be sent	58	
27	Transmission	Flow control - transmitter credit count	Increment by 8 for every FCT received, decrement by 1 for every N-Char sent; while count = 0 : only send L-Chars (NULL, FCT or time-code)	58	
28	Link	Credit error (Link error)	Occurs if a FCT is received and the transmitter credit count is currently > 48; also occurs if an N-Char is received and BUFFER_READY is not asserted → no N-Chars are expected; activated whenever a link is established	58, 62, 70	
29	Transmission	Initial number of FCTs to transmit (after reset / disconnect)	According to receive buffer size, max. 7 FCTs (56 N-Chars)	58	

30	Transmission	Flow control - receiver (outstanding) credit count	Increment by 8 for every FCT sent, decrement by 1 for every N-Char received; while count > 48 : do not send FCTs; count = 0 after reset	59	
31	Transmission	Character / control-code transmission priority	1. Time-Code, 2. FCTs, 3. N-Chars, 4. NULLs; when transmission of time code or FCT req. → send immediately after current character / control code	59	
32	Link	Link maintenance	Transmit NULL control code (NULLs) to keep link active (sent when there are no N-Chars, FCTs or Time-Codes to transmit)	60	
33	Transmission	Host system to transmitter interface	TICK_IN assert: send current Time-Code from TIME_IN + CONTROL-FLAGS_IN immediately; TX_READY assert: transmitter ready to receive next N-Char from host → host puts N-Char on TX_DATA line and asserts TX_WRITE; TX_READY de-assert: transmitter has registered N-Char	61	
34	Transmission	Receiver to host system interface	TICK_OUT assert: valid Time-Code received → Time-Code is put on TIME_OUT and CONTROL-FLAGS_OUT output; BUFFER_READY assert: host ready to receive next N-Char from receiver → receiver puts N-Char on RX_DATA line and asserts BUFFER_WRITE; BUFFER_READY de-assert: host has registered N-Char	61, 62	
35	Link	Receiver before first NULL (during link initialization)	Receiver ignores all N-Chars, L-Chars, parity errors and escape errors until first NULL is received (first NULL sequence detected)	61	
36	Transmission	Initial receive buffer space	Due to NULL / FCT handshake during initialization the host system must initially flag to be able to receive (at least) 8 N-Chars (before / during link init) → FCT for NULL / FCT handshake can be sent (see item no. 29)	63	
37	Transmission	Transmit / receive FIFO buffering	FIFOs simplify the interfaces to the host system; they are initially empty after reset; flags for “half-full” or “half-empty” can be used	63	

38	Link	Disconnect error (Link error)	Occurs if no new data bit is received within a 850 ns (727 ns - 1000 ns) timeout window after last bit received → time interval from last transition on either D or S exceeds 850 ns; activated after first bit received	30, 64, 75, 83	
39	Link	Link error during link initialization	Disconnect error OR Parity error OR Escape error OR { gotNULL AND [(gotFCT) OR gotN-Char OR gotTime-Code] }	64	
40	Link	NULL / FCT handshake rules	Transmit at least one NULL during initialization (even if first NULL was received before); reset if no NULL is received within 32 us after last reset if AutoStart is enabled; reset if no FCT is received within 12.8 us after first NULL received; → see item no. 7	66	
41	Link	Link error on active (running) link	Disconnect error OR Parity error OR Escape error OR Credit error = RxErr OR Credit error	67	
42	Link	Receiver NULL (first NULL) detection sequence	b011101000	68	
43	Link	Receiver error / RxErr (Link error)	Disconnect error OR Parity error OR Escape error → see item no. 38, 21, 19	69	
44	Link	Character sequence error (Link error)	Ignore N-Chars and L-Chars received before first NULL received; once a NULL is received → FCT received before a NULL is sent; N-Chars or time-codes received before both a NULL and FCT is received; only during initialization	70	
45	Transmission	AutoStart during link initialization	Start sending NULLs (start link) automatically after first NULL detected (while link is not disabled)	70	
46	Transmission	Empty packet handling	Empty packets - EOP or EEP followed directly by another EOP or EEP (e.g. due to a link error) shall be discarded → discard second EOP or EEP	75, 77, 88	
47	Transmission	Flag link errors up to the network level	Disconnect error + Parity error + Escape error + Credit error; only while link is established (Run state) → do not flag/report during link initialization	76, 77	
			Bilateral and mutual disconnection detection due		

48	Link	“Exchange of silence” protocol	to: 1. Link disconnection or 2. Link error at either end of the link → force link re-initialization at both ends	77	
49	Link	Exception conditions	Disconnect error while waiting to start → use timeouts (disconnect, waiting for NULL) to force reset + disconnect error detection	78	
50	Link	Exception conditions	One-sided link connection A to B → e.g. cable break / link being plugged; A does not receive NULLs from B, B does not receive FCTs in the next state → A times out and resets → B detects disconnect and resets too	80	
51	Link	Exception conditions	Parity error while waiting to start → effect identical to a disconnect error during link initialization	81	
52	Link	Exception conditions	One end starts as other end disconnects → End A arrives at Started state 12.8 us before end B → 12.8 us timeout at A expires, A resets → B detects disconnect error and resets too	81	
53	Link	Exception conditions	Strobe line disconnected → Data sequence of 01010101010… results in initialization fail; produces parity errors after first NULL received if sequence is seen as control characters; produces series of data characters with value AA hex after first NULL received if sequence is seen as data characters	82	
54	Link	Exception conditions	Data line disconnected → Data sequence of 11111111111… results in initialization fail; produces parity errors after first NULL received for both control characters and data characters (produces even parity)	82	
55	Link	Link timing (exchange timeout periods)	6.4 us nominal = 5.82 us - 7.22 us; 12.8 us nominal = 11.64 us - 14.33 us	83	
56	Transmission	System time distribution	6-bit time counter in each node / router; if TICK_IN asserts → time-master interface increments time-counter and sends out a time-code with the new time-counter value; time-code only sent when connection	84	

			established; reset time-counter on reset/re-connect/disconnect		
57	Transmission	System time validation	Check if received time-code > previously received time-code (one more, modulo 64); update with invalid time-codes but do not assert TICK_OUT	84, 85	
58	Network	SpaceWire nodes	Comprise one or more SpaceWire link interfaces + interface to host system → bidirectional interface between SpaceWire network and application system	100	
59	Network	Network level errors	Link error (see item no. 47), EEP received, invalid destination address	101	
60	Network	Network level error detection	Error detected at source or destination node: flag up to the application level; Error detected within routing switch: flag to external pin or internal status register	101	
61	Link	Link error recovery scheme	Detect error → disconnect link / stop current packet transmission → discard remaining (incoming) N-Chars up to and including next EOP or EEP by link interface → terminate received part of the packet with EEP (if a complete packet has just been received up to the EOP or EEP then an EEP does not need to be added to the receiver buffers (receive FIFOs)) → delete data in the transmitter buffers (transmit FIFOs) until the next EOP; no link re-initialization after link error until receiver buffer(s) (receive FIFO(s)) has / have space for more N-Chars when 1. FIFO(s) currently full so EEP cannot be written into the receiver buffer(s) 2. No space for at least 9 N-Chars (EEP + 8 N-Chars) → link interface unable to send a FCT (re-initialization fails)	101, 108, 109	
62	Network	Flag link errors up to the application level	When an EEP is received at a destination node → discard corresponding packet or use it; can resend packet that was being sent when link error was reported (for important information)	102, 109	
			Received packet terminated by an EEP means the packet		

63	Network	Network level error recovery - EEP received	was terminated prematurely; EEP received by routing switch → transfer like EOP; EEP received by destination node → flag up to the application level but transfer as normal (to the application level)	102, 108	
64	Network	Network level error recovery - invalid destination address	Logical address whose routing table entry refers to a non-existent output port; routing switch → discard packet (discard arriving N-Chars unitl and including next EOP or EEP), flag to external pin or internal status register, discard empty packets (EOP or EEP followed by EOP or EEP) by deleting the second EOP or EEP; node → discard packet with unexpected destination address, can flag to host system (application level)	103, 108	
65	Link	Exchange level error recovery scheme	1. Detect error, 2. Disconnect link, 3. Report error to the network level, 4. Link re-initialization if link interface is still enabled	107	
66	Link	Exception link errors	Non p2p link: 1. Corruption / error in first dest. id / header byte (first byte after an EOP or EEP) → routing switch: entire packet lost (first routing switch discards the packet, see item no. 64), destination node: can flag error to host system (application level); 2. Corruption / error in a EOP or EEP → two packets lost: the one before the EOP in error (no EOP received) + the one in the transmitter buffer which is deleted until the next EOP; 3. Corruption / error in a NULL or FCT sent for a packet → packet being sent discarded from that point on (character before corruption unknown)	109, 110	