

# CHALMERS



## Development of a SpaceWire interface in VHDL

*Master of Science Thesis in Programme Electrical Engineering*

DAVID JULIUSSON

Department of Computer Science and Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY  
Göteborg, Sweden, February 2012

The Author grants to Chalmers University of Technology and University of Gothenburg the nonexclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants the he is the author to the Work and warrants that the Work does not contain text, pictures or other material that violates the copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Development of a SpaceWire interface in VHDL

© David Juliusson, February 2012.

Examiner: Kjell Jeppson.

Department of Computer Science and Engineering  
Chalmers University of Technology  
SE-412 96 Göteborg  
Sweden  
Telephone +46 (0)31-772 1000

Department of Computer Science and Engineering  
Chalmers University of Technology

## Abstract

In space, as on earth, there is a need for fast, reliable and reusable point-to-point data communication. The SpaceWire standard developed by the European Space Agency together with the European Space Industry and Academia aims to provide a unified standard that can support most data communication needs during space missions. The goal of this master thesis is to design and implement a SpaceWire Codec, compliant with the SpaceWire standard as described in ECSS-E-ST-50-12C, and make the design efficient enough to support the needs for the foreseeable future.

During this master thesis a SpaceWire Codec have been designed, implemented and verified. During the design, focus has been on size, speed and re-usability as the pre-existing solutions does not meet all the requirements emerging at RUAG Space AB. The SpaceWire Codec implemented during this theses is used to handle the low level reception and transmission of data and control characters over the SpaceWire link and provide an easy to use interface.

Verification and synthesis of the SpaceWire Codec shows that the design can deliver contiguous transfer rates of one useful byte, transferred and received every BusClk, while still being lightweight in terms of digital logic. Further verification steps needs to be performed in order to make the SpaceWire Codec suitable for its planned space missions.

## Sammanfattning

I rymden, såväl som på jorden, finns ett behov av snabba och robusta data interface. SpaceWire standarden, utvecklad av ESA tillsammans med företag och universitet, är ett försök att skapa en standard där större delen av den datakommunikation som finns i såväl satelliter som raketerna ska kunna rymmas. Målet med det här ex-jobbet är att designa och implementera SpaceWire standarden som den beskrivs i ECSS-E-50-12A och samtidigt göra designen tillräckligt effektiv för att RUAG Space ska kunna använda modulen i framtida produkter.

Under arbetets gång har en SpaceWire Codec blivit designad, implementerad och verifierad. Under designfasen låg fokus på storlek, prestanda och användbarhet då äldre lösningar inte längre klarar de krav som växt fram på RUAG Space. Den SpaceWire Codec som implementerats här sköter all lågnivå hantering av data över SpaceWire länken och ger samtidigt resten av designen lättanvänt interface.

Verifikation och syntes av den föreslagna lösningen visar att designen klarar den prestanda som sattes upp som mål i början av arbetet. Designen är dessutom förhållandevis effektiv vad gäller digital logik samt går att konfigurera beroende på vilka behov som finns. Ytterligare tester och en full verifierings svit krävs dock innan den SpaceWire Codec som implementerats är klar för sina kommande missioner.

## About the author

**David Juliusson** is a Master of Science student in Integrated Electronic System Design (Department of Computer Science and Engineering) and began his studies at Chalmers University of Technology in 2006 in the Electrical Engineering programme. Having prior experience from developing modules for space applications for both FPGAs and ASICs the main challenge during this thesis has been to write the requirements and to design robust asynchronous interfaces.

## Preface

This master thesis was performed at the department of Integrated Electronics System Design at Chalmers. The work started at the spring of 2010 and the final report was completed during April 2012. The practical work was performed at the company RUAG Space AB in Gothenburg.

RUAG Space is a subdivision of RUAG Group in Switzerland. The Company specializes in designing and producing both hardware and software for space missions with data handling systems as one of their major product areas.

Of the people involved in this project I would like to extend a special thanks to the following from RUAG Space.

I would like to thank Andreas Karlsson my tutor at RUAG Space, for his support in making the design fit company needs. I would also like to thank Peter Spjuth and Karl Engström for their insight into asynchronous interfaces and ASIC-design.

From Chalmers I would like to thank Kjell Jeppson for his support and knowledge of microelectronics.

Göteborg 2012

David Juliusson

## Bit Numbering

The following conventions are used for bit numbering:

- The Most Significant Bit (MSB) of a vector has the leftmost position.
- The Least Significant Bit (LSB) of a vector has the rightmost position.
- Unless otherwise indicated, the MSB of a vector has the highest bit number and the LSB the lowest bit number.

## Radix

The following conventions are used for writing numbers:

- Binary numbers are indicated by the subscript “<sub>2</sub>”, e.g.  $1_2$ ,  $1011\_1010\_1011\_1110_2$ ,  $010010_2$  etc.
- Decimal numbers are indicated by the subscript “<sub>10</sub>”, e.g.  $67,8723_{10}$ ,  $47860_{10}$ .
- Hexadecimal numbers are indicated by the subscript “<sub>16</sub>”, e.g.  $E_{16}$   $BABE_{16}$ .
- Unless the Radix is explicitly declared as above the number should be considered to be decimal number.

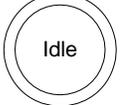
## Signal Names

The following conventions are used for signal names:

- Signal names are written in italics, e.g. *SignalName*.
- Bus indices are indicated with brackets, e.g. *SignalName[12:3]*.
- Signals maybe grouped into subsignals, e.g. *SignalName.SubSignal*.

## Graphics legend

Standard graphics for state- and mode- graphs.

	State or modes are pictured as circulars. Double circle indicates the Reset/Initial state or mode.
	Single circle indicates State or modes.
	Normal transition
	Exceptional transition

## Basic data types

Byte	8 bits of data
HalfWord	16 bits of data
Word	32 bits of data

## Character types

N-Char	Data characters, EOP and EEP.
L-Char	FCT, NULL and TimeCode

## Abbreviations

ASIC	Application Specific Integrated Circuit
BusClk	System clock
DFF	Digital Flip-Flop / register
DS	Data-strobe
ESA	European Space Agency
ECSS	European Cooperation for Space Standardization
FPGA	Field Programmable Gate Array
HW	Hardware
Id	Identifier
I/F	Interface
IO	Input/Output
IP	Intellectual Property
LSB	Least Significant Bit
LSW	Least Significant Word
LVDS	Low Voltage Differential Signal
MSB	Most Significant Bit
MSW	Most Significant Word
MTBF	Mean Time Between Failures
N/U	Not Used
RTL	Register Transfer Layer
SDF	Standard Delay Format
SEU	Single Event Upset
SW	Software
TB	Test Bench
TBC	To Be Confirmed
TBD	To Be Determined
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

# Contents

1	Introduction .....	1
1.1	Purpose .....	1
1.2	Delimitations .....	1
1.3	Method .....	2
1.4	Key results .....	2
1.5	Decomposition .....	3
2	System view and background information .....	4
2.1	SpaceWire as a concept .....	4
2.1.1	Applications .....	5
2.1.2	From test to mission .....	6
2.1.3	Network .....	6
2.1.4	Additional protocols .....	7
2.2	SpaceWire heritage and improvements .....	8
2.2.1	Pre-existing standards .....	8
2.2.2	Improvements .....	8
2.2.3	Revision .....	8
2.3	SpaceWire protocol .....	9
2.3.1	Protocol levels .....	9
2.3.2	Levels of the Codec .....	9
2.4	The SpaceWire Codec .....	10
2.5	Digital logic in space .....	12
2.5.1	Effects of radiation .....	12
2.5.2	ASIC technology .....	12
2.5.3	FPGA technology .....	12
3	Requirements and demands .....	13
3.1	Company specific demands .....	13
3.1.1	Small footprint .....	13
3.1.2	Straightforward configuration parameters .....	13
3.1.3	Configurable transfer rates .....	13
3.1.4	Small BusClk asynchronous regions .....	14
3.1.5	Easy post synthesis placement .....	14
3.1.6	Self calibrating timeout counters .....	14
3.1.7	BusClk synchronous interface .....	14
3.2	Non-formal demands .....	14
3.2.1	Coding standard .....	14
3.2.2	Design partitioning .....	14
3.3	Requirements .....	15
3.3.1	Signal encoding and transition constraints .....	15
3.3.2	Data signal rate constraints .....	16
3.3.3	Serial interface characters .....	17
3.3.4	Link timing .....	18
3.3.5	Parity generation and error detection .....	18
3.3.6	Link status signals .....	19
3.3.7	Start-up procedure .....	20
3.3.8	N-Char buffers .....	22
3.3.9	Flow control .....	23

3.3.10	TimeCode interface .....	24
3.3.11	Data and control interface.....	25
4	Functional design .....	27
4.1	Design partitioning.....	27
4.1.1	Codec as described in the SpaceWire standard.....	27
4.1.2	The proposed design .....	28
4.2	Overview .....	29
4.2.1	Function .....	29
4.3	Data Interface .....	29
4.4	Configuration .....	30
4.4.1	Data sample groups.....	30
4.4.2	Rx Async IF FIFO .....	30
4.4.3	Character Buffer .....	31
4.4.4	Rx FIFO Size .....	31
4.4.5	Data transmit groups .....	31
4.4.6	Tx FIFO Size .....	31
4.5	Rx Pipeline.....	32
4.5.1	Receiver .....	32
4.5.2	Rx asynchronous interface.....	33
4.5.3	Rx token handler .....	34
4.5.4	Rx FIFO .....	35
4.6	Tx Pipeline.....	36
4.6.1	Timer .....	37
4.6.2	Tx FIFO .....	37
4.6.3	Tx token generator.....	37
4.6.4	Tx asynchronous interface.....	37
4.6.5	Data strobe generator .....	39
4.6.6	Tx driver.....	39
5	Verification .....	41
5.1	Simulation.....	41
5.1.1	Test bench description.....	41
5.1.2	Test bench commands .....	42
5.2	Test bench receiver capabilities .....	43
5.2.1	Monitors .....	43
5.3	Test bench transmitter capabilities .....	43
5.4	Test procedures.....	44
5.4.1	Start-up test.....	44
5.4.2	TimeCode test .....	45
5.4.3	Data test .....	45
5.4.4	FCT test.....	47
5.4.5	FIFO flush test .....	48
5.5	Verification in hardware .....	49
6	Results .....	50
6.1	Implementation results .....	50
6.1.1	Design .....	50
6.1.2	RTL implementation .....	51
6.1.3	Test bench implementation .....	51
6.1.4	Hardware verification .....	51
6.2	Size .....	52

6.2.1	Minimum configuration .....	52
6.2.2	Maximum configuration.....	53
6.2.3	Size comparison .....	53
6.3	Performance .....	54
6.3.1	MH1.....	54
6.3.2	ATC18 .....	55
6.4	Future development.....	55
7	Conclusions .....	56
8	References .....	57

# 1 Introduction

RUAG Space is interested in developing a new SpaceWire interface to complement the existing solution provided by the European Space Agency (ESA). The main reasons for designing a new company controlled SpaceWire interface are twofold. The existing solution does not meet all size and performance needs for future ASIC and FPGA designs. There is a need for both light weight configurations, suitable for FPGA remote terminal applications, as well as high bandwidth configurations that can handle transmitting as well as receiving a large number of bits every clock cycle. Secondly the licensing process, when using the existing solution, as well as the cumbersome procedures post synthesis is time consuming.

The new module should fit three major roles. The first is as a fast, yet lightweight SpaceWire interface in the modular test environment called COFTA. The second role is to provide small remote terminal FGPA designs with an area effective SpaceWire interface. The third is to take the role of the existing solution as a fast data interface, preferably being able to handle more bits per *BusClk* (System clock) than the design used today.

## 1.1 Purpose

The purpose of this master thesis is to write the specification for the new SpaceWire interface, design and implement the SpaceWire interface in VHDL and finally verify the design by running the design in a test-bench as well as in hardware, on a FPGA platform.

The design goals are to make the design an all purpose SpaceWire module that is able to handle receiving and transmitting one useful byte of data every *BusClk* cycle, to design the input and output regions as small and uncomplicated as possible in order to minimize skew and to make the design suitable for high bandwidth applications.

## 1.2 Delimitations

The thesis covers the implementation of a SpaceWire Codec, aimed to be used for space missions, as described in [ECSS SPW] with a few exceptions.

- Chapter 5: The physical level of the SpaceWire interface is not covered as it is not a part of the digital design.
- Chapter 10: The network level is only covered briefly as it is not a part of the work done during the digital design.
- The work does not cover Low Voltage Differential Signal (LVDS) drivers as these will be supplied by third party.
- Flight readiness: The SpaceWire Codec designed and tested during this thesis will not be ready for flight production. A more complete test-bench covering all requirements will be needed, together with code inspection and validation.

## 1.3 Method

During the three different stages of the design the following methods were used.

- The method used to design the SpaceWire Codec was to first find the key features needed for each application. This was done by studying both the SpaceWire standard as well as the complementing SpaceWire protocol standards. The design proposed in the SpaceWire standard was analysed and a new design, with a different topology more suitable for the needs of RUAG Space AB was proposed.
- During the development of the code, focus was on implementing the design as efficiently as possible and to implement effective ways to tailor size versus performance. The development of the code also took into account the hazards of digital logic in space in order to make the design robust enough to qualify for space missions.
- During the verification of the design, test cases were written and a full spacewire test bench was developed. The basic functionality of the SpaceWire Codec was verified in both simulation and during tests on a FPGA platform. The tests worked as a proof of concept for the design and the SpaceWire Codec has since been submitted to a full verification by RUAG Space AB in order to qualify for use during missions.

## 1.4 Key results

The basic design of the SpaceWire Codec developed during this thesis proved to be good enough for use in space missions, after further testing by RUAG Space AB. The codec has already been incorporated into ASIC and FPGA designs as well as into IP-cores. The design proved to be easy to integrate into projects and the BusClk synchronous interface spawned the idea of a standardized internal interface for data packets, now used by many of the designs developed at RUAG Space AB in Gothenburg. The implementation of the SpaceWire Codec has proved to be effective in terms of digital logic and the different configuration options allow the hardware designer instantiating the codec to tailor performance and size the specific needs of the project. Synthesis of the design revealed that the SpaceWire Codec is capable of speeds exceeding that of the standard. As a final note the more than 11 000 lines of VHDL code written during this thesis has and hopefully will continue to serve RUAG Space AB for years to come.

## 1.5 Decomposition

The chapters of this master thesis are distributed in the following manner.

- Chapter 2 will present the SpaceWire protocol from a system point of view. This chapter also includes the protocol levels of the SpaceWire Codec implemented during this thesis as well as a short description of the different hazards associated with digital logic in space.
- Chapter 3 contains the requirements derived from the SpaceWire standard as well as the company specific demands placed upon the design by RUAG Space AB.
- Chapter 4 contains an in depth description of the functional design for every block of the design. The chapter describes design decisions made during the development as well as justifications for these.
- Chapter 5 contains the verification of the SpaceWire Codec. The chapter describes both the functional tests performed during the verification as well as the setup for the tests done in hardware.
- Chapter 6 contains the final results of work done during this thesis. This includes the test results as well as the performance and size of the final design. The chapter also include future improvements to further develop the SpaceWire Codec proposed in this thesis.
- Chapter 7 contains the conclusion.
- Chapter 8 contains the references.

## **2 System view and background information**

This chapter covers the SpaceWire interface and its role from a system point of view as well as a brief technology study. The chapter is included to give a better understanding of the protocol, where it originated from and the demands and limitations that this applies to the actual design and verification work done during this thesis.

The chapter starts with a short description of the SpaceWire link as a concept, focusing on the system view and possible applications. A description of the development of the SpaceWire standard follows, including pre-existing standards on which the SpaceWire protocol is built and possible future updates. After the introduction to the standard, two sections follows that handle the protocol layers of the SpaceWire interface and the layers that the SpaceWire Codec is responsible for maintaining. The last subsection includes an overview of the specific demands placed on digital hardware designed for space applications as well as a description of the technologies available.

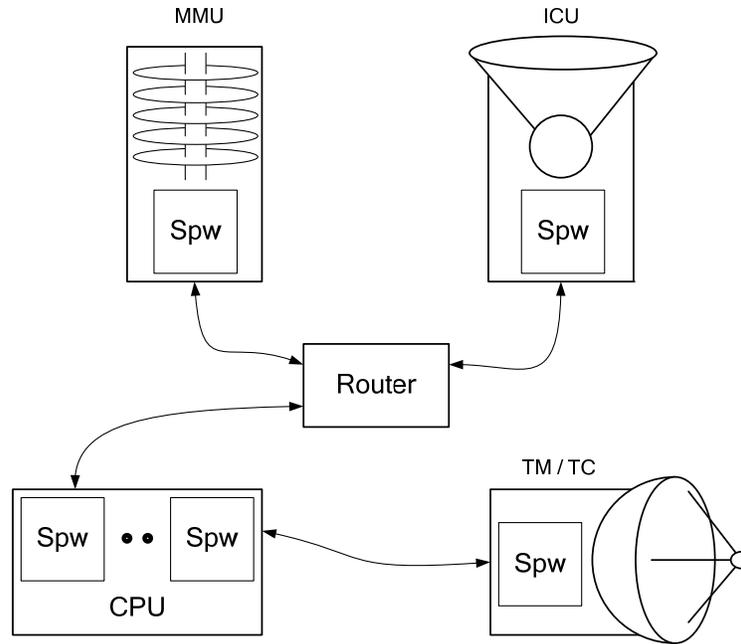
### **2.1 SpaceWire as a concept**

The SpaceWire standard was developed in order to give space missions a versatile, standardized data interface capable of handling multiple roles. The standard specifies a full-duplex, point-to-point, serial data communication link capable of data rates between 2 Mbps and 400 Mbps. The ESA SpaceWire standard [ECSS SPW] covers everything needed to pass information over the link, from the physical level to the network level. The SpaceWire standard also addresses the need for time distribution as well as includes a well defined start-up and error recovery scheme.

Beside the SpaceWire standard, there exist additional protocols that give the SpaceWire link more functionality. The remote memory access protocol (RMAP), as described in [ECSS RMAP] gives the system remote memory access capabilities, enabling the possibility for remote terminal applications without the direct support of a central processing unit (CPU). CCSDS Packet Transfer Protocol (CPTP), as specified in [ECSS PTP] is another protocol that is contained within a SpaceWire packet. PTP is a transfer protocol used to send one packet through a SpaceWire network to the appropriate destination.

### 2.1.1 Applications

The SpaceWire link can support a wide range of applications thanks to the large span of available data rates and the inherent data safety of the protocol. There are also some benefits of using a protocol that is capable of accommodating applications that range from rather slow control interfaces to high speed data buses. This enables the use of standardized reusable equipment like routers and CPUs to be available off-the-shelf. An example of the possible applications is shown in Figure 1 below.



**Figure 1:** The SpaceWire link connecting the system

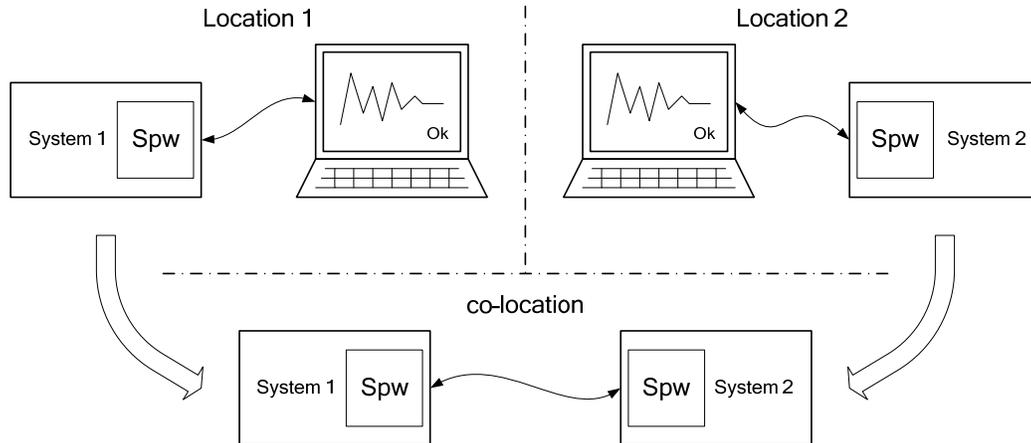
The SpaceWire protocol is able to support various tasks like connecting mass memory units (MMU) and instrument control units (ICU) with the CPU and the Telemetry/Telecommand devices (TM/TC) as seen in the example in Figure 1.

The main benefit of using the same protocol for everything from low speed control applications to high speed data transfers is the re-usability of components and subsystems, lowering the development cost for new systems and increasing system reliability.

The routing schemes make it possible to bypass routers if needed, when using the Target logical address as routing byte. It should be possible to, for instance, connect the MMU directly to the CPU in Figure 1 without any alterations. This can be useful during validation campaigns when the entire system might not be present at one location.

### 2.1.2 From test to mission

The SpaceWire protocol is suitable for use both during space missions and during the development and debugging phase pre-flight.



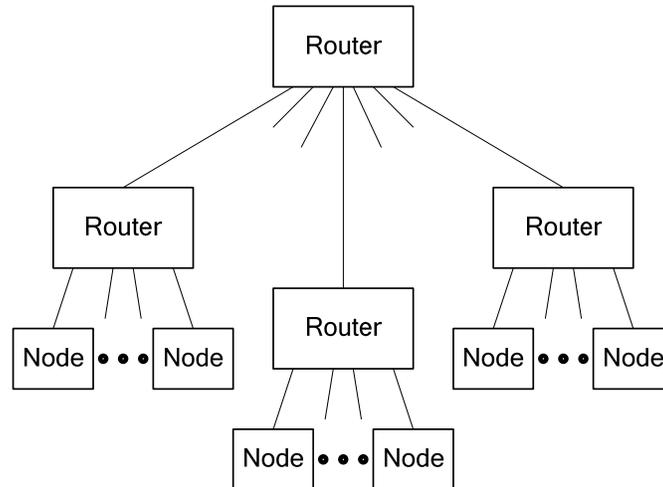
**Figure 2:** Benefits of a standardized interface

Commercial products make it possible to connect a normal PC, via an Ethernet to SpaceWire-bridge, to the system under test as seen in Figure 2. When both systems have been validated, possibly at different location, the integration work can be performed at the assembly site.

### 2.1.3 Network

The network chapters of [ECSS SPW] are not covered during this thesis but a brief introduction is included in order to give a better understanding of the system as a whole.

As the SpaceWire link offer only point-to-point data communication, a network of routers is needed to make efficient use of the resources. Figure 3 depicts one type of network topology available for SpaceWire systems.



**Figure 3:** Network topology

A SpaceWire packet travelling through the network will be fitted with one or more routing bytes in the primary header of the packet. There are two routing schemes that can be used;

- The first scheme uses only a Target Logical Address to guide the packet to the correct destination.
- The second makes use a series of Target SpaceWire Addresses, preceding the Target Logical Address of the packet. These Target SpaceWire Address bytes will be stripped of one by one as they guide the packet to its final destination.

### 2.1.4 Additional protocols

The SpaceWire standard on its own does not cover all the functionality needed during missions and for that purpose, additional protocols can be used to give the system more capabilities. The CCSDS Packet Transfer Protocol (PTP) and the Remote Memory Access Protocol (RMAP) are not covered during the work done for this thesis but a short description is included in order to give an idea of normal system usage.

The CCSDS PTP, as described in [ECSS PTP], is a generic packet transfer protocol used to guide a packet of data from source to destination in a SpaceWire network. The protocol does not provide services that ensure correct or timely data delivery.

RMAP is used to distribute memory access rights to other members of a SpaceWire network. This enables a CPU to control remote terminal devices by accessing memory locations as well as status and control registers.

## 2.2 SpaceWire heritage and improvements

This subsection covers the pre-existing standards on which the SpaceWire protocol was built together with improvements, specific to the SpaceWire protocol and possible future updates.

### 2.2.1 Pre-existing standards

The SpaceWire protocol is a derivative of the IEEE 1355-1995 standard [IEEE 1355-1995]. The SpaceWire standard share the same character setup for data and control character handling but have some additions and clarifications making it suitable for use in space applications.

The electrical interface of the Low Voltage Differential Signalling (LVDS) interface is compliant with the standard ANSI/TIA/EIA-644.

The Data-strobe (DS) encoding of the transmitted bits is defined in IEEE 1355-1995 as well as in IEEE 1394.1995, better known as FireWire.

### 2.2.2 Improvements

Improvements to the existing IEEE 1355-1995-standard have been made to make the SpaceWire protocol more rugged, lower the power consumption, improve EMC-performance and to address ambiguities in the pre-existing standard. The SpaceWire Standard also cover networking solutions and router functionality not implemented during this thesis work.

The main differences between the IEEE 1355-1995-standard and the SpaceWire standard that are relevant to this thesis are listed below:

- Signal level: The SpaceWire protocol use LVDS instead of PECL.
- Character level: The use of the ESC character is explicitly defined.
- Exchange level: Ambiguities during link start-up resolved.

### 2.2.3 Revision

The SpaceWire standard, in its current form, cover transfer rates from 2 Mbps to 400 Mbps but a revision of the standard is pending. One suggestion is that the 2 Mbps boundary might be lowered together with a lengthening of the timeout times.

The upcoming revision also includes a clarification of the TimeCode characters as the existing standard leaves room for interpretations.

## 2.3 SpaceWire protocol

This section will give a brief introduction to the protocol levels associated with the SpaceWire protocol and the levels covered by the SpaceWire Codec developed during this master thesis.

### 2.3.1 Protocol levels

The SpaceWire standard covers the following protocol levels:

- Physical: Defines cables, connectors etc.
- Electrical: Defines voltage levels and noise margins.
- Signal: Defines signal encoding and data signalling rates.
- Character: Defines data and control characters.
- Exchange: Defines link initialization, error detection and recovery as well as flow control and time distribution.
- Packet: Defines packets and data flow over the link.
- Network: Defines the structure for networks of routers and end users.

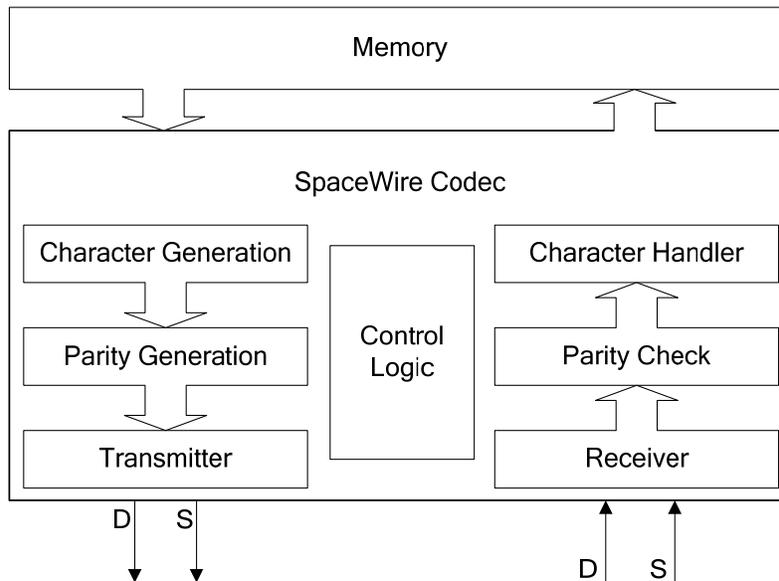
The list of protocol levels above is altered slightly from the original as described in [ECSS SPW]. In order to get a straightforward mapping of the protocol layers associated with the SpaceWire Codec, the original Signal layer have been split into the Electrical level and the Signal level.

### 2.3.2 Levels of the Codec

The SpaceWire Codec is responsible for handling the levels ranging from the Signal level to the Exchange level. The SpaceWire Codec's main purpose is maintaining the link, performing the links initialization sequence when restarted and performing flow control by reception and distribution of flow control tokens. This also means that the SpaceWire Codec is only responsible for the low level maintenance of the link and does not contain any network services.

## 2.4 The SpaceWire Codec

The main functions of the SpaceWire Codec are to handle the low level bit transfers over the link, as well as handle the link initialisation sequence and the error detection and recovery scheme. The Codec is also responsible for the handling of the specific characters used for time distribution and flow control, as well as the characters used for defining packet boundaries and erroneous packet terminations.



**Figure 4:** SpaceWire Codec overview

There are three main structures inside a SpaceWire codec.

- Tx-pipeline: Responsible for transmitting data and control characters.
- Rx-pipeline: Responsible for receiving and validating characters.
- Control logic: Holds the state of the codec and is responsible for the start-up and the termination of the link.

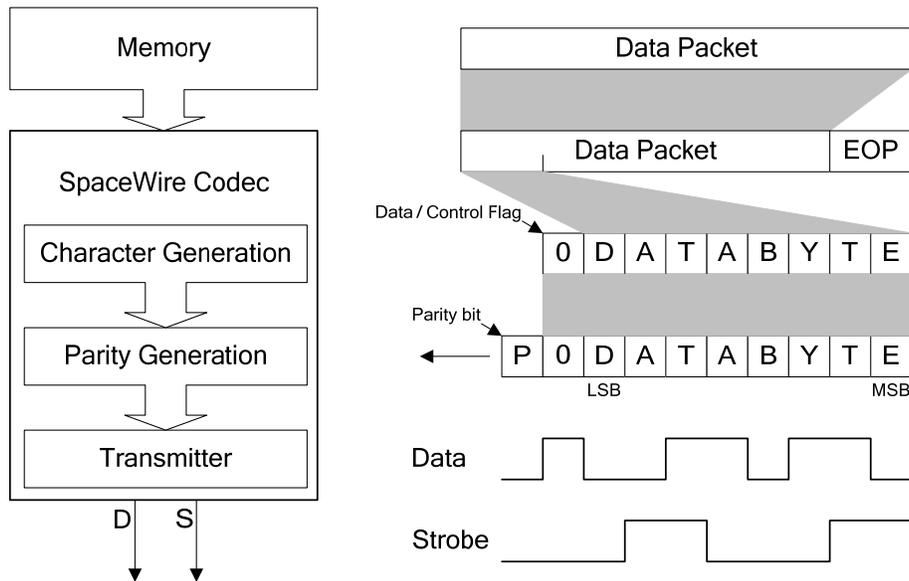
In Figure 4 the Tx-pipeline can be recognised as all structures that have downwards pointing arrows. The transmitted characters can originate from outside sources, depicted as memory in Figure 4, or from internal ones, i.e. the Control logic.

The Rx-pipeline can be recognised as all structures with upwards pointing arrows in Figure 4. The received characters can be consumed within the codec, as in the case of link specific characters, or be distributed to outside receivers for storage or processing.

The Control logic is the glue that keeps the Rx- and the Tx-pipelines in sync.

The SpaceWire Codec as designed during the thesis aims at being a self contained entity providing an easy to use BusClk synchronous interface to the rest of the design. All non BusClk synchronous blocks of the Codec and their asynchronous interfaces are kept module internal. The reason for this is twofold. First, interfacing with a BusClk synchronous module is not as time consuming as interfacing over a clock boundary. The work of designing an asynchronous interface is only performed once, during the initial module design. Further more, including more of the functionality within the Codec makes it easier to optimize the design both in terms of performance and in terms of size.

To clarify the actions performed by a SpaceWire Codec, during nominal operation, a walkthrough of the actions done by the Tx-pipeline during packet transmission follows below.



**Figure 5:** Data flow for transmission

The SpaceWire Codec receives packets of data directly from memory or from other supporting circuits as seen in Figure 5. The codec then generates data and control characters from the incoming data packet to make it ready for transmission. After a character is scheduled for transmission it will receive the parity bit, used for character validation by the receiving end of the SpaceWire link. In the last step, inside the transmitter in Figure 5, the character is serialized, Data-Strobe (DS) encoded and accelerated to double data rate before being handed over to the LVDS driver.

## 2.5 Digital logic in space

This subsection will cover some of the specific problems associated with designing digital hardware for space applications and the techniques used to handle or contain them.

### 2.5.1 Effects of radiation

All digital hardware used in space needs to be designed to take the effects of radiation into account. Besides degradation of the silicon, the one big risk for digital hardware in hazardous environments is a single upset event (SEU). These are caused by the digital hardware being struck by an ion and can cause a register to shift value, thus making the execution falter or fail.

There are several techniques used to deal with the effects of radiation. The logic cells need to be radiation hardened, meaning that they are designed to have a better tolerance to radiation. But as the available technologies get smaller and faster, their radiation tolerance gets worse. To compensate for this triple buffering with voting is used. This ensures that no erroneous values get propagated even if one of the registers would get an SEU, but it does not protect against the sampling of an erroneous value. If an ion were to strike at the same time as the registers are sampling in a new value all three could sample the incorrect value with no chance of recovery. To overcome this triple buffering with a skewed clock can be used. This will force the registers to sample their value at different points in time giving them better radiation protection at the cost of performance.

But even well designed libraries can get an SEU more work needs to be done. The effects of an SEU can be lessened by making the state machines and other control logic SEU proof. This means that even if an SEU occurs, the state machines will have a valid state to fall back to and continue execution.

### 2.5.2 ASIC technology

There are a few ASIC technologies available for digital logic designed for space. The two that the SpaceWire codec, designed during the thesis, was synthesized to were;

- MH1: a 350 nm technology provided by Atmel.
- ATC18: a 180 nm technology provided by Atmel.

One of the goals with the SpaceWire Codec was that it should be capable of 200 Mbps in both receive and transmit with MH1 and double that in ATC18.

### 2.5.3 FPGA technology

There are not many manufacturers of FPGA technology designed for space mission. The FPGAs suitable for space are not as big as commercial ones. One of the goals with this design was that it should be capable of residing in a small Microsemi RTSX-FPGA with only 2000 registers and still leave room to spare for useful digital logic.

## 3 Requirements and demands

This chapter will list the requirement and demands for the design. The chapter starts with specific demands and limitations that RUAG Space put on top of SpaceWire standard, followed by the requirements derived from the SpaceWire standard as described in [ECSS SPW]. The demands and requirements are written to give the SpaceWire Codec a solid foundation on which to design, build and verify. The main difference between the demands and the requirements is that demands use the less binding term “should” where requirements use the term “shall”.

### 3.1 Company specific demands

This sub section lists the company specific demands, as described by RUAG Space to make the design fit company needs. Demands are not written in a way so that they can be easily verified, instead focus have been on describing the general idea of the module and specify what areas of the design to focus on.

#### 3.1.1 Small footprint

The module should ideally be light weight in terms of digital logic. This point is one of the main concerns of RUAG Space as the main goal of the design was to be able to fit inside the rather small FPGAs suitable for space mission and still leave room for useful digital logic. The minimum configuration should aim at using less then 500 registers including all data storage needed to get the SpaceWire link up and running.

#### 3.1.2 Straightforward configuration parameters

In order to facilitate future implementations of the codec into projects, well defined and efficient configuration parameters are needed. The size of the data handling parts of the Rx and Tx regions as well as buffers and data storage in the *BusClk* region should ideally scale linearly with the bandwidth requirements for the specific application.

#### 3.1.3 Configurable transfer rates

Generics should make it possible to configure the SpaceWire Codec for everything from small light weight designs to full 10 bits received and transmitted every *BusClk*. The design and configuration should allow asynchronous data rates for Rx and Tx meaning that the design should be able to handle a combination of full and minimum speed in the same design.

### 3.1.4 Small BusClk asynchronous regions

The design should be such that the Rx- and Tx-regions contain a minimum of digital logic. This is important in order to keep power demands low as the receiver and the transmitter usually operate at a much higher frequency than the system clock.

### 3.1.5 Easy post synthesis placement

The design should allow for easy implementation into projects. In order to make the back-end work as streamlined as possible the design should only include the bare minimum of falling edge registers. Also the need for hand placing of logic or registers should be kept to a minimum.

### 3.1.6 Self calibrating timeout counters

The timeout counters, responsible for the disconnect timeout as well as the exchange timeout periods, should be self calibrating. This will make the design more self sustained and minimize the need for configuration pins and / or processor interference.

### 3.1.7 BusClk synchronous interface

Make sure all asynchronous interfaces over clock boundaries should be module internal to facilitate future implementations of the module into designs.

## 3.2 Non-formal demands

This section covers the non-formal demands applied to the design in order to make it fit company needs and make it suitable for future development.

### 3.2.1 Coding standard

Write the code using company coding standard as described in [RUAG CSTD].

### 3.2.2 Design partitioning

Partition the design in a tidy manner and make sure its modular enough to be able to be updated to future SpaceWire standards.

### 3.3 Requirements

The following subsections contain the formal requirements for the SpaceWire Codec. The requirements are divided into sections based on what function or part of the system they belong to. Each requirement is also written to target only one aspect and to be easily mapped into test scenarios.

The requirements in this document are numbered. The syntax is:

R/SWC.<increment> - <requirement heading>  
<requirement text>  
<requirement comment>

Where:

<increment> is an incremental number used within each group of requirements.

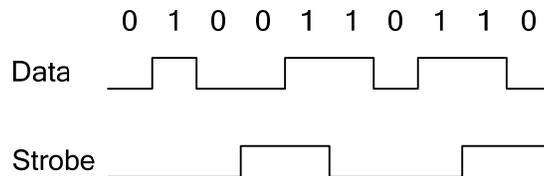
<requirement comment> Notes are not part of the formal requirement and should be considered as an extra explanation to the requirement or as a definition of the meaning of words within the requirement text.

If a requirement is no longer valid this will be indicated by the text "Deleted". Requirement numbers may not be removed or reused.

#### 3.3.1 Signal encoding and transition constraints

**R/SWC.005** - Signal encoding:

The SpaceWire Codec shall use Data-Strobe (DS) encoding as described in IEEE Standard 1355-1995. An example is included in Figure 6 below.



**Figure 6:** Data-Strobe example

**R/SWC.010** - Receiver simultaneous signal transition:

The receiver shall be fault tolerant during simultaneous transitions on both Data and Strobe input signals.

Note 1: Simultaneous transitions on a link that is up might force the link to go through the Link initialization procedure.

Note 2: Fault tolerant means that link errors are acceptable but no dead-lock should occur as a result of simultaneous transitions.

**R/SWC.015** - Transmitter simultaneous signal transition:

The transmitter shall not toggle both Data and Strobe signals during normal operation, this includes link down reset during the Link initialization procedure.

**R/SWC.016** - Transmitter duty cycle:

The duty cycle of the transmitted clock shall be 50-50 during contiguous operation.

**R/SWC.017** - Transmitter duty cycle at data rate shift:

During data rate shift, no half cycle of the transmitted clock shall be shorter than the half cycle of the faster data rate.

### 3.3.2 Data signal rate constraints

**R/SWC.020** - Minimum data signalling rate:

The minimum data signalling rate at which the SpaceWire shall operate is 2 Mbps.

Note 1: The minimum data signalling rate is set by the disconnect timeout 850 ns.

**R/SWC.025** - SpaceWire Codec maximum signalling rate:

The SpaceWire Codec shall be able to be configured to reach a maximum signalling rate of 10 bits received and transmitted every *BusClk*.

Note 1: The actual maximum signalling rate is also depending on the electrical characteristics of the link.

Note 2: The SpaceWire Codec should be capable of at least 200 Mbps throughput using MH1 and 400 Mbps using ATC18.

**R/SWC.120** – Link initialization signalling rate:

During link initialization the SpaceWire Codec shall be capable of sending and receiving characters at 10 (+/- 1) Mbps.

**R/SWC.030** - SpaceWire Codec operational signalling rate for receiver:

The SpaceWire shall be able to receive data at any rate between the minimum signalling rate and the maximum signalling rate.

Note 1: The maximum signalling rate is dependent upon configuration.

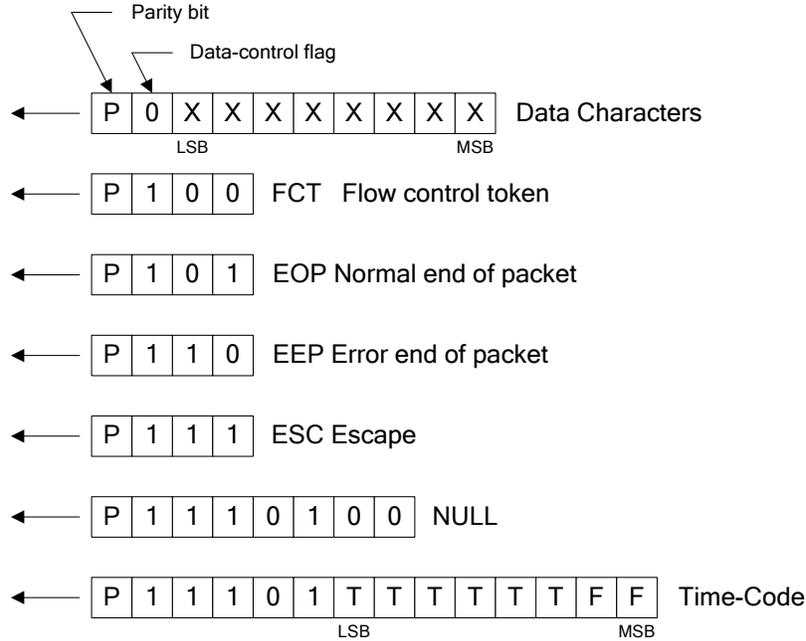
**R/SWC.035** - SpaceWire Codec operational signalling rate for transmitter:

The SpaceWire shall be able to transmit data at any rate between the minimum signalling rate and the maximum signalling rate given that the rate is the same as or an integer division of *SpwClk* \* 2.

### 3.3.3 Serial interface characters

**R/SWC.040 - Data and control characters:**

The data and control characters shall be as described in chapter 7.2 and 7.3 in [ECSS SPW]. The different characters are illustrated in Figure 7 below for reference.



**Figure 7: Data and control characters**

Note 1: NULL is made up of an ESC character followed by an FCT.

Note 2: Time-Code is made up of an ESC character followed by a Data Character.

**R/SWC.041 – Escape Error**

ESC followed by any other character then an FCT or a Data Character is considered a forbidden combination and shall cause an Escape error event.

**R/SWC.045 - Data and control character priority:**

The priority of the data and control characters shall be as described in section 8.3.n of [ECSS SPW]. The priority of the characters is included in Table 1 below for reference.

**Table 1: Priority table for character transmission**

Priority	Character
Highest	Time-Code
	FCT Flow control token
	N-Chars
Lowest	NULL

### 3.3.4 Link timing

**R/SWC.050** – Deriving timing time.

The SpaceWire Codec shall derive the disconnect timeout time of 850 ns as well as the Exchange timeout periods of 6.4 us and 12.8 us from the 10 Mbps transmitter clock during the Start-up cycle.

**R/SWC.055** – Disconnect timing:

The SpaceWire Codec shall assert the disconnect timeout event if the link has been quiet for more then 850 ns (between 727 ns and 1000 ns) after the last received bit.

**R/SWC.060** – Disconnect timing start:

The disconnect timer shall not start measuring the disconnect time until the first bit has been received.

**R/SWC.065** – Exchange timeout periods:

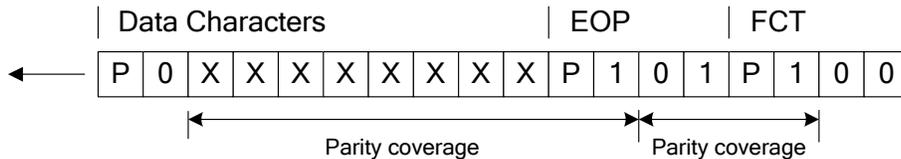
The nominal timeout period of 6.4 us shall be from 5.82 us to 7.22 us long and the 12.8 us timeout period between 11.64 us and 14.33 us long.

**R/SWC.070** – Deleted

### 3.3.5 Parity generation and error detection

**R/SWC.075** - Transmitter parity generation:

The transmitter shall generate a parity bit as described in 7.4 in [ECSS SPW]. The parity coverage is illustrated in Figure 8 below.



**Figure 8:** Parity coverage

Note 1: The parity bit is calculated over character boundaries as seen in Figure 8.

**R/SWC.080** - Receiver parity detection:

The receiver shall be capable of detecting parity errors.

**R/SWC.085** – Character validation:

A received character shall not be acted upon until the parity has been checked.

**R/SWC.090** - Parity error handling:

When the link receives a character with parity error the character associated with the parity error and all following characters until the link has reset and left the ErrorReset state shall be considered invalid.

### 3.3.6 Link status signals

**R/SWC.095** – RxErr:

RxErr shall be asserted when the following statement is true:

Parity error detected or

Escape error detected or

Disconnect timeout event detected.

**R/SWC.100** – gotFCT:

gotFCT shall be asserted when the SpaceWire Codec receives a valid FCT character.

**R/SWC.105** – gotN-Char:

gotN-Char shall be asserted when the SpaceWire Codec receives a valid normal character (N-Char).

**R/SWC.110** – gotTimeCode:

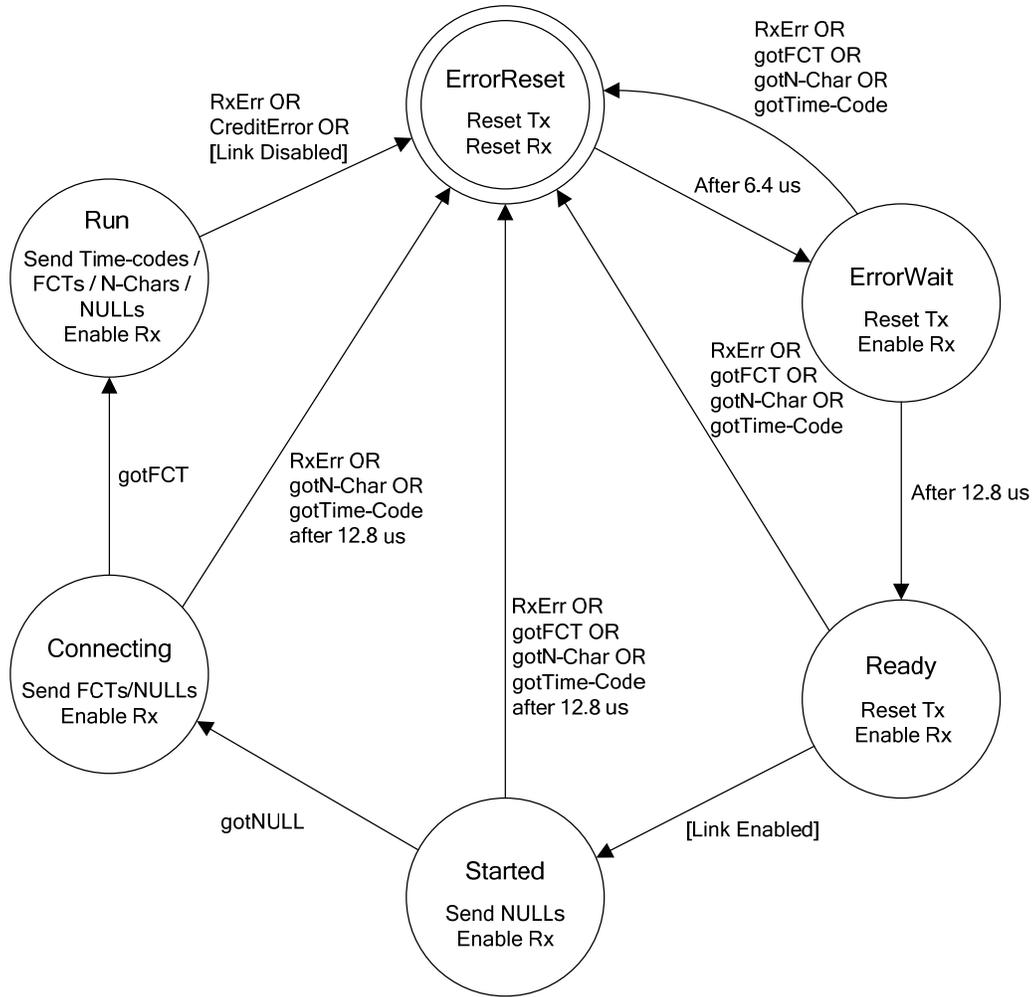
gotTimeCode shall be asserted when the SpaceWire Codec receives a valid TimeCode character.

**R/SWC.115** – CreditError:

CreditError shall be asserted when any of R/SWC.250 or R/SWC.270 is true.

### 3.3.7 Start-up procedure

All states of the SpaceWire Codec and all possible transitions are shown in Figure 9.



**Figure 9:** Start-up procedure and Main FSM

**R/SWC.125** – Link up in noisy environment:

All characters shall be ignored until the first valid NULL-token has been received. After receiving the first valid NULL-token the SpaceWire Codec will consider the link up.

Note 1: Noisy environment means that signal transitions might occur before the first valid NULL-character.

**R/SWC.130** – Link up error detection.

After link up, the SpaceWire Codec shall reset the link upon detecting an unexpected character or a parity error.

Note 1: If the prerequisites for more than one state change is fulfilled, change to ErrorReset take precedence.

**R/SWC.135** – Link start-up error:

The Link Start-up error shall be asserted if the following statement is true:

RxErr asserted or  
gotFCT asserted or  
gotN-Char asserted or  
gotTimeCode asserted.

Note 1: One NULL-character has to have been received.

**R/SWC.140** – States of the SpaceWire Codec:

The SpaceWire Codec shall contain the states: ErrorReset, ErrorWait, Ready, Started, Connecting and Run.

**R/SWC.145** – Receiver during Start-up procedure:

The receiver shall be Reset and disabled as long as the Main FSM is in the ErrorReset state and enabled in ErrorWait, Ready, Started, Connecting and Run.

**R/SWC.150** – Transmitter during Start-up procedure:

The transmitter shall be reset and disabled during ErrorReset, ErrorWait and Ready states and enabled during Started, Connecting and Run.

**R/SWC.151** – NULL-character first:

A NULL-character shall be the first character transmitted after the SpaceWire Codec have left the ErrorReset state.

**R/SWC.155** – ErrorReset proceed:

The SpaceWire Codec shall move to the ErrorWait state, unconditionally, after 6.4 us.

**R/SWC.160** – ErrorWait error:

The SpaceWire Codec shall move to ErrorReset state when a Link start-up error is detected.

**R/SWC.165** – ErrorWait proceed:

The SpaceWire Codec shall move to Ready state, unconditionally, after 12.8 us.

**R/SWC.170** – Ready error:

The SpaceWire Codec shall move to ErrorReset state when a Link start-up error is detected.

**R/SWC.175** – Ready proceed:

The SpaceWire Codec shall move to Started state if one or both of the following are true:

- *LinkDisabled* deasserted and *AutoStart* asserted and *GotNull*.
- *LinkDisabled* deasserted and *LinkStart* is asserted.

**R/SWC.180** – Started error:

The SpaceWire Codec shall move to ErrorReset state when a Link start-up error is detected.

**R/SWC.190** – Started time error:

The SpaceWire Codec shall move to ErrorReset state if it has not received a valid NULL-token for 12.8 us.

**R/SWC.195** – Started Tx:

The transmitter (Tx) shall send NULL-characters while in the Started state.

Note 1: the normal start-up bit rate of 10 Mbps is used.

**R/SWC.200** – Started proceed:

The SpaceWire Codec shall move to Connecting state on reception of a valid NULL-token.

**R/SWC.205** – Connecting error:

The SpaceWire Codec shall move to ErrorReset state when the following is true

- RxErr asserted or
- gotN-Char asserted or
- gotTimeCode asserted or
- not gotFCT after 12.8 us.

**R/SWC.210** – Connecting Tx:

The transmitter shall send FCT-characters and NULL-characters during Connecting state.

**R/SWC.215** – Connecting proceed:

The SpaceWire Codec shall move to Run state on reception of a valid FCT-character.

**R/SWC.220** – Run error:

The SpaceWire Codec shall move to ErrorReset state when the following is true

- RxErr asserted or
- CreditError asserted or
- LinkDisabled* asserted.

**R/SWC.225** – Run Tx:

The transmitter shall be able to send all types of valid characters when in Run state.

### 3.3.8 N-Char buffers

**R/SWC.230** – Receiver buffer size:

The receive buffer shall be able to hold between 8 and 56 N-Chars.

Note 1: The exact size of the receive buffer depends on RxFifoSize\_G.

**R/SWC.235** – Transmit buffer size:

The transmit buffer shall be able to hold between 1 and 56 N-Chars.

Note 1: The exact size of the transmit buffer depends on TxFifoSize\_G.

### 3.3.9 Flow control

A flow control token (FCT) indicate that the receive buffer can store eight more N-Chars and shall be transmitted when there is enough room reserved in the receive buffer.

**R/SWC.240** – Deleted.

**R/SWC.245** – Flow control generation:

The SpaceWire Codec shall send one FCT for every eight N-Chars of reserved space in the reception buffer after the link initialization procedure has reached the Connecting state to indicate that there is room for eight more N-Chars.

Note 1: When in Run state FCTs shall be generated as soon as the receive buffer has room another eight N-Chars.

Note 2: If for some reason the reception buffer does not have room for eight more N-Chars during the link initialization procedure, Null-characters shall be sent until the recipient resets the link.

**R/SWC.250** – Receive buffer overflow:

Credit error shall be asserted when the number of received N-Chars is greater then the number of outstanding N-Char requests.

Note 1: Outstanding N-Char requests equals to the number of FCTs transmitted minus the number of N-Chars received.

**R/SWC.255** – N-Char credit counter:

The SpaceWire Codec shall implement a credit counter to keep track of how many N-Chars it is allowed to transmit.

**R/SWC.260** – N-Char credit counter behaviour:

The credit counter shall increment its value by eight every time a FCT is received and decrement its value by one every time an N-Chars is transmitted.

**R/SWC.265** – N-Char credit counter maximum value.

The credit counter shall hold a maximum credit count of 56.

**R/SWC.270** – N-Char credit counter error:

Credit error shall be asserted if the credit counter goes above its maximum value.

**R/SWC.275** – N-Char credit count at zero.

The SpaceWire Codec shall seize transmitting N-Chars when the credit counter is zero.

**R/SWC.280** – Link up credit counter:

The credit counter shall be set to zero when the link is in ErrorReset state.

### 3.3.10 TimeCode interface

The time interface of the SpaceWire Codec shall comprise of two signals, *TickIn* and *TickOut*, a six-bit time output port *TimeOut*, a six-bit time input port *TimeIn*, a two-bit control flag input port *CtrlIn*, a two-bit control flag output port *CtrlOut*, the signal *WrTimeCode* to write new values to the *TickIn* and *CtrlIn* ports and the signal *ExtTimeCode*. The signals are defined in Table 7.

**R/SWC.285** – Deleted.

**R/SWC.290** – TimeCode generation:

When *TickIn* is asserted and the SpaceWire Codec is in the Run state a TimeCode shall be transmitted as soon as possible.

Note 1: The TimeCode shall have priority over all other types of characters but since there is an asynchronous interface in the signal path the exact timing can not be specified.

**R/SWC.295** – TimeCode update.

When the *WrTimeCode* is asserted the internal *TimeIn* and *CtrlIn* registers shall be updated with the values applied to the port.

**R/SWC.300** – TimeCode validity:

The received TimeCode shall be considered valid if the time is one more modulo 64 than the previously received TimeCode.

**R/SWC.305** – TimeCode reception:

*TickOut* shall be asserted when the SpaceWire Codec is in the Run state and a valid TimeCode is received.

**R/SWC.310** – Extended TimeCode:

The two control flags shall be set to/checked against zero when *ExtTimeCode* is deasserted and be propagated when *ExtTimeCode* is asserted.

### 3.3.11 Data and control interface

#### **R/SWC.315** – Control interface

The following signals are provided to control the overall functionality of the SpaceWire codec.

**Table 2:** Control interface

Signal	Direction	Description
BusClk	In	System Clock
SpwClk	In	SpaceWire transmit Clock
TxCkDiv	In	Down conversion rate for TxClk
Reset	In	Reset signal
ResetEnd	In	Reset End signal
LinkStart	In	Flag indicating that the link is ready to start
LinkDisable	In	Flag indicating that the link is disabled
AutoStart	In	Flag indicating that the link should start on reception of a NULL Character

#### **R/SWC.315** – BusClk asynchronous data reception interface

The following signals are used to receive data.

**Table 3:** RxLink interface

Signal	Direction	Description
DIn	In	Data signal to Receiver Block
SIn	In	Strobe signal to Receiver Block

#### **R/SWC.320** – BusClk synchronous data reception interface

The following signals are used to receive data.

**Table 4:** RxData interface

Signal	Direction	Description
RxData	Out	Byte wide data from Rx FIFO
RxDValid	Out	RxData valid
RxDAck	In	Ack the data currently at RxData
RxPkt	Out	Deasserted at packet end
RxErr	Out	Asserted at packet end when failure

**R/SWC.325** – BusClk asynchronous data transmit interface

The following signals are used to transmit data over the SpaceWire link.

**Table 5:** TxLink interface

Signal	Direction	Description
DOut	Out	Data signal from Transmitter Block
SOut	Out	Strobe signal from Transmitter Block

**R/SWC.330** – BusClk synchronous data transmit interface

The following signals are used to transmit data.

**Table 6:** TxData interface

Signal	Direction	Description
TxData	In	Byte wide data to Tx FIFO
TxDValid	In	TxData valid
TxDAck	Out	Ack the data currently at TxData
TxPkt	In	Deasserted to indicate packet end
TxErr	In	Asserted at packet end to indicate failure

**R/SWC.335** – TimeCode interface signals

The following signals are used to transmit data.

**Table 7: Time-Code interface**

Signal	Direction	Description
WrTimeCode	In	Updates the value for <i>CtrlIn</i> and <i>TimeIn</i>
TickIn	In	Signal from time-master to send TimeCode
CtrlIn	In	The two MSB of the TimeCode
TimeIn	In	Time-Code to be transmitted
TickOut	Out	Asserted when a valid Time-Code is received
CtrlOut	Out	The last received Ctrl-field of the TimeCode
TimeOut	Out	The last received Time-Code

**R/SWC.335** – Deleted.

## 4 Functional design

This chapter includes the functional design of the SpaceWire Codec designed during this thesis. The chapter will start with a justification of the design partitioning developed during this theses followed by an overview of the proposed design. The next subsection gives an explanation of the data interface and the configuration parameters. The reminder of this chapter will give an in depth description of the design where each of the functional blocks will be described in detailed, with one exception, the Control Logic block. This part of the design will be mentioned in the context where it is deemed fit. This is done as the Control Logic is not confined in its entirety to one VHDL file but rather distributed amongst some of the files making up the rest of the design.

### 4.1 Design partitioning

This subchapter will discuss the usual partitioning of a SpaceWire codec and compare it to the design proposed in this thesis.

#### 4.1.1 Codec as described in the SpaceWire standard

The design solution for the SpaceWire codec as described in [ECSS SPW] have only two clock regions, one being the *RxClk* region synchronous to the incoming Data-Strobe signals, and the other being the *SpwClk* region, see [ECSS SPW] §8.4 for details. This approach has a few drawbacks:

- It makes the implementation for each new application cumbersome as an asynchronous interface between the *SpwClk* and the *BusClk* regions needs to be designed for each new implementation.
- This also means that a large part of the design is residing in the fast *SpwClk* region increasing power needs and putting unnecessary demands on the synthesis and at worst, limiting performance due to hard timing constraints.
- Increase size of the design as it is harder to make a lean asynchronous interface further away from the receiver and the transmitter.

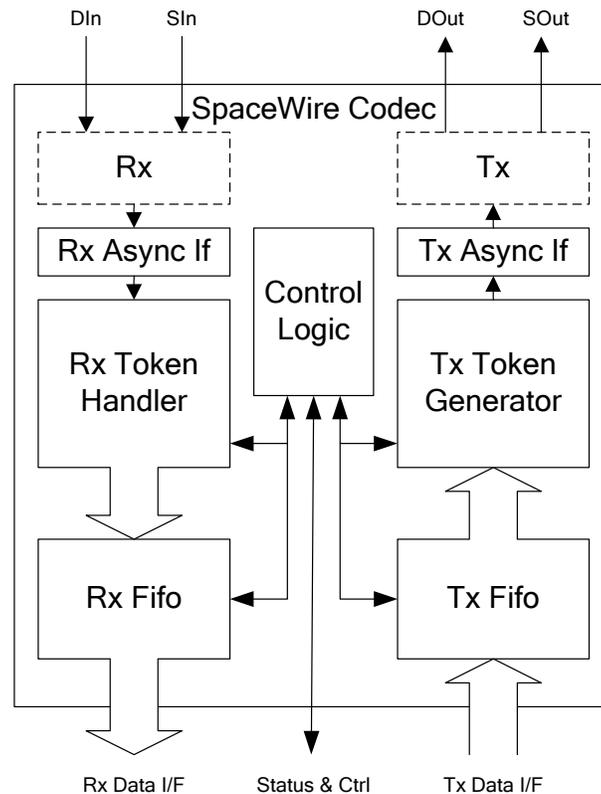
There are also benefits with this approach:

- If *SpwClk* and *BusClk* are synchronous, there is no need for an asynchronous interface making the design use less registers.
- You are free to implement the data interface in any way you see fit, making the codec suitable for a wider range of applications.

## 4.1.2 The proposed design

The design solution proposed during this thesis differs from the one suggested in [ECSS SPW] on a number of things. The main points are:

- Very small Rx and Tx regions with slim to no complex logic inside. All intelligent decisions are made in the system clock region in order to enhance receiver / transmitter performance while keeping power consumption down.
- Internal data buffers capable of storing enough N-Chars to keep the SpaceWire Codec running stutter free. A side effect of this is that all flow control is handled within the SpaceWire Codec, removing the need to control this externally.
- BusClk synchronous interfaces. The benefit of having the system interface in the system clock region is that no extra driver or asynchronous interfaces are needed outside of the SpaceWire Codec.
- Self-calibrating timeout timers. The length of the timeout times is generated internally in order to reduce the number of configuration pins and / or register operations needed.



**Figure 10:** SpaceWire overview

The SpaceWire Codec seen in Figure 10 contains every block needed for the link to start up and provide a BusClk synchronous data and control interface to the rest of the design, see §4.3 for details. Each block is configurable in size except the Control Logic. For details on how to configure the size and speed of the SpaceWire Codec, please refer to the subchapters of §4.4 for description or guidelines.

## 4.2 Overview

The overview of the SpaceWire Codec seen in Figure 10 can be mapped, to some extent, to the general description of a SpaceWire codec as seen in Figure 4. The overview in Figure 10 is more detailed as it depicts the functional blocks of the design rather than the function of the codec. Also note that the direction of the data flow in the pictures is reversed.

### 4.2.1 Function

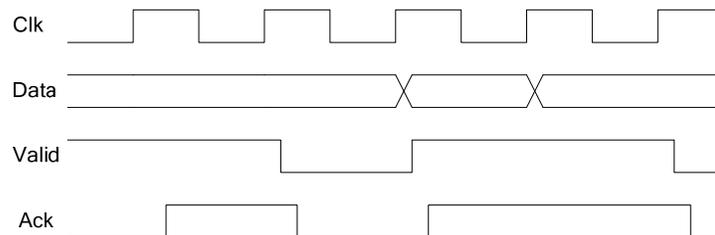
To further explain the function of the SpaceWire Codec seen in Figure 10 a short description will follow.

The receiver pipeline in the leftmost part of Figure 10, containing all functional blocks starting with Rx, is responsible for receiving all N- and L-Characters on the link during normal operation. Inside the dashed line, in Rx-region, the Data-Strobe signals generates *RxClk* that is used to clock in all incoming data. There is one bit received every rising and falling edge of *RxClk*, Double Data Rate (DDR), meaning that a 100 MHz clock carries data at a rate of 200 Mbps. The bits received in Rx will be made *BusClk* synchronous in the Rx Async IF, decoded in the Rx Token Handler and finally stored in the Rx FIFO before being handed over to the system. Note that only the N-Characters are handed over to the system level and that all L-Characters are handled internally by the Control Logic.

The transmitter pipeline in the rightmost part of Figure 10, containing all functional blocks starting with Tx. The Tx-pipeline will get all N-Characters from the system level, via the Tx FIFO, and all L-Chars from the Control Logic. The Tx Token generator is responsible for preparing the characters, before being handed over to the Tx Async IF. The Tx Async IF-block is responsible for taking the *BusClk* synchronous signals and making them *SpwClk* synchronous. Inside the dashed line, in the Tx-region, all signals are *SpwClk* synchronous. The data-rate of the outgoing traffic can be divided down by setting the *TxClkDiv* in Table 2, creating a fictive clock signal, the *TxClk*. The Data-Strobe signal transmitted over the link is DDR, meaning that one bit is transmitted each rising and falling edge of *TxClk*.

## 4.3 Data Interface

This section covers all data IO in the SpaceWire Codec entity.



**Figure 11:** BusClk synchronous data interface

In order for the SpaceWire Codec to be able to send as well as receive one byte of data every *BusClk* cycle, the *RxDValid* and the *TxDValid* as well as the *RxDAck* and the *TxDAck* need to keep up with the transmission at full speed.

The signals associated with the RxData I/F and the TxData I/F are designed to be able to communicate without the use of glue logic, meaning that the receiver data interface signals are able to drive the transmitter data interface signals when connected back to back in a router type design.

## 4.4 Configuration

The following chapters describe the configuration parameters needed in order to setup a working synthesis of the SpaceWire Codec with a certain performance. The chapter starts by introducing the different configuration parameters in Table 8 followed by an in depth description of each parameter, what it configures and how to calculate the correct value for a given  $RxClk / BusClk$  ratio.

**Table 8:** Configuration table

Generic	Description
DataSampleGroups_G	The width in bit-pairs of the FIFO transporting bits across Rx Async IF
RxAsyncIfFifo_G	The width in bit-pairs of the received data that can be concatenated each $BusClk$
CharacterBuffers_G	The number of characters that can be received each $BusClk$
RxFifoSize_G	The number of bytes that can be stored in the Rx FIFO, needs to be larger then eight bytes.
DataTransmitGroups_G	The width (in 9 bit characters) of the Tx Async IF
TxFifoSize_G	The number of bytes that can be stored in the Tx FIFO

### 4.4.1 Data sample groups

DataSampleGroups\_G determines the number of bit pairs in the Rx reception buffer as well as the width of the Rx Async IF in Figure 10. The number of bit pairs needed in the asynchronous interface depends on the  $RxClk / BusClk$  ratio. Remember to use the maximum  $RxClk$  value as it can shift up to 10%.

The minimum value is calculated as:

$$DataSampleGroups\_G \geq (RxClk / BusClk) * 2 + 2$$

### 4.4.2 Rx Async IF FIFO

RxAsyncIfFifo\_G determine the maximum number of bit pairs that is allowed in FIFO that concatenates the incoming data from the Data sample groups to form whole characters. The number of bits needed is the length of one data character minus two plus the maximum number of new bytes per  $BusClk$  plus two. Remember to use the maximum  $RxClk$  as it can shift up to 10%.

The minimum value is calculated as:

$$\text{RxAsyncIfFifo\_G} \geq (8 + (\text{RxClk} / \text{BusClk}) * 2 + 2)$$

#### 4.4.3 Character Buffer

CharacterBuffers\_G determines the number of Character Buffers needed to handle the incoming characters from the Rx Async IF FIFO in Figure 10 each *BusClk*. The number of Character Buffers needed is one plus the maximum number of Characters in one *BusClk*. Remember to use the maximum *RxClk* as it can shift up to 10%.

The minimum value is calculated as:

$$\text{CharacterBuffers\_G} \geq 1 + ((\text{RxClk} / \text{BusClk}) * 2 - 2) / 4$$

#### 4.4.4 Rx FIFO Size

RxFifoSize\_G determines the number of bytes that can be stored in the Rx FIFO seen in Figure 10. RxFifoSize\_G can be any value between 8 and 56 but the value should be chosen large enough to make the reception of N-Chars go smoothly. The higher the *RxClk* / *BusClk* ratio the larger the buffer needs to be.

The minimum practical limit is 10 but the optimal value for a given *RxClk* / *BusClk* ratio also depends on the transmission rate as a FCT needs to be transported over the link.

#### 4.4.5 Data transmit groups

DataTransmitGroups\_G determines the width of the Tx Async IF in Figure 10. The configuration value of DataTransmitGroups\_G is determined by the number of bits needed to supply the transmitter with data, EOP and EEP characters. During transmission the bulk of the transmitted data is Data Characters so each Data Transmit Group can be counted as 10 bits.

There is no absolute limit for this configuration parameter but in order for the transmission to go stutter free, meaning no unnecessary NULL characters transmitted during data transmission, there needs to be one Data transmit group for every 2.5 bits transmitted every *BusClk*.

#### 4.4.6 Tx FIFO Size

TxFifoSize\_G determines the number of bytes that can be stored in the Tx FIFO as seen in Figure 10. The size of the Tx FIFO does not depend directly on the rate of transmission but rather on the rate and/or chunk size of data reception via the Tx Data interface.

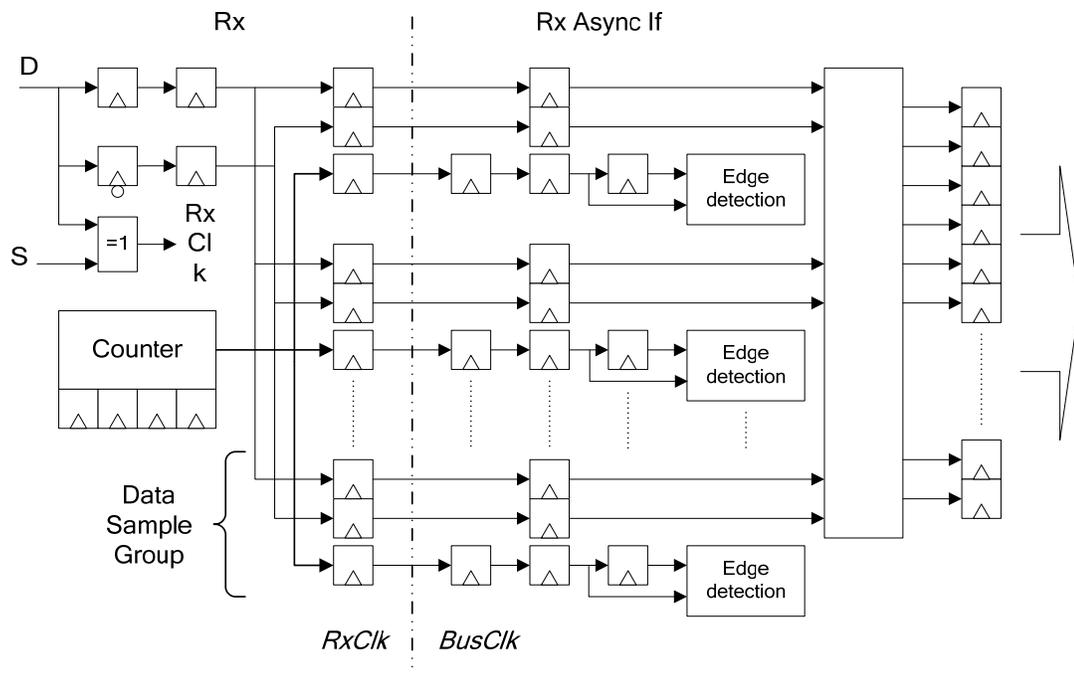
There are no easy ways to calculate the optimal value of TxFifoSize\_G but a recommendation is to set the value to at least one more than the value of DataTransmitGroups\_G.

## 4.5 Rx Pipeline

The following sub chapters will give an in depth description of the Rx pipeline as seen in Figure 10 starting at the DS-signals and following the data flow down to the data interface at system level.

### 4.5.1 Receiver

The SpaceWire receiver, depicted as Rx in Figure 10, receives both data and strobe from the transmitter at the other end of the link. The detailed description of the Receiver and the supporting asynchronous interface is shown in Figure 12 below. The design has been made especially to facilitate place and route as well as to make the design as configurable and as cost effective in gates as possible.



**Figure 12:** Receiver and asynchronous interface

To further explain Figure 12:

$RxClk$  is generated as  $D_{in} \text{ xor } S_{in}$  and all registers in the Rx region, the region to the left of the dash-dotted line, are triggered by  $RxClk$ . Since the D-signal is used to generate  $RxClk$  as well as carries the data content of the incoming bit-stream, the timing between the D-signal and the generated  $RxClk$  needs to be adjusted post synthesis.

There is only one time critical falling edge DFF in the proposed design for the receiver, the DFF that samples incoming falling edge data. This helps to keep demands on place and route low and makes it possible to reach quite high data rates. The other falling edge DFF in the receiver is the LSB of the Counter but since all incoming data is handled in pairs, this DFF is not used for anything time critical.

The Counter in Figure 12 keeps track of the number of incoming data bit pairs and enables the correct Data Sample Group to buffer the incoming data and alert the Rx Async IF about arriving bit-pairs. The Enable signal from the Counter and a detailed description of the Data Sample Group is shown in Figure 13 below.

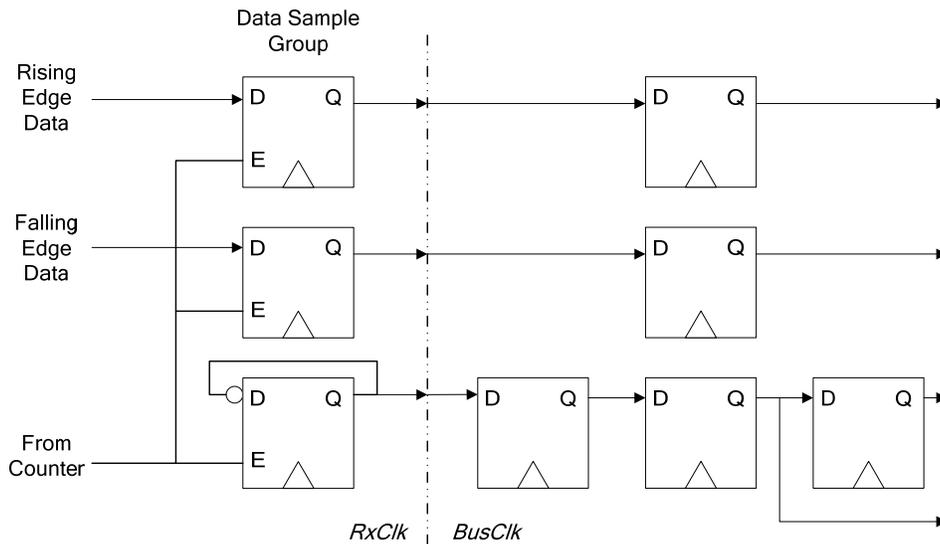
The SpaceWire Codec has a configurable amount of Data Sample Groups, using `DataSampleGroups_G`, in order to optimize size for the desired performance. See 4.4.5 for configuration details.

Once the Edge detection is asserted in the `BusClk` region all new data in the Rx Async IF DFFs will be moved and concatenated in the Rx Async IF FIFO in the rightmost part of Figure 12.

The SpaceWire Codec has a configurable Rx Async IF FIFO using `RxAsyncIfFifo_G`. See 4.4.4 for configuration details.

### 4.5.2 Rx asynchronous interface

This section will cover the design of the asynchronous interface for one Data Sample Group in detail followed by a short description about the asynchronous interface for the Counter as well as the reset signal for the Rx region.



**Figure 13:** Rx Asynchronous interface in detail

The details in Figure 13 can be seen as the structure marked Data Sample Group in Figure 12. A description follows below:

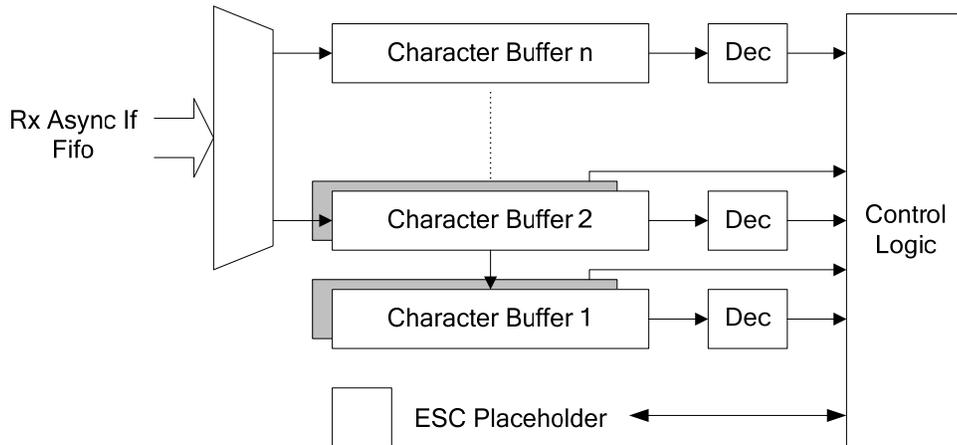
Once the Enable signal from the counter points at the Data Sample Group, Rising edge and Falling edge data will be sampled on the next positive *RxClk* transition. At the same time the Data Valid signal, shown at the bottom of Figure 13 toggles. The Data Valid signal is propagated into the BusClk region and is used to trigger the Edge Detection, shown in Figure 12. The Edge Detection will let the supporting circuits of the Rx Async IF FIFO know that new data can be sampled in and concatenated to the bit-stream. The solution for Rx Async IF as seen above does have an overhead in terms of DFFs and logic compared to other solutions but was chosen as it's a fast, configurable and robust way of crossing a clock boundary.

The LSB of the Counter in Figure 12 have a standard asynchronous protocol #1 interface as described in [RUAG ASYNC]. The Counter together with the Edge Detection circuit is used to reset the Disconnect timeout timer every time a bit has been received.

The asynchronous interface of the Reset-signal for the Rx-region is a standard asynchronous protocol #6 interface as described in [RUAG ASYNC]. All DFFs, except the ones that are used to sample DIn and SIn, use asynchronous reset since *RxClk* might be absent. The DFFs that are used to sample DIn and SIn are not reset to improve link timing.

### 4.5.3 Rx token handler

Once the incoming data is concatenated in the Rx Async IF FIFO the data needs to be decoded, split up into characters and moved to the Rx Token Handler shown in Figure 10. Once the data reaches the Rx Token handler, as shown below, a parity check is performed together with character decoding to be able to recognize, sort and signal specific commands to the Control Logic shown in Figure 10. The Token handler is shown in the figure below.



**Figure 14: Rx Token handler in detail**

The different sized characters will be recognized by peeping at the data-control flag. Every full sized character found in the Rx Async IF FIFO is moved to the Character Buffers labelled 2 to n seen in Figure 14. The last Character is always moved to Character Buffer 1 at the bottom of Figure 14 so that the parity check can be performed before the Character is allowed to have any effect on the system. The parity check is shown in grey in Figure 14. The ESC Placeholder is asserted when the last Character to pass the Parity check is an Escape Character. The ESC Placeholder is only there to minimize DFFs.

The number of Character Buffers is configurable using CharacterBuffers\_G, see 4.4.4 for configuration details.

After parity check and detection, the characters will either be moved to the Rx FIFO seen in Figure 10 if the character is an N-Char or signal the Control Logic if the character is an L-Char.

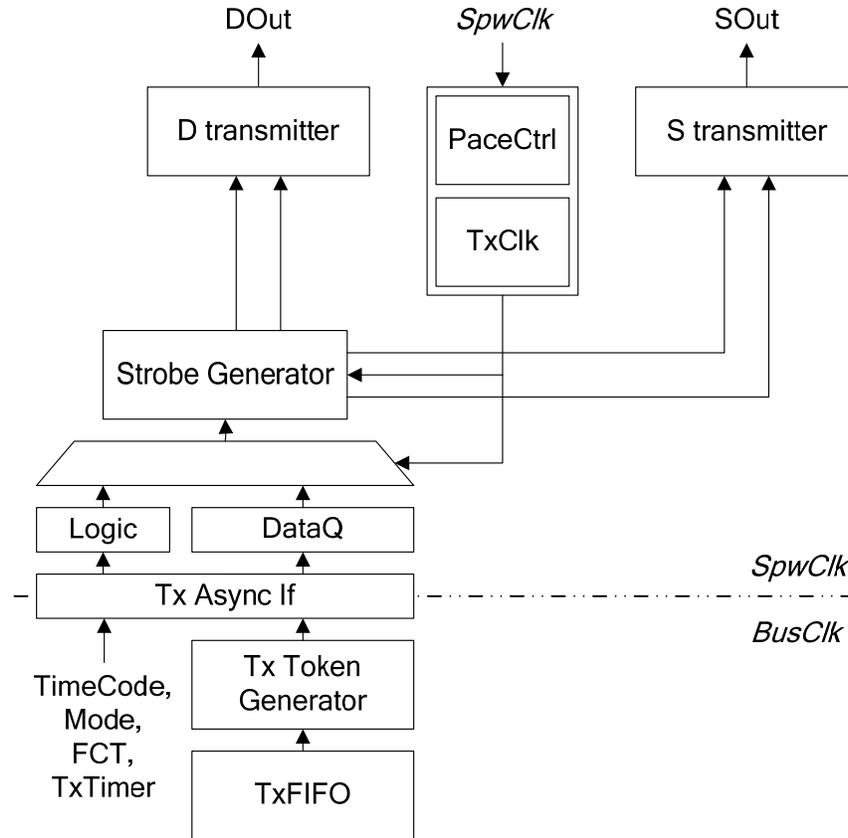
#### 4.5.4 Rx FIFO

The Rx FIFO is configurable, using RxFifoSize\_G, in 1 byte blocks to give the designer the ability to optimize for size or performance. See 4.4.4 for configuration details. The Control Logic is responsible for keeping track of the read and write counters for the Rx FIFO as well as the sending of FCTs.

## 4.6 Tx Pipeline

The following sub chapters will give an in depth description of the Tx pipeline as seen in Figure 10. The description will start with an overview of the transmitter topology. After that the description will follow the flow of data from the BusClk synchronous data interface to the DS-drivers in the transmitter.

The transmitter pipeline within Figure 10 can be viewed in detail in the figure below together with the Tx asynchronous interface and associated signals.



**Figure 15:** Transmitter overview

*SpwClk* is divided down to the rate indicated by *TxClkDiv*, creating the fictive clock signal *TxClk*. The *TxClk* is used to drive the flow of data inside the Tx-region.

As seen in Figure 15 the transmitter consists of two identical transmitters, one for Data and one for the Strobe. The counter directs bit pairs from The D FIFO or the Logic generating the NULL and Time-Code Characters through the Strobe Generator.

### 4.6.1 Timer

In order to have self calibrating timeout timers as demanded in 3.1.6 it is necessary to calculate the number of BusClk cycles that make up the 6.4 us timeout time. From those 6.4 us the SpaceWire Codec can derive the 850 ns disconnect timeout time as well as the 12.8 us time.

The reference for measuring the time is the generated 5 MHz TxClk that is used during the Link initialization procedure. In order to get a correct count of the time, a start measure time pulse is first synchronised over to the *SpwClk* region where a counter returns a stop measure time pulse after 29 RxClk cycles, 32 RxClk equals 6400 ns - 2 RxClk for the start pulse asynchronous interface and -1 for the time activating the stop pulse.

In the BusClk region a counter is measuring time in parallel from the time that the start measuring time pulse is activated, until it receives the stop measuring time pulse. The value is then deducted by 2 to compensate for the asynchronous interface from *SpwClk* to *BusClk*.

### 4.6.2 Tx FIFO

The size of the Tx FIFO is configurable, in one byte blocks, using TxFifoSize\_G. This enables the hardware designer instantiating the SpaceWire Codec the means to tailor the circuit for optimal performance / size. For details on how to configure, see 4.4.6 for details. The Tx FIFO is responsible for the data interface of the Tx-pipeline and is capable of receiving one byte per BusClk during packet transmissions.

### 4.6.3 Tx token generator

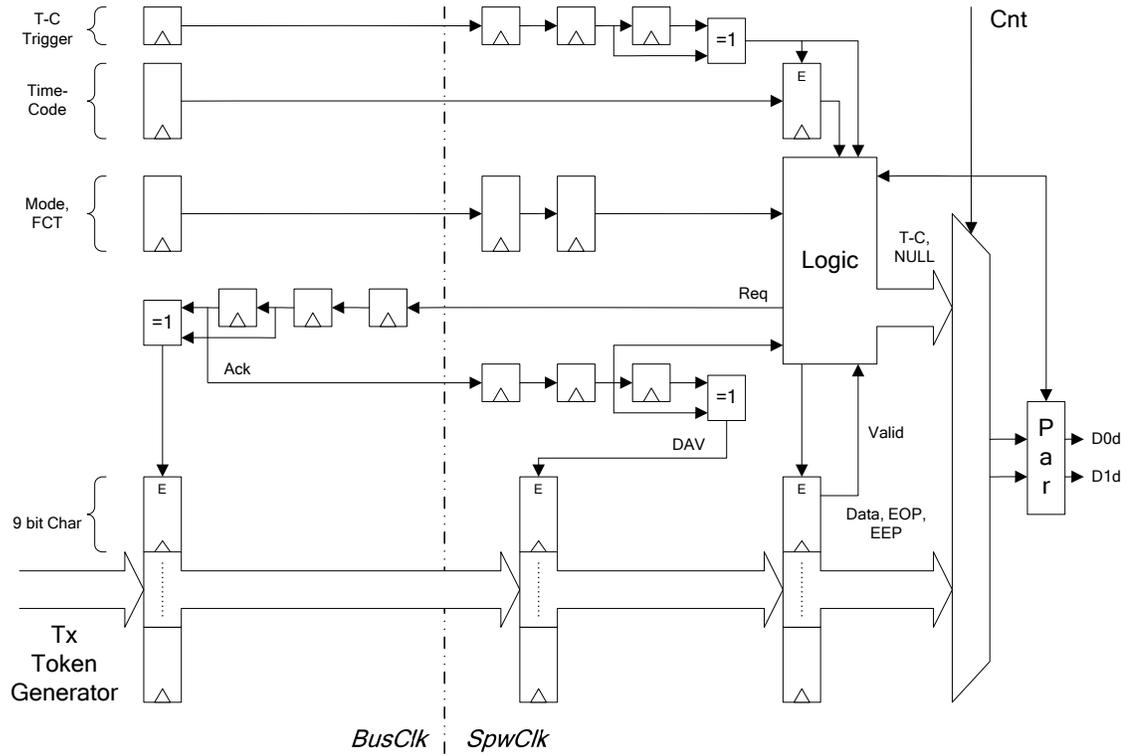
The Tx Token Generator is responsible for serving the transmitter with data characters as well as for keeping track of how many more N-characters the SpaceWire Codec is allowed to transmit. The number of N-characters that the SpaceWire Codec is allowed to transmit is increased every time that one or more FCTs are received and decreased every time that the Tx Token Generator makes one N-Char available to the transmitter. These N-chars are taken from the Tx FIFO, concatenated and made available to the Tx Async IF when it is ready to receive more.

### 4.6.4 Tx asynchronous interface

As seen in Figure 15 there are four types of signals flowing over the *BusClk* / *SpwClk* boundary, these are:

- Time-Code.
- FCT
- Data, EEP and EOP
- Mode Ctrl

The Time-Code needs special attention due to the time-critical response once a TICK-IN has been detected. Data, EOP and EEP-characters are made available to the *SpwClk* region through the Tx Token Generator. Mode Ctrl sets the Tx internal mode so that the state of the SpaceWire Codec have full control over start up /shut down- procedures as well as the automatic generation of NULLs when the transmitter is idle. A detailed view of the Tx Asynchronous IF is shown below.



**Figure 16:** Tx Asynchronous interface and supporting circuits

The T-C Trigger seen in Figure 16 is a toggling signal that determines when to sample a new Time-Code and generate a Time-Code Character. Since Time-Codes have priority, special attention has been made to make sure that it is fast enough.

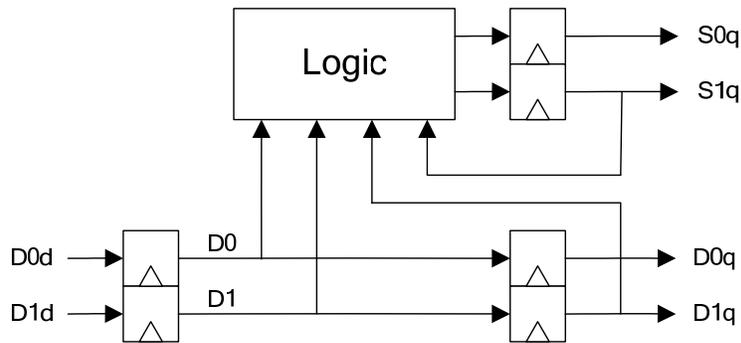
The Mode lets the Logic within the Tx-region know when the SpaceWire link is enabled and the FCT part is a toggling signal that lets the transmitter know when to generate a new FCT-character.

All data-characters as well as all EOP and EEP characters, transmitted via the Tx Token Generator into the Tx-region, are 9 bit vectors plus a valid flag. The valid flag indicates that the 9 bit vector contains valid information to be transmitted when possible. The 9 bit vector contains the data/control-bit and 8 more bits for data or control-character information, see 4.4.5 for configuration details.

The parity-bit is generated inside the Par-block in Figure 16. The Parity-block is allowed to update and proceed at rising edge of TxClk.

#### 4.6.5 Data strobe generator

The Strobe Generator-block seen in Figure 15 is shown in detail below. All DFFs in the Strobe Generator is allowed to update on rising edge of TxClk.



**Figure 17:** Data strobe generator

The logic inside Figure 17 consists of:

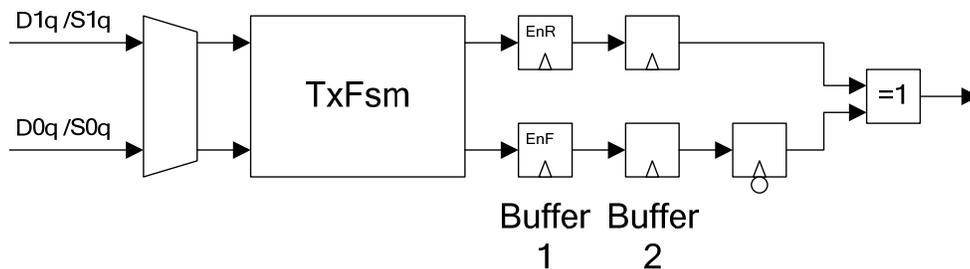
$$S0q \leq (D1q \text{ xor } D0) \text{ xor } S1q$$

$$S1q \leq (D0 \text{ xor } D1) \text{ xor } S0$$

It is possible to disable the automatic strobe generation via the Mode control interface. Each of the S- and D-pairs is connected to a Transmitter, shown in detail in the next subsection.

#### 4.6.6 Tx driver

The Transmitter make use of two identical internal transmitter channels, one for S and one for the D that is making up the bit stream. The active internal structure of these channels can be viewed in Figure 18 below.



**Figure 18:** Transmitter drive stage

The upper path through Buffer 1 and Buffer 2 is for bits that are to be transmitted on rising edge of TxClk. This means MSB of the bit pair on uneven multiples of  $TxClk / SpwClk$  or both bits on even multiples of  $TxClk / SpwClk$ . The lower path carries only the LSB of the bit pair during uneven multiples of  $TxClk / SpwClk$ .

The propagation speed is set by the enable signals EnR (enable rising) and EnF (enable falling). TxR (TxClk rising edge) is responsible for propagating the bit pairs from the mux to the left of the PAR-block in Figure 16 and through the Strobe Generator.

## 5 Verification

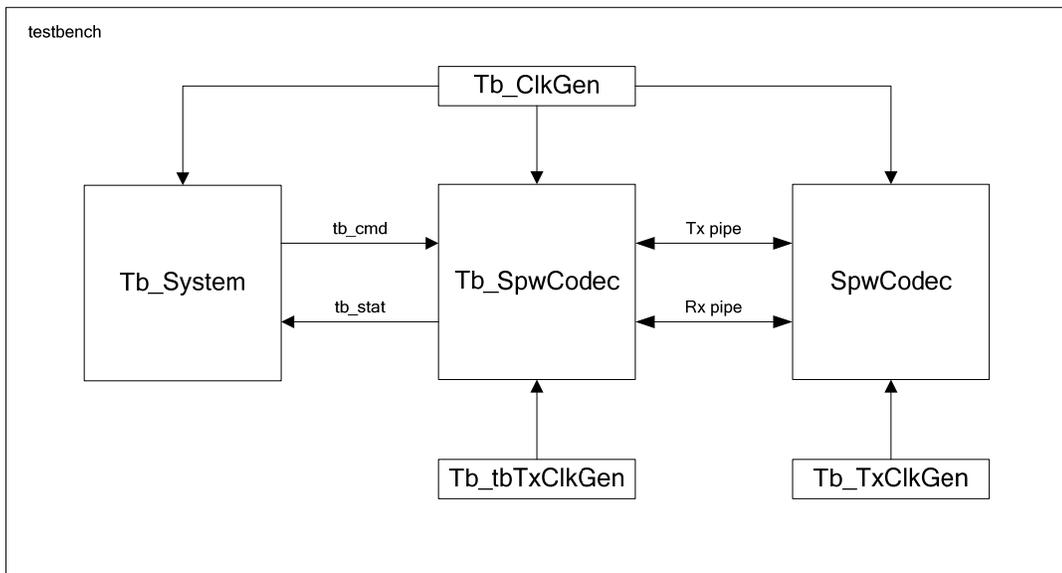
This chapter explains the verification procedure, via simulation as well as on hardware. The verification step performed here is not enough to make the module flight ready but they cover enough to make sure that the modules main functionality works correctly as well as gives the foundation for future more complete tests.

### 5.1 Simulation

This chapter will start with an in depth description of the test-bench modules followed by a functional and functionality description of the test-bench including its capabilities, fault and error detection mechanisms and the errors its capable of generating in order to give stimuli to the SpaceWire Codec. Finally the test procedure will be explained together with the desired outcome and what requirements the test is suppose to cover.

#### 5.1.1 Test bench description

The test-bench concept is described below. The SpaceWire Codec, the module under test, is connected to Tb\_SpwCodec. Tb\_SpwCodec is responsible for the generation of stimuli to the SpaceWire Codec as well as sample all signals of interest.



**Figure 19:** Test bench overview

All clock inputs have their own clock driver in order to give full control of the timing and the speed of every clock pulse to be able to simulate a proper asynchronous system.

Every data and control input / output of the SpaceWire Codec is connected to the tb\_SpwCodec. The tb\_SpwCodec module works as a driver for the SpwCodec and handles all signal transitions as well as monitors all activity of the SpwCodec module. The tb\_SpwCodec is commanded by the tb\_System module where all test procedure are executed. All communication between tb\_SpwCodec and tb\_System is handles via the Tb\_SpwCodecCmd and the Tb\_SpwCodecStat signals.

Tb\_SpwCodecCmd is a record containing data to the tx-pipe of both Tb\_SpwCodec and SpwCodec as well as TimeCode value, a bit-vector and an unspecified integer used for many of the operations performed by Tb\_SpwCodec. Tb\_SpwCodec is commanded to perform actions by the Command signal in Tb\_SpwCodecCmd and a short description of the most important of them is included in Table 9.

### 5.1.2 Test bench commands

At present there are 55 different commands performed by Tb\_SpwCodec and the table below covers the most important ones. The E / D column depicts Enable / Disable capabilities for the command.

**Table 9:** Commands to tb\_SpwCodec

Command name	E / D	Clarification
ResetTbSpw		Resets the Tb_SpwCodec.
ResetAllConfig		Undo all configuration without resetting counters etc.
ResetAllCnt		Resets all counters without resetting the configuration.
ResetSpw		Resets the SpwCodec.
SetTxClkDivNomTbSpw		Set the start-up nominal clock division rate for Tb_SpwCodec.
SetTxClkDivMaxTbSpw		Set the run clock division rate for Tb_SpwCodec.
ManualModeTbSpw	E / D	Toggle manual mode for TbSpw.
AutoStartTbSpw	E / D	Toggle auto start functionality for Tb_SpwCodec.
RxTbSpw	E / D	Disables or re-enables the receiver in Tb_SpwCodec.
TxTbSpw	E / D	Disables or re-enables the transmitter in Tb_SpwCodec
TxBabble	E / D	Toggles the ability to send predefined noise to the SpwCodec during a phase that it should not be possible to transmit.
SendBit		Toggles the strobe signal from Tb_SpwCodec.
SendBitStream		Transmit one predefined bit-stream to SpwCodec.
BitStreams	E / D	Transmit or stops transmitting a contiguous bit-stream to SpwCodec.
SendNull		Transmit a Null character to the SpwCodec
SendNullParErr		Transmit a Null character with parity error to SpwCodec.
SendFct		Transmit one FCT character to the SpwCodec.
SendFctParErr		Transmit an FCT character with parity error to SpwCodec.
Fct	E / D	Disables or re-enables the automatic transmission of

		FCT characters from Tb_SpwCodec.
SendTbData		Adds the data stored in TbData to the Tb_SpwCodec send queue.
ReadTbData		Read one entry of the data received by Tb_SpwCodec, if any.
SendTb Eop / Eep		Transmit one End of Packet or Error end of Packet char to SpwCodec.
SendTbEsc		Transmit one Esc character.
SendTbEscParErr		Transmit one Esc character with parity error.
SendTbTimeCode		Transmit one time code character.
SendTbTimeCodeParErr		Transmit one time code character with parity error.
ResetSpw		Reset the SpaceWire Codec under test.
EnableLinkStartSpw		Enables link start-up for the SpaceWire Codec under test.

## 5.2 Test bench receiver capabilities

This section covers the receiver part of the SpaceWire test bench and its mechanisms to verify the functionality of the SpaceWire Codec transmitter. The test bench receiver aims at verifying that the SpaceWire transmitter always works in a predictable manner and in accordance with the specification. To guaranty this, the receiver part of the test bench checks incoming characters as well as pulse lengths and start-up procedures. The fault detection mechanisms and the latent monitors of the SpaceWire test bench are described in detail below.

### 5.2.1 Monitors

Monitors are used for autonomous control of certain functions. The two most important monitors are:

- ReceiveBpsMoni     The receive bit per second monitor controls that the transmitted clock have the correct duty cycle and that no glitches appear.
- ParityErrMoni     The parity error monitor controls that the parity bit of the transmitted characters is correct.
- CreditErrorMoni     The credit error monitor controls that the SpaceWire Codec does not send more N-Chars than it is allowed to.

## 5.3 Test bench transmitter capabilities

The transmitter in tb\_SpwCodec is capable of transmitting any sequence of bits. Most of the commands in Table 9 are dedicated to the control of characters or bit patterns transmitted to the SpaceWire Codec. See the test procedures below for more information.

## 5.4 Test procedures

This section covers the test procedures used to verify the functionality of the SpaceWire Codec. The procedures are not enough to verify the module for flight operation but give enough confidence that the Codec as a whole performs within specification and that the concept works. Further the test-bench infrastructure developed during the thesis work gives future test developers a solid foundation on which to build a larger more complete test suite qualifying the SpaceWire Codec for flight missions.

### 5.4.1 Start-up test

The Start-up test aims at verifying the start-up procedure of the SpaceWire Codec. This is done by measuring the timing of the SpaceWire Codec's state transitions as it is subjected to a variety of different scenarios. The test also checks the codec's sensitivity to stimuli, both changes in configuration and input signals via the receiver. There are a total of five scenarios in the Start-up test.

1. Normal start-up, start the SpaceWire Codec in auto start mode and measure the state transitions. The parameters of most importance during this scenario is to verify that the automatic time measuring algorithm responsible for deriving the different timeout times from the known 5 MHz TxClk is within specification. All mode transitions are timed and check together with character output. The link is reset by cutting the bit stream to the SpaceWire Codec and observing that the correct link reset procedure is observed.
2. Start-up with inverted s-level just before the first Null Character is received by the SpaceWire Codec. All mode transitions is checked to specification in the same way as test case #1 but the most important functionality checked during this test case is that the receiver is capable of receiving and decoding incoming Characters even if the data flow is out of phase with the local clock, with out of phase meaning that the double dated input signal is out of sync with the Character boundaries so that each boundaries is received during falling edge RxClk. The link is reset by cutting the bit stream to the SpaceWire Codec and observing that the correct link reset procedure is observed.
3. Start-up in a noisy environment. This test case simulates that the SpaceWire Codec tries to start-up connected to an extremely noisy link, this is tested by forcing the test bench transmitter to babble incoherently for 20 us before transmitting the first valid Null Character. All this as the SpaceWire Codec performs its start-up procedure with auto start disabled, making it wait in Ready state. As before all state transitions are checked and the link is reset by violating the 850 ns timeout time.
4. Normal start-up, but let the test bench complete one start-up cycle before asserting Link Enabled. Make sure that the FCT generation from TbSpw is disabled long enough to test the 12.8 us timeout time. As before state transitions are checked but the most important characteristic tested is that the SpaceWire Codec is not affected by incoming characters when disabled and that the start-up work as specified after a failed start-up procedure on the link. The longest possible working delay for Null reception is the second thing tested. Link reset as before.

5. Normal start-up but disable test bench FCT transmit capabilities for long enough to test the 12.8 us timeout time. The last of the nominal start-up scenarios test the SpaceWire Codec's capability to start the link even when starved on FCT characters for the longest possible time. Link reset as before.

#### 5.4.2 TimeCode test

This test aims at verifying the functionality of the TimeCode interface. The main thing tested is that the SpaceWire Codec is capable of decoding incoming TimeCode characters and to the expected stimuli to the system. The SpaceWire capability to generate proper TimeCodes is also tested.

1. Start-up the SpaceWire link and verify that everything is nominal.
2. Send a TimeCode equal to one and verify the TimeCode and that TickOut is asserted.
3. Send a TimeCode equal to one again and verify that TickOut remains deasserted.
4. Send a TimeCode equal to zero and verify that TickOut remains deasserted.
5. Send a TimeCode equal to one and verify that TickOut is asserted.
6. Send a TimeCode equal to 63 and verify that TickOut remains deasserted.
7. Send a TimeCode equal to zero and verify that TickOut is asserted.
8. Send a TimeCode equal to one and expect that TickOut is asserted.
9. Send a TimeCode equal to two with parity error and verify Link down.
10. Wait until the link reset procedure is finished and the link is up and running.
11. Loop over different values for TimeIn and CtrlIn and assert TickIn, verify that the correct TimeCode is received by tb\_SpwCodec.

#### 5.4.3 Data test

This test aims at verifying data reception and transmission as well as nominal FCT behaviour. The first part of the Data Test aims at testing the receiver pipe of the SpaceWire Codec as well as FCT character generation. A number of scenarios are tested making sure that the SpaceWire Codec acts in a predictable way. The second part of the test, starting at #12 aims at verifying the transmitter pipe of the SpaceWire Codec. The main interest during these steps is to make sure that the SpaceWire transmitter is capable of transmitting packet in an orderly fashion as well as are able to keep track of the number of N-chars it is allowed to transmit. The last step aims at verifying that the SpaceWire Codec is capable of duplex operation.

1. Start up the SpaceWire and make sure it is running normal.
2. Send a 7 byte packet to the SpaceWire and make sure that the correct data is available in the SpaceWire.
3. Send a 58 byte packet to the SpaceWire and make sure that the correct data is available in the SpaceWire.
4. Send a 58 byte packet with a parity error in the EOP character to the SpaceWire, make sure that the correct data is available in the SpaceWire and that the packet is signalled erroneous.
5. Send a 16 byte packet to the SpaceWire with a parity error on the last byte.
6. Read out 10 bytes of data from the SpaceWire and wait until the SpaceWire have restarted the link.
7. Send a packet larger than the SpaceWires receiver buffer and make sure that one of the last bytes has a parity error.
8. Read back the rest of the data from #5 and enough of the packet from #7, allowing the new parity error to be transmitted.
9. Read back the rest of the packet from #7, check all data and expect an erroneous end of packet.
10. Send a 60 byte packet to the SpaceWire and force a Null character with parity error once the receiver buffer is full.
11. Read the data from the SpaceWire, expect a buffer sized data packet with an erroneous end of packet.
12. Send a 7 byte packet from the SpaceWire and make sure that the correct data is received by the test bench.
13. Send a 58 byte packet from the SpaceWire and make sure that the correct data is received by the test bench.
14. Disable the automatic FCT generation of the test bench before sending a 63 byte packet from the SpaceWire. Make sure that the packet is not received in full for several ms.
15. Enable the automatic FCT generation once again and prepare the SpaceWire to send one more 17 byte packet as soon as the first is completed.
16. Make sure that both packets are received by the test bench.
17. Disable the automatic FCT generation of the test bench before sending a large enough packet from the SpaceWire. The packet is of a size that makes it occupy some of the SpaceWire transmission FIFO. Make sure the packet is not received in full by the test bench for several ms.

18. Send a 73 byte packet from the SpaceWire, this shall not be stuck in the transmission FIFO waiting for the previous packet to depart.
19. Force a Null character with parity error from the test bench and wait until the link is reset. Then read back the broken packet from #17 and enable the automatic FCT generation before reading back the 73 byte packet from #18. Expect an erroneous end of packet for the first packet and a full correct packet for the data packet from #18.
20. Disable the automatic FCT generation of the test bench before sending a large enough packet from the SpaceWire. The packet is of a size that makes it occupy some of the asynchronous interface between the Tx clock region and the bus clock region. Make sure the packet is not received in full by the test bench for several ms.
21. Send an 80 byte packet from the SpaceWire, this shall not be stuck in the transmission FIFO waiting for the previous packet to depart.
22. Force a Null character with parity error from the test bench and wait until the link is reset. After the link is restarted, read back the broken packet from #20 and enable the automatic FCT generation before reading back the 80 byte packet from #21. Expect an erroneous end of packet for the first packet and a full correct packet for the data packet from #21.
23. Send a 120 byte packet from the SpaceWire and at the same time, send a 122 byte packet to the SpaceWire. Make sure both packets are received in full.

#### 5.4.4 FCT test

This test aims at verifying that the FCT of the SpaceWire works as intended. The SpaceWire will be subjected to different scenarios, nominal as well as erroneous, to cover as much of the functionality as possible. The SpaceWire Codec needs to keep track of both how much room it has in its own receiver FIFO and how much room there is left in the receiver FIFO on the other side of the link, failure to do so will result in a Link down procedure to restart the link. The first part of the test from #1 to #11 mainly targets the receive buffer of the SpaceWire Codec and the remainder of the test aims to verify that the SpaceWire can keep track of the size of the buffer on the other side of the link.

1. Start the link in autostart mode, make sure everything is normal.
2. Send a data packet to the SpaceWire Codec that is one byte smaller than the receiver FIFO, this in order to utilize the entire FIFO and maximize the number of FCT characters sent.
3. Make sure that the correct data have arrived and that the correct number of FCT characters have been transmitted.
4. Send in more data than the receive FIFO in the SpaceWire Codec can handle, expect the link to reset.

5. Read back as much data as can be stored in the buffer and make sure it's correct.
6. Wait until the link has been restarted and then send in as much data as can be stored in the receive buffer of the SpaceWire Codec.
7. Read the data back and make sure the correct amount of FCT characters have been transmitted.
8. Send a packet of size Rx FIFO size \* 2 + 5 to the SpaceWire Codec.
9. Read back Rx FIFO size + 4 of the data transmitted in #8.
10. Flood the FIFO by forcing the test bench to transmit more data than the receiver FIFO can handle.
11. Read back the rest of the data and make sure that it is correct.
12. Send a 6 byte packet from the SpaceWire Codec.
13. Force the test bench to send too many FCT characters and verify that the link is reset. Verify the data sent from the SpaceWire Codec.

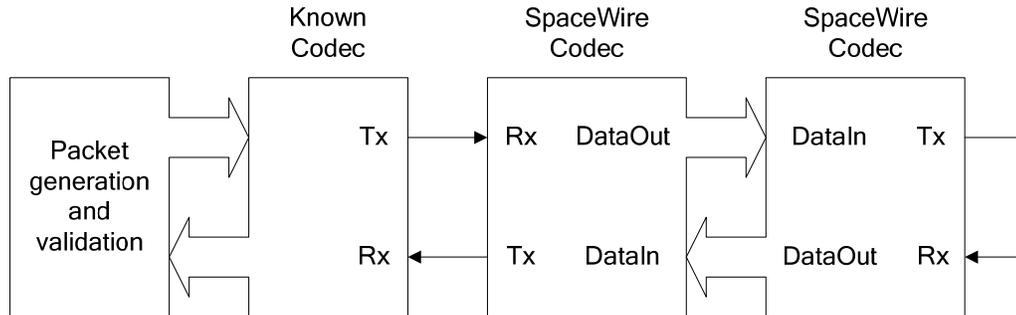
#### 5.4.5 FIFO flush test

This test aims at verifying that the Rx and Tx FIFO flush works as specified.

1. Start the link and disable FCT generation in the test bench after the SpaceWire have reached Run state.
2. Send one packet from the SpaceWire Codec that is the size of the test bench FIFO + 2.
3. Send one more packet from the SpaceWire Codec and make sure that it's big enough to not fit in the space that is left in the SpaceWire Tx FIFO.
4. Send one packet to the SpaceWire that is large enough to fill the Rx FIFO.
5. Send one FCT with parity error and wait until the link resets. At this point the packet from #2 is flushed by the link down procedure and the packet from #3 is waiting to be sent. The packet from #4 is in the Rx FIFO and.
6. Flush both the Rx and the Tx FIFO.
7. Make sure that the next packets transmitted and received over the link is correct.

## 5.5 Verification in hardware

This chapter explains the verification procedure in hardware. The chapter starts with a description of the test platform the module will be running on followed by an explanation of the setup, its capabilities and why this method was used and the limitations that the setup inflicts on the verification.



**Figure 20:** Setup for SpaceWire validation.

In order to test as much of the SpaceWire capabilities as possible two SpaceWire Codec's were connected back to back with the last one connected as a loop back. The first SpaceWire Codec was then connected to a proven and tested spacewire codec to verify that the design works together with an standard device. The setup was chosen in order to both verify the front end link while connected to a standard device and the capability of the SpaceWire Codec to drive another SpaceWire codec without any glue logic for the data path. The test was performed at a low data rate since the standard device did not support more then  $4 / 3$  times the Rx- / Tx-speed / *BusClk*.

The test comprised of sending data from the first standard device, trough the two SpaceWire Codec's connected back to back and out through the second SpaceWire Codec and back into its own receiver. Verification was simply making sure that the data received after it had been transferred was equal to the data sent. This test does not verify that the SpaceWire Codec is capable of handling every situation and failure case but it shows that it can connect and transmit data to both a known working device and to itself.

## 6 Results

A fully functional SpaceWire codec was designed, implemented and verified during this thesis. The following subchapters will go through the criteria for evaluating the design, its implementation as well as the results gained from verification and synthesis.

### 6.1 Implementation results

The bulk of the work done during this thesis was aimed towards developing a good design for the SpaceWire Codec, implementing the design using VHDL and finally testing it both in a module level test bench and on a FPGA platform.

#### 6.1.1 Design

As seen in §4 Functional design a well-defined design was developed from a top down perspective followed by a bottom up implementation of each hierarchal block.

The major design decisions that panned out well are;

- Keeping the *RxClk* region small and minimizing the number of falling edge registers facilitates both synthesis and post synthesis work i.e. made synthesis reach higher rates for *RxClk* as well as keeps manual placing of logic to a minimum. The task of the logic inside the *RxClk* region is reduced to supplying the *BusClk* region with data bit pairs making it robust and rather failsafe.
- Keeping all intelligent decisions out of the transmitter made the *BusClk* asynchronous logic in the *TxClk* region smaller and less complicated. This together with only two falling edge registers, one in each driver for the data and strobe pipelines, helped synthesis of the *TxClk* region as well as kept manual placement of logic post synthesis low. The task of the transmitter is more or less reduced to the handling of a few asynchronous interfaces in parallel connected to a strict priority multiplexer choosing the appropriate character to transmit. This is followed by a pipeline used to generate the parity bit, strobe signal and finally accelerating to double data rate.
- Module internal data buffers. Keeping both the Rx- and TxFifo module internal and supplying all data through a well defined internal interface was the final touch that made the design of the SpaceWire Codec a self contained entity. All tedious work of handling the reception and generation of flow control tokens are hidden from the user who only needs to worry about data packets, data characters and TimeCodes.

The design decisions that did not pan out that well;

- The asynchronous interface associated with the reception of data bits does not scale well with some FPGA synthesis tools. When aiming for high *RxClk* / *BusClk* ratios the logic used to concatenate the bit-stream and find the different characters grows more then linearly.

## 6.1.2 RTL implementation

All in all there were more than 5000 lines of RTL code written during the development of the SpaceWire Codec. The code is distributed over 25 different files making up the sub blocks of the design.

All RTL-code has been written to comply with the coding standard [RUAG CSTD] as supplied by RUAG Space AB.

## 6.1.3 Test bench implementation

The test bench for the SpaceWire codec is comprised of 17 files containing more than 6000 lines of code but is not nearly enough to fully verify the design. The scope of this thesis would be too large if tests for a full verification were to be developed. Instead the verification cycled through all nominal test cases together with the most probably failures cases and verified that the SpaceWire Codec behaved as expected.

After the first easy bugs, non-connected signals and the like, the tests did not find that many. The reason for this is two-fold:

- The lengthy design stage of the development meant that the core functionality of the SpaceWire Codec was thought through before the coding started.
- The test bench was aimed at verifying the core functionality of the design.

## 6.1.4 Hardware verification

The hardware verification did not run into any problems and no bugs were revealed. This does not mean that the design is fully functional but it does prove a few features;

- The SpaceWire Codec developed during this thesis can communicate, exchange both N-Chars and L-Chars, with a known working SpaceWire codec without forcing link restarts or dropping data bytes.
- The SpaceWire Codec can communicate, exchange both N-Chars and L-Chars, with itself in loop-back.
- One SpaceWire Codec can drive the internal data interface of another instantiation of itself without any glue logic.

## 6.2 Size

The size and performance of the SpaceWire Codec can be tailored by use of the configuration parameters described in §4.4. The final size of the design was evaluated for two different settings; the minimum configuration capable of 4 bits received and transmitted each BusClk cycle and the maximum configuration capable of 10 bits.

### 6.2.1 Minimum configuration

The configuration parameters for this setting can be found in Table 10 below:

**Table 10 Minimum configuration**

<b>Parameter name</b>	<b>Value</b>
DataSampleGroups	5
RxAsyncIfFifo	16
CharacterBuffers	1
RxFifoSize	10
DataTransmitGroups	1
TxFifoSize	2

The results gained from synthesis are as follows;

**Table 11 Register count minimum configuration**

<b>Clock region</b>	<b>Register count</b>
BusClk rise reaches	328 cells
RxClk rise reaches	21 cells
RxClk fall reaches	2 cells
SpwClk rise reaches	122 cells
SpwClk fall reaches	2 cells

The total amount of registers, including all buffers needed to run stutter free, are 475. This is less than the 500 registers that the design aimed for as a minimum configuration.

From Table 11 we can also see that the registers in the *RxClk* region are a mere 23 of which only 1 is a falling edge DFF with hard timing constraints. The registers in the *TxClk* region are in total 124. The main reason for the seemingly large count of *TxClk* registers are that the asynchronous interfaces often need two buffers to keep up with transmission speeds.

Of the 328 registers in the *BusClk* region around 120 are in the dedicated input / output buffers for the data path and another 45 are used for the timeout time measuring logic, two of the features that were asked for by RUAG Space AB.

## 6.2.2 Maximum configuration

The configuration parameters for this setting are as described in table Table 12 below;

**Table 12 Maximum configuration**

Parameter name	Value
DataSampleGroups	12
RxAsyncIfFifo	20
CharacterBuffers	3
RxFifoSize	16
DataTransmitGroups	4
TxFifoSize	6

The results gained from synthesis are as follows;

**Table 13 Register count maximum configuration**

Clock region	Register count
BusClk rise reaches	555 cells
RxClk rise reaches	43 cells
RxClk fall reaches	2 cells
SpwClk rise reaches	184 cells
SpwClk fall reaches	2 cells

The total amount of registers, including all buffers needed to run stutter free are 786. This is roughly the same size as just the asynchronous interface for the data path of the previous solution with the same performance.

From Table 13 we find that there is 45 registers in total in the *RxClk* region. The registers in the *TxClk* region are in total 186 and the increase from the minimum setting are more or less exclusively from the wider data path through the asynchronous interface.

There are 555 registers in the BusClk region and out of these 220 are in the dedicated input / output buffers for the data path and another 45 are used for the timeout time measuring logic, two of the features that were asked for by RUAG Space AB.

## 6.2.3 Size comparison

When comparing the differences between the two synthesis results the following can be seen. The maximum configuration can handle 2.5 times the data rate of the minimum configuration and the size scales as follows;

- From Table 11 we find that there are 23 registers in the RxClk region with the minimum configuration and from Table 13 we find that there is 45 registers in total in the RxClk region for the maximum configuration. This makes the size of the RxClk region scale better than the increase in performance with 1.96 times the registers.

- From Table 11 we find that there are 124 registers in the TxClk region with the minimum configuration and from Table 13 we find that there is 186 registers in total in the TxClk region for the maximum configuration. This means that the TxClk region scales 1.5 times the registers for a 2.5 time increase in performance.
- For the BusClk region the following can be seen. Dividing the size for the maximum configuration with the size of the minimum we find the relationship of  $555 / 328 = 1.69$  with the same 2.5 times the performance.

One of the demands placed on the design was that the size of the data path should scale more or less linearly with performance and this seems to be the case. The reason why the scaling factor is better than 2.5 is that only the data paths and not the control logic scale with the different speed settings.

## 6.3 Performance

The design was synthesized in two different configurations for two different rad-hard ASIC technologies, these are;

- The MH1 technology, a 350 nm radiation hardened library. The target speed for this was 60 MHz for the BusClk region and 100 MHz, or 200 Mbps, for both the RxClk and the TxClk regions.
- The ATC18 technology, a 180 nm radiation hardened library. The target speed for this was 100 MHz for the BusClk region and at least 200 MHz, or 400 Mbps, for both RxClk and TxClk regions.

All values are gathered in a worst case scenario, meaning high temperature, low voltage and a bad batch at the fab. All values are gathered with symmetric duty-cycles and medium effort on synthesis. All this adds up to worst case everything.

### 6.3.1 MH1

The results for when using the minimum configuration:

BusClk passed with 16.5 ns clock period, equal to 60 MHz BusClk frequency.

RxClk passed with 8 ns clock period, equal to 125 MHz RxClk frequency.

TxClk passed with 8 ns clock period, equal to 125 MHz TxClk frequency.

The results for when using the maximum configuration:

BusClk passed with 16.5 ns clock period, equal to 60 MHz BusClk frequency.

RxClk passed with 8 ns clock period, equal to 125 MHz RxClk frequency.

TxClk passed with 8 ns clock period, equal to 125 MHz TxClk frequency.

The SpaceWire Codec fulfils all requirements on clock speed that was placed on the design.

### 6.3.2 ATC18

The results for when using the minimum configuration:

BusClk passed with 10 ns clock period, equal to 100 MHz BusClk frequency.

RxCclk passed with 4 ns clock period, equal to 250 MHz RxCclk frequency.

TxCclk passed with 4 ns clock period, equal to 250 MHz TxCclk frequency.

The results for when using the maximum configuration:

BusClk passed with 10 ns clock period, equal to 100 MHz BusClk frequency.

RxCclk passed with 4 ns clock period, equal to 250 MHz RxCclk frequency.

TxCclk passed with 4 ns clock period, equal to 250 MHz TxCclk frequency.

The SpaceWire Codec fulfils all requirements on clock speed that was placed on the design.

## 6.4 Future development

The SpaceWire standard is still a rather young standard and the SpaceWire Codec designed during this thesis most probably needs updates in the near future. The upcoming revision of the standard will change the TimeCode format.

Upcoming projects also need a SpaceWire with RMAP support, making the need for an updated and compatible RMAP module a necessity.

The core part of the SpaceWire codec should be able to remain as is since the bit-rate is high enough for all plausible implementations. The one thing I would change in the design is the partitioning of the asynchronous interface for the Tx pipeline. With a little more care the SpaceWire codec could shed some more DFFs in small remote terminal applications where the SpwClk and the BusClk are synchronous.

## 7 Conclusions

The implementation of the SpaceWire Codec was a success. The codec does meet all of the company specific demands listed in 3.1 as well as comply with [ECSS SPW]. Most of the design ideas worked out well and the core functionality of the design has been proven in both verification using a test bench and in a validation test on a commercial FPGA-platform.

The amount of testing and validation done during this thesis is sufficient to know that the concept works and that the general design is good enough, but it not sufficient to make the codec space worthy. Further testing by a team company employed engineers must be performed at RUAG Space AB in order to prove that the design is good enough for space mission.

A spacification of the SpaceWire Codec has been performed during the time that this paper was written and the design is now deemed fit for space missions. The design, as developed during this thesis, has already been used in IP-cores and is scheduled for use in the next ASIC developed at RUAG Space AB.

## 8 References

- [ECSS SPW] ESA Requirement & Standards Division ESTEC (2008)  
SpaceWire – Links, nodes, routers and networks; ECSS-E-ST-50-12C
- [ECSS PTP] ESA Requirement & Standards Division (2010)  
CCSDS Packet Transfer Protocol; ECSS-E-ST-50-53C
- [ECSS RMAP] ESA Requirement & Standards Division (2010)  
Remote Memory Access Protocol; ECSS-E-ST-50-52C
- [IEEE 1355-1995] IEEE Standards Board (1995)  
IEEE Standard for Heterogeneous InterConnect; IEEE Std 1355-1995
- [RUAG CSTD] RUAG Space GDC  
VHDL Coding Standard; S-DSTD-HWI-00029-SE
- [RUAG ASYNC] RUAG Space GDC  
ASIC, Multiple Clocks and Asynchronous Interfaces; S-DSTD-HWI-00030-SE