



---

## SpaceWire Remote Terminal Controller

---

### PRELIMINARY USER GUIDE

---

#### Features

- SPARC V8 integer unit with 5-stage pipeline, 4 kbyte instruction and 4 kbyte data cache
- Double precision IEEE-754 floating point unit
- EDAC protected interface to multiple 8/32-bits PROM/SRAM memory banks and I/O
- Advanced on-chip debug support unit
- UARTs, Timers, Watchdog, GPIO, Digital ADC/DAC interfaces, Interrupt controller
- Two SpaceWire links with RMAP
- Redundant CAN 2.0 interface with DMA
- FIFO interface with DMA
- Up to 50 MHz system frequency
- Up to 200 Mbit/s SpaceWire data rate
- 349-pin MCGA with 50 mil pin spacing

---

#### Description

The SpaceWire Remote Terminal Controller (RTC) is a bridge between the SpaceWire network and the CAN bus, providing a fully integrated system. Additional features are provided to cater for autonomy of remote terminals and to relieve the central processing chain of repetitive standard acquisitions and management duties.

Page Left Blank Intentionaly

## Table of Contents

---

1. INTRODUCTION .....	5
1.1 Scope .....	5
1.2 Reference documents .....	5
1.3 Source reference .....	5
1.4 System overview .....	5
1.5 Block diagram .....	6
1.6 Description of typical systems using the device .....	7
2. FUNCTIONAL OVERVIEW .....	7
2.1 General functionality .....	7
2.2 General interfaces .....	8
3. PROCESSOR AND PERIPHERALS .....	10
3.1 LEON integer unit .....	10
3.2 Cache sub-system .....	14
3.3 On-chip peripherals .....	18
3.4 External memory access .....	32
3.5 Hardware debug support .....	39
3.6 Vendor and device id .....	47
4. ON-CHIP MEMORY .....	48
4.1 Overview .....	48
4.2 Operation .....	48
4.3 Vendor and device id .....	49
4.4 Registers .....	49
5. FIFO INTERFACE .....	51
5.1 Overview .....	51
5.2 Interface .....	53
5.3 Waveforms .....	54
5.4 Transmission .....	55
5.5 Reception .....	57
5.6 Operation .....	59
5.7 Registers .....	59
6. ADC / DAC INTERFACE .....	68
6.1 Overview .....	68
6.2 Operation .....	69
6.3 Registers .....	72
7. 32-BIT TIMERS .....	78
7.1 Overview .....	78
7.2 Operation .....	78
7.3 Vendor and device id .....	78
7.4 Registers .....	79
8. 24-BIT GENERAL PURPOSE INPUT OUTPUT .....	82
8.1 Overview .....	82
8.2 Registers .....	82
9. CAN INTERFACE .....	86
9.1 Overview .....	86
9.2 Interface .....	87
9.3 Protocol .....	87
9.4 Status and monitoring .....	87

9.5	Transmission	88
9.6	Reception	90
9.7	Global reset and enable	92
9.8	Interrupt	93
9.9	Vendor and device id	93
9.10	Registers	93
9.11	Memory mapping	104
<b>10.</b>	<b>SPACEWIRE LINK INTERFACE</b>	<b>106</b>
10.1	System overview	106
10.2	Functions	106
10.3	Interfaces	106
10.4	Module overview	106
10.5	Definitions	110
10.6	Functional behaviour	123
10.7	Register definition summary	153
10.8	Vendor and device id	173
<b>11.</b>	<b>AMBA AHB CONTROLLER</b>	<b>174</b>
11.1	Overview	174
11.2	Operation	174
<b>12.</b>	<b>AMBA AHB/APB BRIDGE</b>	<b>176</b>
12.1	Overview	176
12.2	Operation	176
12.3	Vendor and device id	176
<b>13.</b>	<b>MEMORY AND REGISTER MAP, INTERRUPT ASSIGNMENT</b>	<b>177</b>
13.1	Addressing information	177
13.2	Plug & Play information	178
13.3	Registers	179
13.4	Interrupts	186
<b>14.</b>	<b>INTERFACES AND SIGNALS</b>	<b>188</b>



# 1. INTRODUCTION

## 1.1 Scope

This document establishes the User's Manual for the SpaceWire Remote Terminal Controller (RTC) device developed in the scope of the "TopNet SpaceWire Controller / Remote User Interface" activity initiated by the European Space Agency.

## 1.2 Reference documents

[AMBA]	AMBA™ Specification, Rev 2.0, ARM IHI 0011A, 13 May 1999, Issue A, first release, ARM Limited
[GRLIB]	GRLIB IP Library User's Manual, Version 1.0.7, Gaisler Research
[SPARC]	The SPARC Architecture Manual, Version 8, Revision SAV080SI9308, SPARC International Inc.
[SPWSTD]	SpaceWire - Links, Nodes Routers and Networks, ECSS-E-ST-50-12C
[CANSTD]	CAN Specification Version 2.0 Part B, BOSCH
[ISO11898]	ISO 11898:1993 and Amendment 1 (ISO 11898:1995) Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for high-speed communication, First Edition 1993, ISO
[ISO11898E]	ISO 11898:1993(E) and Amendment 1, 1995, ISO

## 1.3 Source reference

TheSpaceWire Remote Terminal Controller (RTC) design is based on the following sources:

[LEON2FT-SRC]	LEON-2 FT VHDL model, version 1.0.9.16.1-r85, 2007
[GRLIB-SRC]	GRLIB VHDL source code, version 1.0.7, 2006
[CANESA-SRC]	HURRICANE source code, version 5.1.6, dated 21 Nov 2006
[DUNDEE-SRC]	SPWB source code, version 2.0

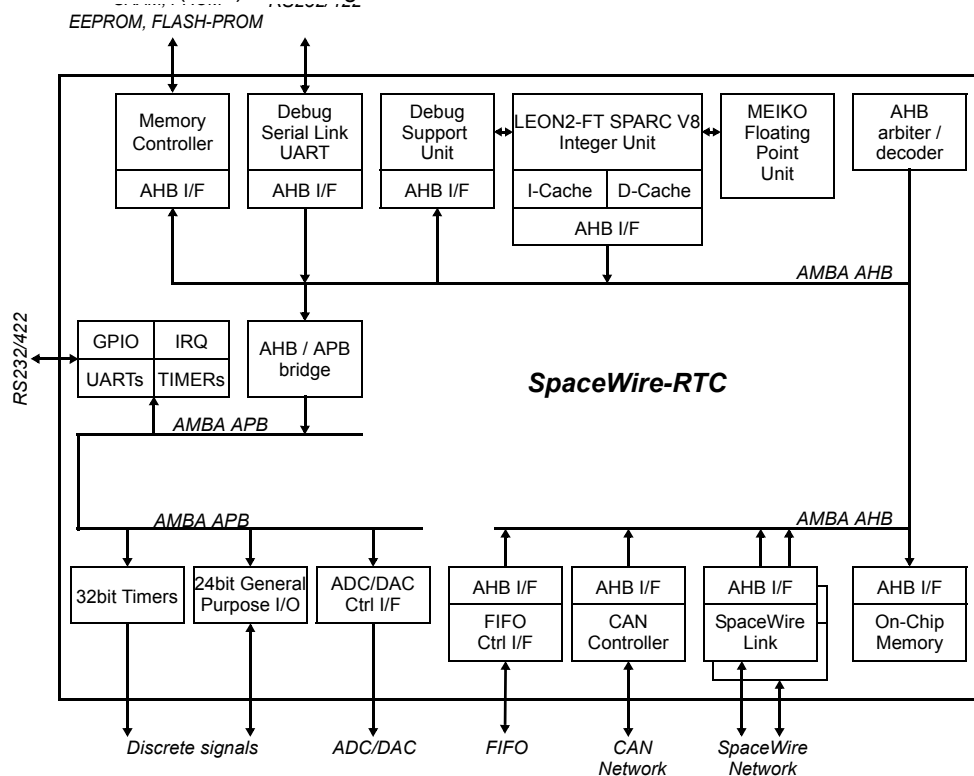
## 1.4 System overview

The SpaceWire Remote Terminal Controller (RTC) device is a bridge between the SpaceWire network (backbone) and the CAN bus, providing a fully integrated system. Additional features are provided to cater for autonomy of remote terminals and to relieve the central processing chain of repetitive standard acquisitions and management duties. The SpaceWire-RTC device can be used both in non-intelligent nodes and in nodes with local intelligence.

The SpaceWire-RTC device includes an embedded microprocessor, a CAN bus controller, ADC/DAC interfaces for analogue acquisition/conversion, standard interfaces and resources (UARTs, timers, general purpose input output).

## 1.5 Block diagram

SpaceWire Remote Terminal Controller (RTC) block diagram is shown hereafter.



**Figure 1-1. SpaceWire Remote Terminal Controller (RTC) block diagram**

The block diagram shows the functional modules that constitute the SpaceWire-RTC. The SpaceWire-RTC is based on the following on-chip buses:

- AMBA AHB bus
- AMBA APB bus

The on-chip bus selection is based on the ESA baseline/directive for using the open standard AMBA Specification (Rev 2.0) as a standard bus.

Although not directly shown in the block diagram, several of the modules on the AMBA AHB bus also have secondary AMBA APB interfaces for configuration and control purposes. These AMBA APB interfaces are logically connected to the AMBA APB bus shown in the block diagram. For the modules that are shown in the block diagram as only connected to the AMBA APB bus, there are no other hidden connections to the AMBA AHB bus.

The AMBA AHB bus is controlled by an AHB controller with plug&play support. The controller implements the AMBA AHB bus with the following sideband information:

- cacheability information
- interrupt bus
- configuration information
- diagnostic information

The AMBA APB bus is controlled by an AHB/APB bridge with plug&play support. The bridge implements the AMBA APB bus with the following sideband information:

- interrupt bus
- configuration information
- diagnostic information

## 1.6 Description of typical systems using the device

The SpaceWire Remote Terminal Controller (RTC) device can be integrated in the instrument controller Unit (ICU) that acts as the payload data processor and mainly receives payload data from instruments and produces processed data to be down linked. The main data communication is performed via the SpaceWire network. The ICU is however controlled and monitored via the CAN network from the On-Board Computer (OBC). The CAN controller in the SpaceWire-RTC device acts as a remote terminal that is being managed by the OBC.

Alternatively, the SpaceWire-RTC device can be integrated in the On-Board Computer (OBC). Since the OBC acts as the network manager on the CAN network, the CAN controller carries capability such as node management and time distribution. The OBC also communicates or manages the SpaceWire network via SpaceWire links.

As can be seen from the above application scenarios, the capabilities of the SpaceWire-RTC device are not limited only to support the CAN bus in the ICU, but also allows it to be used in an OBC. This reduces future development costs since the same device is used in both payload and avionics. This also promotes the usage of hybrid SpaceWire and CAN networks and the TopNet concept.

To bring the concept a step further, one could envisage applications in which the SpaceWire-RTC device actually replaces the processor in an ICU. The integer processing capacity of the LEON2 processor in the SpaceWire-RTC device is in par with what can be expected from current monolithic processor devices. This provides for savings in terms of power and board area, since a single device with external memory is sufficient to form the core of an ICU.

The main application of the SpaceWire-RTC device is however in instruments or individual experiments of the payload. It provides an abundance of interfaces, each with a high degree of programmability and configurability. It is able to acquire analogue and digital data, generated by connected peripherals and to generate discrete commands towards the same peripherals.

The SpaceWire-RTC device can be operated stand-alone or with a number of external devices such as SRAM, PROM and FIFO memories, ADC and DAC converters. The device can be managed locally by the on-chip processor, or remotely via its SpaceWire link interfaces.

SpaceWire-RTC device can operate as a single-chip system, with software being uploaded to its on-chip memory via the SpaceWire link interface, forming a compact solution for remotely controlled applications. Or it can operate in a full-size system, with software being decompressed from local PROM and executed from multiple fast and wide SRAM memory banks.

## 2. FUNCTIONAL OVERVIEW

### 2.1 General functionality

The SpaceWire-RTC ASIC implements the following functions:

- Processor
  - The SpaceWire-RTC ASIC includes the LEON2-FT SPARC V8 Integer Unit, featuring an instruction cache of 4 kbytes, and a data cache of 4 kbytes, and a Meiko Floating Point Unit.
- Debug Support Unit
  - The SpaceWire-RTC ASIC includes the LEON2-FT Debug Support Unit (DSU) with a Trace Buffer of 512 lines of 16 bytes.
- Debug Serial Link UART
  - The SpaceWire-RTC ASIC includes the LEON2-FT serial debug interface for AMBA AHB.
- The SpaceWire-RTC ASIC includes the LEON2-FT peripherals:
  - Interrupt Controller and Secondary Interrupt Controller
  - 32-bit Timers (three)
  - UART Serial Links (two)
  - 16-bit General Purpose Input Output
- Memory Interface
  - The SpaceWire-RTC ASIC includes the LEON2-FT Memory Controller, including EDAC protection and support for SRAM, PROM, EEPROM, and a memory mapped I/O area.
- On-Chip Memory
  - The SpaceWire-RTC ASIC includes 64 kbytes EDAC protected on-chip memory.

- **FIFO Interface**
  - The SpaceWire-RTC ASIC can simultaneously interface two FIFO devices, one for input and one for output. The interface features one DMA channel in either direction.
- **ADC/DAC Interface**
  - The SpaceWire-RTC ASIC can simultaneously interface ADC and DAC devices.
- **32-bit Timers**
  - The SpaceWire-RTC ASIC includes two additional 32-bit timers, being cascable and with optional external clock input.
- **24-bit General Purpose Input Output**
  - The SpaceWire-RTC ASIC includes 24 additional general-purpose input output channels, supporting input, output and pulse generation.
- **CAN Interface**
  - The SpaceWire-RTC ASIC includes the ESA HurriCANE CAN controller. The interface features one DMA channel in either direction and is compatible with the CANopen application layer protocol.
- **SpaceWire Link Interface**
  - The SpaceWire-RTC ASIC includes two University of Dundee SpaceWire links. The interface implements DMA channels and Remote Memory Access Protocol (RMAP).
- **JTAG Interface**
  - The SpaceWire-RTC ASIC includes a JTAG interface with TAP controller for boundary scan testing.

## 2.2 General interfaces

The SpaceWire-RTC ASIC provides the following external and internal interfaces:

- **Debug Serial Link UART**
  - A simple communication protocol is provided to transmit access parameters and data on the internal AMBA bus. A command comprises a control byte, followed by and a 32-bit address, followed by optional write data.
- **Interrupt Controller**
  - External interrupts can be selected from the 16-bit General Purpose Input Output or the data bus of the Memory Interface.
- **32-bit Timers**
  - Watchdog with external trigger signal.
- **UART Serial Links**
  - Two Universal Asynchronous Receiver and Transmitters (UART), supporting optional parity, internal or external clock source, hardware handshake and programmable baud rate.
- **16-bit General Purpose Input Output**
  - Programmable input output channels, shared with the interrupt controller inputs and the UART serial links.
- **Memory Interface**
  - Supports two PROM banks, four SRAM banks and one memory mapped I/O. Features 23 byte-address bits, 32 data bits and 8 check bits. Unused data bits can be used as general purpose input output.
- **FIFO Interface**
  - Supports one input and one output external FIFO device, with 8- or 16-bit wide data. Unused data bits can be used as general purpose input output.
- **ADC/DAC Interface**
  - Supports one ADC and one DAC device, with 8- or 16-bit wide data, and 8-bit address. Unused address and data bits can be used as general purpose input output.
- **32-bit Timers**
  - Supports external clock source and external triggers.
- **24-bit General Purpose Input Output**

- Dedicated programmable input output channels, with input interrupts.
- CAN Interface
  - Supports nominal and redundant transmit and receive pair, with non-simultaneously operation.
- SpaceWire Link Interface
  - Supports nominal and redundant SpaceWire links, with simultaneously operation.
- JTAG Interface
  - Supports standard TAP signal interface.

## 3. PROCESSOR AND PERIPHERALS

### 3.1 LEON integer unit

The LEON integer unit (IU) implements SPARC integer instructions as defined in SPARC Architecture Manual version 8. It is a new implementation, not based on any previous designs. The implementation is focused on portability and low complexity.

#### 3.1.1 Overview

The LEON integer unit has the following features:

- 5-stage instruction pipeline
- Separate instruction and data cache interface
- Support for 8 register windows
- Multiplier 16x16
- Radix-2 divider (non-restoring)

Here is a block diagram of the integer unit.

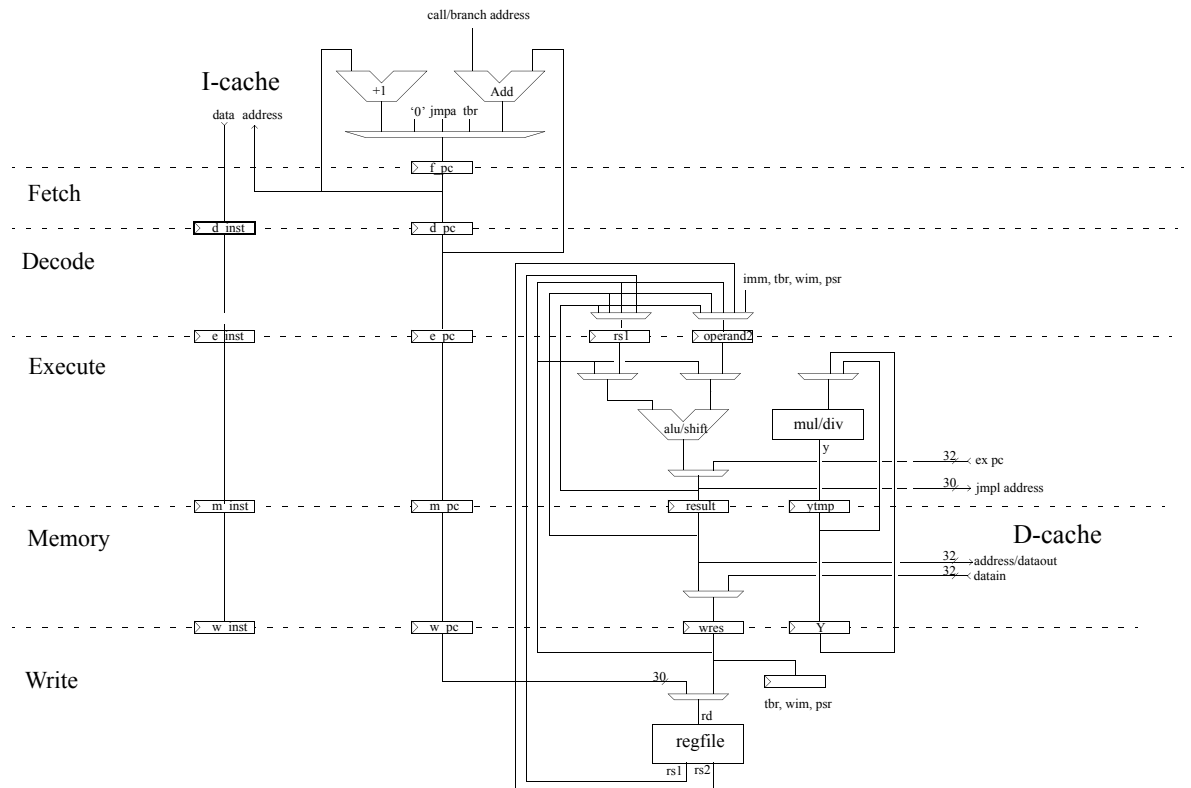


Figure 3-1. LEON integer unit block diagram

#### 3.1.2 Instruction pipeline

The LEON integer unit uses a single instruction issue pipeline with 5 stages:

- FE (Instruction Fetch): If the instruction cache is enabled, the instruction is fetched from the instruction cache. Otherwise, the fetch is forwarded to the memory controller. The instruction is valid at the end of this stage and is latched inside the IU.
- DE (Decode): The instruction is decoded and the operands are read. Operands may come from the register file or from internal data bypasses. CALL and Branch target addresses are generated in this stage.

- EX (Execute): ALU, logical, and shift operations are performed. For memory operations (e.g., LD) and for JMPL/RETT, the address is generated.
- ME (Memory): Data cache is accessed. For cache reads, the data will be valid by the end of this stage, at which point it is aligned as appropriate. Store data read out in the execution stage is written to the data cache at this time.
- WR (Write): The result of any ALU, logical, shift, or cache read operations are written back to the register file.

The following table lists the cycles per instruction (assuming cache hit and no load interlock):

Instruction	Cycles
JMPL	2
Double load	2
Single store	2
Double store	3
SMUL/UMUL	5
SDIV/UDIV	35
Taken Trap	4
Atomic load/store	3
All other instructions	1

**Table 3-1. Instruction timing**

### 3.1.3 Multiply instructions

The LEON processor supports the SPARC integer multiply instructions UMUL, SMUL UMULCC and SMULCC. These instructions perform a 32x32-bit integer multiply, producing a 64-bit result. SMUL and SMULCC performs signed multiply while UMUL and UMULCC performs unsigned multiply. UMULCC and SMULCC also set the condition codes to reflect the result.

### 3.1.4 Divide instructions

Full support for SPARC V8 divide instructions is provided (SDIV/UDIV/SDIVCC/UDIVCC). The divide instructions perform a 64-by-32 bit divide and produce a 32-bit result. Rounding and overflow detection is performed as defined in the SPARC V8 standard.

### 3.1.5 Register file SEU protection

To prevent erroneous operations from SEU errors in the main register file, each word is protected using a 7-bit EDAC checksum. Checking of the EDAC bits is done every time a fetched register value is used in an instruction. If a correctable error is detected, the erroneous data is corrected before being used. At the same time, the corrected register value is also written back to the register file. A correction operation incurs a delay 4 clock cycles, but has no other software visible impact. If an un-correctable error is detected, a register error trap (tt=0x20) is generated.

The implemented protection scheme has an impact on double-store instructions: the write-buffer will delay the request of the memory bus one clock cycle in order to not start any memory store cycle before the second store data word has been checked and (potentially) corrected.

The register file protection operation is controlled using application-specific register 16 (%asr16). The register is accessed using the RDASR/WRASR instructions.

31																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																								</
----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

**Figure 3-2. Register file protection control register (%asr16)**

- [0]: DI - disable checking. If set, will disable the register-file checking function.
- [1]: TE - Test enable.
- [8:2] TCB[6:0] - Test checkbits.
- [11:9] CNT[2:0] - Error counter. This field will be incremented for each corrected error.

The protection can be disabled by clearing the DI bit (this bit is set to '1' after reset). By setting the TE bit, errors can be inserted in the register file to test the protection function. Since a 7-bit EDAC is used, when the test mode is enabled the register checksum is XORed with the TCB field before written to the register file. The CNT field is incremented each time a register correction is performed, but saturates at "111".

### 3.1.6 Processor reset operation

The processor is reset by asserting the RESET input for at least one clock cycle. The following table indicates the reset values of the registers which are affected by the reset. All other registers maintain their value (or are undefined).

Register	Reset value
PC (program counter)	0x0
nPC (next program counter)	0x4
PSR (processor status register)	ET=0, S=1
CCR (cache control register)	0x0

**Table 3-2. Processor reset values**

Execution will start from address 0.

### 3.1.7 Exceptions

LEON adheres to the general SPARC trap model. The table below shows the implemented traps and their individual priority. When PSR (processor status register) bit ET=0, an exception trap causes the processor to halt execution and enter error mode, and the external error signal will then be asserted.

Trap	TT	Pri	Description
reset	0x00	1	Power-on reset
write error	0x2b	2	write buffer error
instruction_access_error	0x01	3	Error during instruction fetch
illegal_instruction	0x02	5	UNIMP or other un-implemented instruction
privileged_instruction	0x03	4	Execution of privileged instruction in user mode
fp_disabled	0x04	6	FP instruction while FPU disabled
cp_disabled	0x24	6	CP instruction while Co-processor disabled
watchpoint_detected	0x0B	7	Hardware breakpoint match
window_overflow	0x05	8	SAVE into invalid window
window_underflow	0x06	8	RESTORE into invalid window
register_hardware_error	0x20	9	register file EDAC error (LEON-FT only)
mem_address_not_aligned	0x07	10	Memory access to un-aligned address
fp_exception	0x08	11	FPU exception

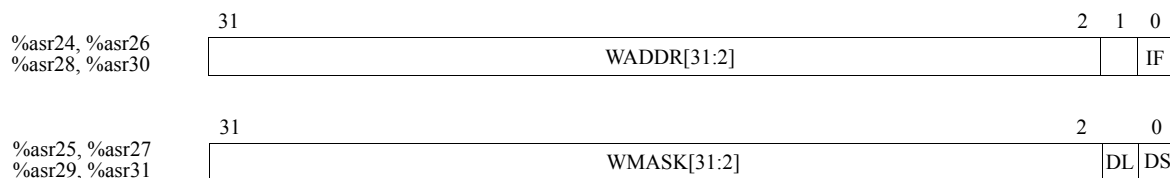


Trap	TT	Pri	Description
cp_exception	0x28	11	Co-processor exception
data_access_exception	0x09	13	Access error during load or store instruction
tag_overflow	0x0A	14	Tagged arithmetic overflow
divide_exception	0x2A	15	Divide by zero
interrupt_level_1	0x11	31	Asynchronous interrupt 1
interrupt_level_2	0x12	30	Asynchronous interrupt 2
interrupt_level_3	0x13	29	Asynchronous interrupt 3
interrupt_level_4	0x14	28	Asynchronous interrupt 4
interrupt_level_5	0x15	27	Asynchronous interrupt 5
interrupt_level_6	0x16	26	Asynchronous interrupt 6
interrupt_level_7	0x17	25	Asynchronous interrupt 7
interrupt_level_8	0x18	24	Asynchronous interrupt 8
interrupt_level_9	0x19	23	Asynchronous interrupt 9
interrupt_level_10	0x1A	22	Asynchronous interrupt 10
interrupt_level_11	0x1B	21	Asynchronous interrupt 11
interrupt_level_12	0x1C	20	Asynchronous interrupt 12
interrupt_level_13	0x1D	19	Asynchronous interrupt 13
interrupt_level_14	0x1E	18	Asynchronous interrupt 14
interrupt_level_15	0x1F	17	Asynchronous interrupt 15
trap_instruction	0x80 - 0xFF	16	Software trap instruction (TA)

**Table 3-3. Trap allocation and priority**

### 3.1.8 Hardware breakpoints

The integer unit can be configured to include up to four hardware breakpoints. Each breakpoint consists of a pair of application-specific registers (%asr24/25, %asr26/27, %asr28/30 and %asr30/31) registers; one with the break address and one with a mask:



*Figure 2. Watch-point registers*

Any binary aligned address range can be watched - the range is defined by the WADDR field, masked by the WMASK field (WMASK[x] = 1 enables comparison). On a breakpoint hit, trap 0x0B is generated. By setting the IF, DL and DS bits, a hit can be generated on instruction fetch, data load or data store. Clearing these three bits will effectively disable the breakpoint function.

### 3.1.9 Floating-point unit

The Meiko FPU is attached using an integrated interface inside the IU pipeline. The integrated FPU interface does not implement a floating-point queue, and the processor is stopped during the execution of floating-point instructions. This means that QNE bit in the %fsr register always is zero, and any attempts of executing the STDFQ instruction will generate a FPU exception trap.

## 3.2 Cache sub-system

### 3.2.1 Overview

The LEON processor implements a Harvard architecture with separate instruction and data buses, connected to two independent cache controllers. In addition to the address, a SPARC processor also generates an 8-bit address space identifier (ASI), providing up to 256 separate, 32-bit address spaces. During normal operation, the LEON processor accesses instructions and data using ASI 0x8 - 0xB as defined in the SPARC standard. Using the LDA/STA instructions, alternative address spaces can be accessed. The table shows the ASI usage for LEON. Only ASI[3:0] are used for the mapping, ASI[7:4] have no influence on operation.

ASI	Usage
0x0, 0x1, 0x2, 0x3	Forced cache miss (replace if cacheable)
0x4, 0x7	Forced cache miss (update on hit)
0x5	Flush instruction cache
0x6	Flush data cache
0x8, 0x9, 0xA, 0xB	Normal cached access (replace if cacheable)
0xC	Instruction cache tags
0xD	Instruction cache data
0xE	Data cache tags
0xF	Data cache data

**Table 3-4. ASI usage**

Access to ASI 4 and 7 will force a cache miss, and update the cache if the data was previously cached. Access with ASI 0 - 3 will force a cache miss, update the cache if the data was previously cached, or allocated a new line if the data was not in the cache and the address refers to a cacheable location.

Address range	Area	Cached
0x00000000 - 0x1FFFFFFF	PROM	Cacheable
0x20000000 - 0x3FFFFFFF	I/O	Non-cacheable
0x40000000 - 0x7FFFFFFF	RAM	Cacheable
0x80000000 - 0x9FFFFFFF	Internal (AHB)	Non-cacheable
0xA0000000 - 0xBFFFFFFF	On-Chip RAM	Cacheable
0xB0000000 - 0xFFFFFFFF	Internal (AHB)	Non-cacheable

**Table 3-5. Default cache table**

## 3.2.2 Instruction cache

### 3.2.2.1 Operation

The instruction cache is configured as a direct-mapped cache. The set size is 4 kbyte and divided into cache lines of 32 bytes. Each line has a cache tag associated with it consisting of a tag field, valid field with one valid bit for each 4-byte sub-block. On an instruction cache miss to a cachable location, the instruction is fetched and the corresponding tag and data line updated.

If instruction burst fetch is enabled in the cache control register (CCR) the cache line is filled from main memory starting at the missed address and until the end of the line. At the same time, the instructions are forwarded to the IU (streaming). If the IU cannot accept the streamed instructions due to internal dependencies or multi-cycle instruction, the IU is halted until the line fill is completed. If the IU executes a control transfer instruction (branch/CALL/JMPL/RETT/TRAP) during the line fill, the line fill will be terminated on the next fetch. If instruction burst fetch is enabled, instruction streaming is enabled even when the cache is disabled. In this case, the fetched instructions are only forwarded to the IU and the cache is not updated.

If a memory access error occurs during a line fill with the IU halted, the corresponding valid bit in the cache tag will not be set. If the IU later fetches an instruction from the failed address, a cache miss will occur, triggering a new access to the failed address. If the error remains, an instruction access error trap (tt=0x1) will be generated.

### 3.2.2.2 Instruction cache tag

A instruction cache tag entry consists of several fields as shown in following figure:

Tag for 4 kbyte set, 16bytes/line

31	12	11	8	7	0
ATAG			0000	VALID	

**Figure 3-3. Instruction cache tag layout**

[31:12]: Address Tag (ATAG) - Contains the tag address of the cache line.

[11:8]: Unused. No affect when written to. Undefined when read.

[7:0]: Valid (V) - When set, the corresponding sub-block of the cache line contains valid data. These bits are set when a sub-block is filled due to a successful cache miss; a cache fill which results in a memory error will leave the valid bit unset. A FLUSH instruction will clear all valid bits. V[0] corresponds to address 0 in the cache line, V[1] to address 1, V[2] to address 2 and so on.

## 3.2.3 Data cache

### 3.2.3.1 Operation

The data cache is configured as a direct-mapped cache. The set size is 4 kbyte and divided into cache lines of 16 bytes. Each line has a cache tag associated with it consisting of a tag field, valid field with one valid bit for each 4-byte sub-block. On a data cache read-miss to a cachable location 4 bytes of data are loaded into the cache from main memory. The write policy for stores is write-through with no-allocate on write-miss. In a multi-set configuration a line to be replaced on read-miss is chosen according to the replacement policy. If a memory access error occurs during a data load, the corresponding valid bit in the cache tag will not be set. and a data access error trap (tt=0x9) will be generated.

### 3.2.3.2 Write buffer

The write buffer (WRB) consists of three 32-bit registers used to temporarily hold store data until it is sent to the destination device. For half-word or byte stores, the stored data replicated into proper byte alignment for writing to a word-addressed device, before being loaded into one of the WRB registers. The WRB is emptied prior to a load-miss cache-fill sequence to avoid any stale data from being read in to the data cache.

Since the processor executes in parallel with the write buffer, a write error will not cause an exception to the store instruction. Depending on memory and cache activity, the write cycle may not occur until several clock cycles after the store instructions has completed. If a write error occurs, the currently executing instruction will take trap 0x2b.

Note: the 0x2b trap handler should flush the data cache, since a write hit would update the cache while the memory would keep the old value due the write error.

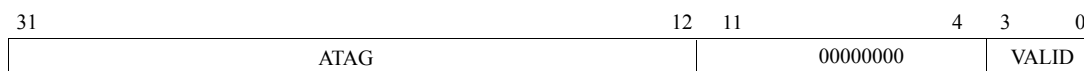
### 3.2.3.3 Data cache snooping

The data cache can optionally perform snooping on the AHB bus. When snooping is enabled, the data cache controller will monitor write accesses to the AHB bus performed by other AHB masters (DMA). When a write access is performed to a cacheable memory location,

the corresponding cacheline will be invalidated in the data cache if present. Cache snooping has no overhead and does not affect performance. It can be dynamically enabled/disabled through bit 23 in the cache control register. Cache snooping requires the target technology to implement dual-port memories, which will be used to implement the cache tag RAM.

### 3.2.3.4 Data cache tag

A data cache tag entry consists of several fields as shown in following figure:



**Figure 3-4. Data cache tag layout**

[31:12]: Address Tag (ATAG) - Contains the address of the data held in the cache line.

[11:4]: Unused. No affect when written to. Undefined when read.

[3:0]: Valid (V) - When set, the corresponding sub-block of the cache line contains valid data. These bits is set when a sub-block is filled due to a successful cache miss; a cache fill which results in a memory error will leave the valid bit unset. V[0] corresponds to address 0 in the cache line, V[1] to address 1, V[2] to address 2 and V[3] to address 3.

### 3.2.4 Cache flushing

The instruction and data cache is flushed by executing the FLUSH instruction, setting the FI bit in the cache control register, or by writing to any location with ASI=0x5. The flushing will take one cycle per cache line and set during which the IU will not be halted, but during which the instruction cache will be disabled. When the flush operation is completed, the cache will resume the state (disabled, enabled or frozen) indicated in the cache control register.

### 3.2.5 Diagnostic cache access

Tags and data in the instruction and data cache can be accessed through ASI address space 0xC, 0xD, 0xE and 0xF by executing LDA and STA instructions. Address bits making up the cache offset will be used to index the tag to be accessed while the least significant bits of the bits making up the address tag will be used to index the cache set.

Diagnostic read of tags is possible by executing an LDA instruction with ASI=0xC for instruction cache tags and ASI=0xE for data cache tags. A cache line and set are indexed by the address bits making up the cache offset and the least significant bits of the address bits making up the address tag. Similarly, the data sub-blocks may be read by executing an LDA instruction with ASI=0xD for instruction cache data and ASI=0xF for data cache data. The sub-block to be read in the indexed cache line and set is selected by A[4:2].

The tags can be directly written by executing a STA instruction with ASI=0xC for the instruction cache tags and ASI=0xE for the data cache tags. The cache line and set are indexed by the address bits making up the cache offset and the least significant bits of the address bits making up the address tag. D[31:10] is written into the ATAG field (see above) and the valid bits are written with the D[7:0] of the write data. Bit D[9] is written into the LRR bit (if enabled) and D[8] is written into the lock bit (if enabled). The data sub-blocks can be directly written by executing a STA instruction with ASI=0xD for the instruction cache data and ASI=0xF for the data cache data. The sub-block to be read in the indexed cache line and set is selected by A[4:2].

Note that diagnostic access to the cache is not possible during a FLUSH operation and will cause a data exception (trap=0x09) if attempted.

### 3.2.6 Cache parity protection

The caches are provided with two parity bits per tag and per 4-byte data sub-block. The tag parity is generated from the tag value and the valid bits. Similarly, the data sub-block parity is derived from the sub-block data. The parity bits are written simultaneously with the associated tag or sub-block and checked on each access. Two parity bits are configured, with the bits corresponding to the parity of odd and even data (tag) bits.

If a tag parity error is detected during a cache access, a cache miss will be generated and the tag (and data) will be automatically updated. All valid bits except the one corresponding to the newly loaded data will be cleared. If a data sub-block parity error occurs, a miss will also be generated but only the failed sub-block will be updated with data from main memory.

### 3.2.7 Cache Control Register

The operation of the instruction and data caches is controlled through a common Cache Control Register (CCR). Each cache can be in one of three modes: disabled, enabled and frozen. If disabled, no cache operation is performed and load and store requests are passed directly to the memory controller. If enabled, the cache operates as described above. In the frozen state, the cache is accessed and kept in sync with the main memory as if it was enabled, but no new lines are allocated on read misses.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DREPL	IREPL	ISETS	DSETS	DS	FD	FI	CPC	CPTE	IB	IP	DP	ITE	IDE	DTE	DDE	DF	IF	DCS	ICS												

Figure 3-5. Cache control register

- [31:30]: Data cache replacement policy (DREPL) - 00 - no replacement policy (direct-mapped cache), 01 - random, 10 - least-recently replaced (LRR), 11 - least-recently used (LRU)
- [29:28]: Instruction cache replacement policy (IREPL) - 00 - no replacement policy (direct-mapped cache), 01 - random, 10 - least-recently replaced (LRR), 11 - least-recently used (LRU)
- [27:26]: Instruction cache associativity (ISETS) - Number of sets in the instruction cache - 1: 00 - direct mapped, 01 - 2-way associative, 10 - 3-way associative, 11 - 4-way associative
- [25:24]: Data cache associativity (DSETS) - Number of sets in the data cache - 1: 00 - direct mapped, 01 - 2-way associative, 10 - 3-way associative, 11 - 4-way associative
- [23]: Data cache snoop enable [DS] - if set, will enable data cache snooping.
- [22]: Flush data cache (FD). If set, will flush the data cache. Always reads as zero.
- [21]: Flush Instruction cache (FI). If set, will flush the instruction cache. Always reads as zero.
- [20:19]: Cache parity bits (CPC) - Indicates how many parity bits are used to protect the caches (00=none, 01=1, 10=2)
- [18:17]: Cache parity test bits. (CPTE). These bits are XOR'ed to the data and tag parity bits during diagnostic writes.
- [16]: Instruction burst fetch (IB). This bit enables burst fill during instruction fetch.
- [15]: Instruction cache flush pending (IP). This bit is set when an instruction cache flush operation is in progress.
- [14]: Data cache flush pending (DP). This bit is set when a data cache flush operation is in progress.
- [13:12]: Instruction cache tag error counter (ITE) - This field is incremented every time an instruction cache tag parity error is detected.
- [11:10]: Instruction cache data error counter (IDE) - This field is incremented each time an instruction cache data sub-block parity error is detected.
- [9:8]: Data cache tag error counter (DTE) - This field is incremented every time a data cache tag parity error is detected.
- [7:6]: Data cache data error counter (DDE) - This field is incremented each time an instruction cache data sub-block parity error is detected.
- [5]: Data Cache Freeze on Interrupt (DF) - If set, the data cache will automatically be frozen when an asynchronous interrupt is taken.
- [4]: Instruction Cache Freeze on Interrupt (IF) - If set, the instruction cache will automatically be frozen when an asynchronous interrupt is taken.
- [3:2]: Data Cache state (DCS) - Defines the current data cache state according to the following: X0= disabled, 01 = frozen, 11 = enabled. Set to '00' at reset.
- [1:0]: Instruction Cache state (ICS) - Defines the current data cache state according to the following: X0= disabled, 01 = frozen, 11 = enabled. Set to '00' at reset.

If the DF or IF bit is set, the corresponding cache will be frozen when an asynchronous interrupt is taken. This can be beneficial in real-time system to allow a more accurate calculation of worst-case execution time for a code segment. The execution of the interrupt handler will not evict any cache lines and when control is returned to the interrupted task, the cache state is identical to what it was before the interrupt.

If a cache has been frozen by an interrupt, it can only be enabled again by enabling it in the CCR. This is typically done at the end of the interrupt handler before control is returned to the interrupted task.

### 3.3 On-chip peripherals

#### 3.3.1 On-chip registers

A number of system support functions are provided directly on-chip. The functions are controlled through registers mapped APB bus according to the following table:

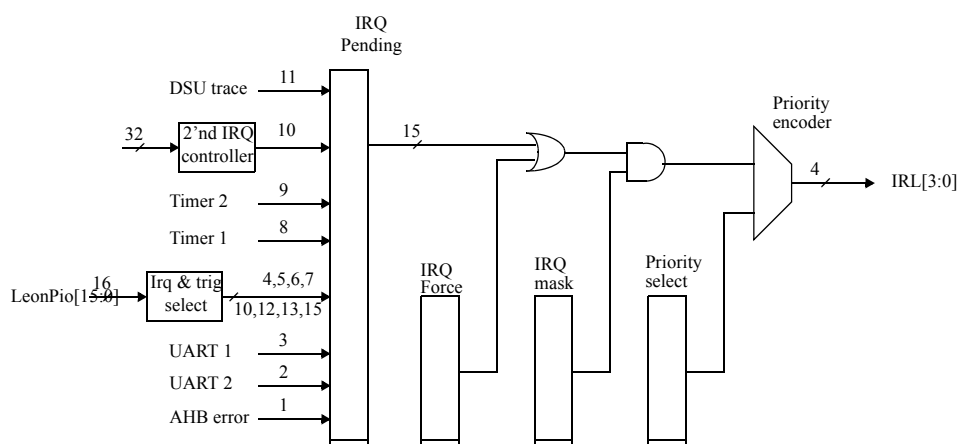
Address	Register	Address	
0x80000000	Memory configuration register 1	0x800000B0	Secondary interrupt mask register
0x80000004	Memory configuration register 2	0x800000B4	Secondary interrupt pending register
0x80000008	Memory configuration register 3	0x800000B8	Secondary interrupt status register
0x8000000C	AHB Failing address register	0x800000B8	Secondary interrupt clear register
0x80000010	AHB status register		
0x80000014	Cache control register	0x800000C4	DSU UART status register
0x80000018	Power-down register	0x800000C8	DSU UART control register
0x8000001C	Write protection register 1	0x800000CC	DSU UART scaler register
0x80000020	Write protection register 2		
0x80000024	LEON configuration register	0x800000D0	Write protect start address 1
0x80000040	Timer 1 counter register	0x800000D4	Write protect end address 1
0x80000044	Timer 1 reload register	0x800000D8	Write protect start address 2
0x80000048	Timer 1 control register	0x800000DC	Write protect end address 2
0x8000004C	Watchdog register		
0x80000050	Timer 2 counter register		
0x80000054	Timer 2 reload register		
0x80000058	Timer 2 control register		
0x80000060	Prescaler counter register		
0x80000064	Prescaler reload register		
0x80000070	UART 1 data register		
0x80000074	UART 1 status register		
0x80000078	UART 1 control register		
0x8000007C	UART 1 scaler register		
0x80000080	UART 2 data register		
0x80000084	UART 2 status register		
0x80000088	UART 2 control register		
0x8000008C	UART 2 scaler register		
0x80000090	Interrupt mask and priority register		
0x80000094	Interrupt pending register		
0x80000098	Interrupt force register		
0x8000009C	Interrupt clear register		
0x800000A0	I/O port input/output register		
0x800000A4	I/O port direction register		

Address	Register	Address	
0x800000A8	I/O port interrupt config. register 1		
0x800000AC	I/O port interrupt config. register 2		

**Table 3-6. On-chip registers**

### 3.3.2 Interrupt controller

The LEON interrupt controller is used to prioritize and propagate interrupt requests from internal or external devices to the integer unit. In total 15 interrupts are handled, divided on two priority levels. The following figure shows a block diagram of the interrupt controller.



**Figure 3-6. Interrupt controller block diagram**

#### 3.3.2.1 Operation

When an interrupt is generated, the corresponding bit is set in the interrupt pending register. The pending bits are ANDed with the interrupt mask register and then forwarded to the priority selector. Each interrupt can be assigned to one of two levels as programmed in the interrupt level register. Level 1 has higher priority than level 0. The interrupts are prioritised within each level, with interrupt 15 having the highest priority and interrupt 1 the lowest. The highest interrupt from level 1 will be forwarded to the IU - if no unmasked pending interrupt exists on level 1, then the highest unmasked interrupt from level 0 will be forwarded. When the IU acknowledges the interrupt, the corresponding pending bit will automatically be cleared.

Interrupt can also be forced by setting a bit in the interrupt force register. In this case, the IU acknowledgement will clear the force bit rather than the pending bit.

After reset, the interrupt mask register is set to all zeros while the remaining control registers are undefined.

Note that interrupt 15 cannot be maskable by the integer unit and should be used with care - most operating system do safely handle this interrupt.

#### 3.3.2.2 Interrupt assignment

The following table shows the assignment of interrupts.

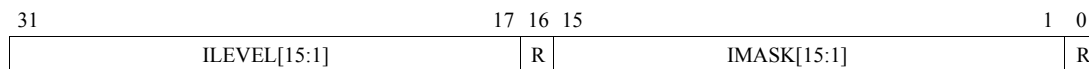
Interrupt	Source
15	Parallel I/O [7]
14	SpaceWire 1

Interrupt	Source
13	SpaceWire 0 Parallel I/O [6]
12	CAN interface Parallel I/O [5]
11	DSU trace buffer
10	Second interrupt controller Parallel I/O [4]
9	Timer 2
8	Timer 1
7	Parallel I/O[3]
6	Parallel I/O[2]
5	Parallel I/O[1]
4	Parallel I/O[0]
3	UART 1
2	UART 2
1	AHB error

**Table 3-7. Interrupt assignments**

### 3.3.2.3 Control registers

The operation of the interrupt controller is programmed through the following registers:

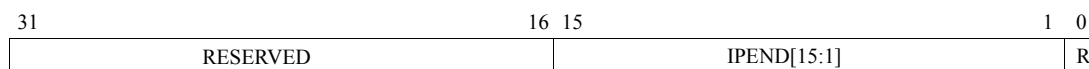


**Figure 3-7. Interrupt mask and priority register**

[31:17]: Interrupt level (ILEVEL[15:1]) - indicates whether an interrupt belongs to priority level 1 (ILEVEL[n]=1) or level 0 (ILEVEL[n]=0).

[15:1]: Interrupt mask (IMASK[15:1]) - indicates whether an interrupt is masked (IMASK[n]=0) or enabled (IMASK[n]=1).

[16], [0]: Reserved. No effect when written to. Undefined when read.

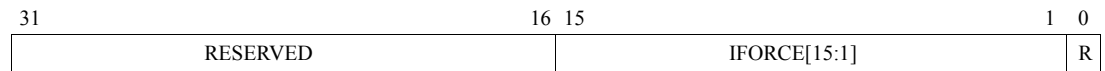


**Figure 3-8. Interrupt pending register**

[15:1]: Interrupt pending (IPEND[15:1]) - indicates whether an interrupt is pending (IPEND[n]=1).

[31:16], [0]: Reserved. No effect when written to. Undefined when read.

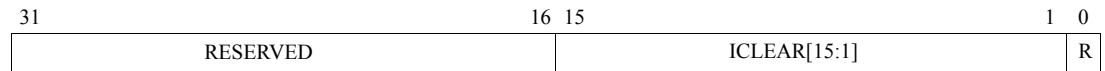




**Figure 3-9. Interrupt force register**

[15:1]: Interrupt force (IFORCE[15:1]) - indicates whether an interrupt is being forced (IFORCE[n]=1).

[31:16], [0]: Reserved. No effect when written to. Undefined when read.



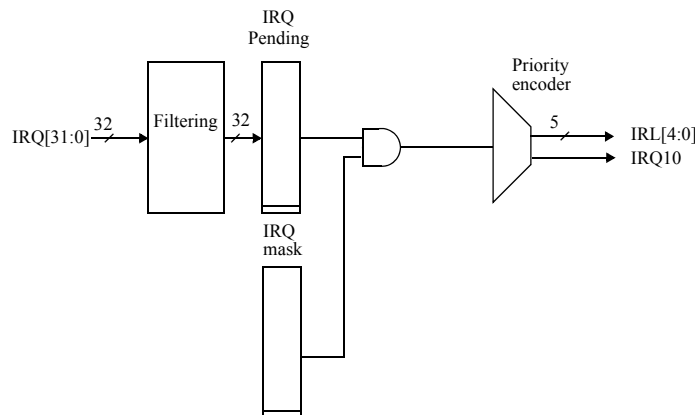
**Figure 3-10. Interrupt clear register**

[15:1]: Interrupt clear (ICLEAR[15:1]) - if written with a '1', will clear the corresponding bit(s) in the interrupt pending register. A read returns zero.

[31:16], [0]: Reserved. No effect when written to. Undefined when read.

### 3.3.3 Secondary interrupt controller

The secondary interrupt controller is used add up to 32 additional interrupts, to be used by on-chip units in system-on-chip designs. Here is a block diagram of the secondary interrupt controller.



**Figure 3-11. Secondary interrupt controller block diagram**

#### 3.3.3.1 Operation

The incoming interrupt signals are filtered. The filtering condition is positive edge-triggered. When the condition is fulfilled, the corresponding bit is set in the interrupt pending register. The pending bits are ANDed with the interrupt mask register and then forwarded to the priority selector. If at least one unmasked pending interrupt exists, the interrupt output will be driven, generating interrupt 10 (by default). The highest unmasked pending interrupt can be read from the interrupt status register (see below).

Interrupts are not cleared automatically upon a taken interrupt - the interrupt handler must reset the pending bit by writing a '1' to the corresponding bit in the interrupt clear register. It must then also clear interrupt 10 in the primary interrupt controller. Testing of interrupts can be done by writing directly to the interrupt pending registers. Bits written with '1' will be set while bits written with '0' will keep their previous value.

After reset, the interrupt mask register is set to all zeros while the remaining control registers are undefined.

3.3.3.2 Control registers

The operation of the secondary interrupt controller is programmed through the following registers:

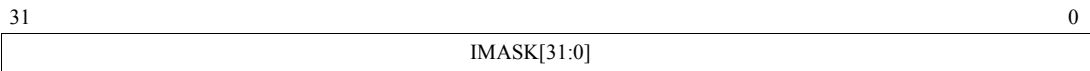


Figure 3-12. Secondary interrupt mask register

[31:0]: Interrupt mask - indicates whether an interrupt is masked (IMASK[n]=0) or enabled (IMASK[n]=1).

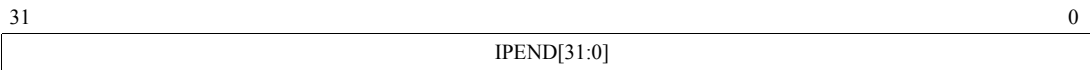


Figure 3-13. Secondary interrupt pending register

[31:0]: Interrupt pending - indicates whether an interrupt is pending (IPEND[n]=1).

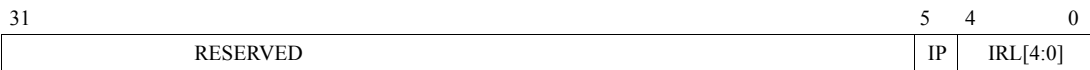


Figure 3-14. Secondary interrupt status register

[4:0]: Interrupt request level - indicates the highest unmasked pending interrupt.  
[5]: Interrupt pending - if set, then IRL is valid. If cleared, no unmasked interrupt is pending.



Figure 3-15. Secondary interrupt clear register

[31:0]: Interrupt clear - if written with a '1', will clear the corresponding bit(s) in the interrupt pending register.

3.3.3.3 Interrupt assignment

Next table shows the assignment of interrupts for the secondary interrupt controller.

Interrupt	Source	
31	GPIO / Gpio[23]	24-bit GPIO input interrupt
30	GPIO / Gpio[22]	24-bit GPIO input interrupt
29	GPIO / Gpio[21]	24-bit GPIO input interrupt

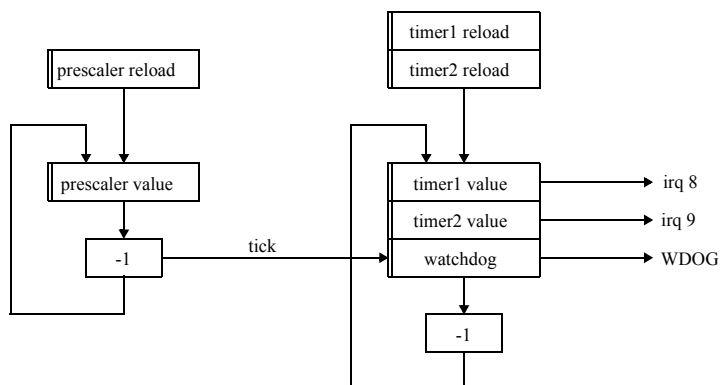
Interrupt	Source	
28	GPIO / Gpio[20]	24-bit GPIO input interrupt
27	GPIO / Gpio[19]	24-bit GPIO input interrupt
26	GPIO / Gpio[18]	24-bit GPIO input interrupt
25	GPIO / Gpio[17]	24-bit GPIO input interrupt
24	GPIO / Gpio[16]	24-bit GPIO input interrupt
23-20	-	Unused
19	CAN/RxSync	Synchronization message received
18	CAN/TxSync	Synchronization message transmitted
17	CAN/IRQ	Common output from interrupt handler
16	SpaceWire 1/ Tick	Synchronization received
15	SpaceWire 1 / Interrupt	Common output from interrupt handler
14	SpaceWire 0 / Tick	Synchronization received
13	SpaceWire 0 / Interrupt	Common output from interrupt handler
12	FIFO/RxParity	Parity error during reception
11	FIFO/RxError	AHB access error during reception
10	FIFO/RxFull	Circular reception buffer full
9	FIFO/RxIrq	Successful reception of data block
8	FIFO/TxError	AHB access error during transmission
7	FIFO/TxEmpty	Circular transmission buffer empty
6	FIFO/TxIrq	Successful transmission of data block
5	ADC/DAC	DAC conversion ready
4	ADC/DAC	ADC conversion ready
3	32-Bit Timer/Timer 2	Timer expired
2	32-Bit Timer/Timer 1	Timer expired
1	GPIO/PULSE	Pulse command completed
0	-	Unused

**Table 3-8. Secondary interrupt controller assignments**

- Notes:
1. Interrupt 17, 15 and 13 are available in primary interrupt controller and should therefore be used restrictively in the secondary interrupt controller. The secondary interrupt controller uses edge detection, whereas the aforementioned interrupt sources use level. The interrupt handling software must thus ensure that the sources for the aforementioned interrupts do not have an additional pending interrupt when clearing the corresponding bit in the pending interrupt register in the secondary interrupt controller. This limitation does not exist for the primary interrupt controller.
  2. Interrupts 31 down to 24 are connected to the inputs of the 24-bit General Purpose Input Output interface. The secondary interrupt controller uses edge detection. The 24-bit General Purpose Input Output interface must therefore only be programmed for edge detection, not for level, to ensure that multiple interrupts can be detected.

### 3.3.4 Timer unit

The timer unit implements two 32-bit timers, one 32-bit watchdog and one 10-bit shared prescaler.



**Figure 3-16. Timer unit block diagram**

#### 3.3.4.1 Operation

The prescaler is clocked by the system clock and decremented on each clock cycle. When the prescaler underflows, it is reloaded from the prescaler reload register and a timer tick is generated for the two timers and watchdog. The effective division rate is therefore equal to prescaler reload register value + 1.

The operation of the timers is controlled through the timer control register. A timer is enabled by setting the enable bit in the control register. The timer value is then decremented each time the prescaler generates a timer tick. When a timer underflows, it will automatically be reloaded with the value of the timer reload register if the reload bit is set, otherwise it will stop (at 0xffffffff) and reset the enable bit. An interrupt will be generated after each underflow.

The timer can be reloaded with the value in the reload register at any time by writing a 'one' to the load bit in the control register.

The watchdog operates similar to the timers, with the difference that it is always enabled and upon underflow asserts the external signal WDOG. This signal can be used to generate a system reset.

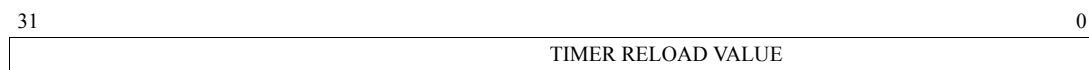
To minimise complexity, the two timers and watchdog share the same decremter. This means that the minimum allowed prescaler division factor is 4 (reload register = 3).

### 3.3.4.2 Registers

The following figures show the layout of the timer unit registers.



**Figure 3-17. Timer 1/2 and Watchdog counter registers**



**Figure 3-18. Timer 1/2 reload registers**

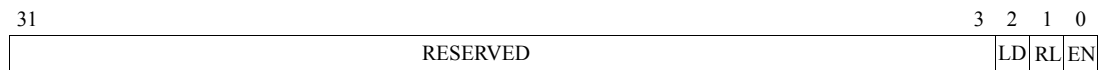


Figure 3-19. Timer 1/2 control registers

- [2]: Load counter (LD) - when written with 'one', will load the timer reload register into the timer counter register. Always reads as a 'zero'.
- [1]: Reload counter (RL) - if RL is set, then the counter will automatically be reloaded with the reload value after each underflow.
- [0]: Enable (EN) - enables the timer when set.
- [31:3]: Reserved. No effect when written to. Undefined when read.

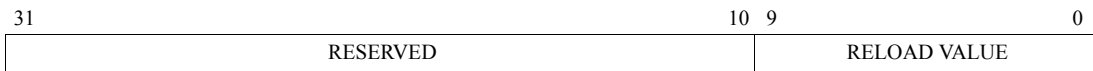


Figure 20.

Figure 3-20. Prescaler reload register

- [31:10]: Reserved. No effect when written to. Undefined when read.

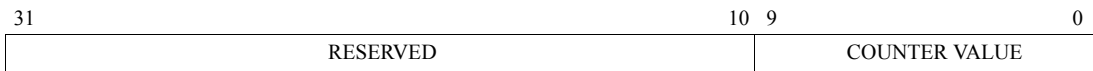


Figure 21.

Figure 3-21. Prescaler counter register

- [31:10]: Reserved. No effect when written to. Undefined when read.

3.3.5 UARTs

Two identical UARTs are provided for serial communications. The UARTs support data frames with 8 data bits, one optional parity bit and one stop bit. To generate the bit-rate, each UART has a programmable 12-bits clock divider. Hardware flow-control is supported through the RTSN/CTSN hand-shake signals. Here is a block diagram of the UART module.

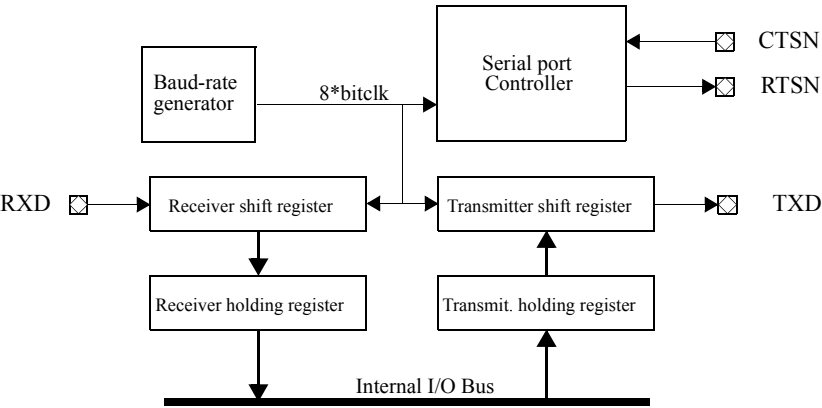


Figure 3-22. UART block diagram

3.3.5.1 Transmitter operation

The transmitter is enabled through the TE bit in the UART control register. When ready to transmit, data is transferred from the transmitter holding register to the transmitter shift register and converted to a serial stream on the transmitter serial output pin (TXD). It automatically sends a start bit followed by eight data bits, an optional parity bit, and one stop bits. The least significant bit of the data is sent first

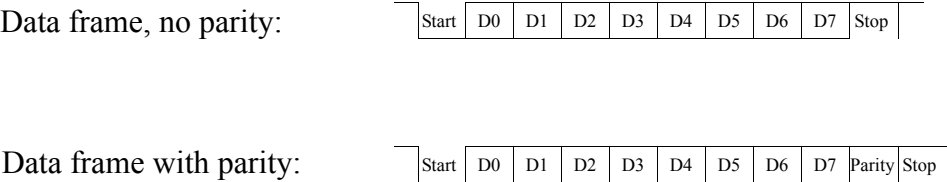


Figure 3-23. UART data frames

Following the transmission of the stop bit, if a new character is not available in the transmitter holding register, the transmitter serial data output remains high and the transmitter shift register empty bit (TSRE) will be set in the UART control register. Transmission resumes and the TSRE is cleared when a new character is loaded in the transmitter holding register. If the transmitter is disabled, it will continue operating until the character currently being transmitted is completely sent out. The transmitter holding register cannot be loaded when the transmitter is disabled.

If flow control is enabled, the CTSN input must be low in order for the character to be transmitted. If it is deasserted in the middle of a transmission, the character in the shift register is transmitted and the transmitter serial output then remains inactive until CTSN is asserted again. If the CTSN is connected to a receivers RTSN, overrun can effectively be prevented.

3.3.5.2 Receiver operation

The receiver is enabled for data reception through the receiver enable (RE) bit in the UART control register. The receiver looks for a high to low transition of a start bit on the receiver serial data input pin. If a transition is detected, the state of the serial input is sampled a half bit clocks later. If the serial input is sampled high the start bit is invalid and the search for a valid start bit continues. If the serial input is still low, a valid start bit is assumed and the receiver continues to sample the serial input at one bit time intervals (at the theoretical centre of the bit) until the proper number of data bits and the parity bit have been assembled and one stop bit has been detected. The serial

input is shifted through an 8-bit shift register where all bits have to have the same value before the new value is taken into account, effectively forming a low-pass filter with a cut-off frequency of 1/8 system clock.

During reception, the least significant bit is received first into the receiver shift register (RSR). The data is then transferred to the receiver holding register (RHR) and the data ready (DR) bit is set in the UART status register. If RHR was not empty when a character was received, the transfer from RSR, and setting of DR, will not occur until RHR has been emptied by a read access to the UART data register. The parity, framing, break and overrun error bits are set at the received character boundary, at the same time as the data ready (DR) bit would have been set, but no new character is transferred to RHR.

The parity error (PE) bit is set or cleared for each received character. The parity error (PE) bit is also cleared when a '0' is written to it via the UART status register.

The break (BR) bit is set when all-zero bits have been received and the stop bit is zero. The break (BR) bit is not cleared when a new character has been received, it is cleared when a '0' is written to it via the UART status register.

The framing error (FE) bit is set when any non-zero bits have been received and the stop bit is zero. The framing error (FE) bit is not cleared when a new character has been received, it is cleared when a '0' is written to it via the UART status register.

If both receiver holding (RHR) and shift (RSR) registers contain an un-read character when a new start bit is detected, then the character held in the receiver shift register (RSR) will be lost and the overrun (OV) bit will be set in the UART status register. The overflow bit (OV) is not cleared when a new character has been received, it is cleared when a '0' is written to it via the UART status register.

If flow control is enabled, then the RTSN will be negated (high) when a valid start bit is detected and the receiver holding register contains an un-read character. When the holding register is read, the RTSN will automatically be reasserted again.

### 3.3.5.3 Baud-rate generation

Each UART contains a 12-bit down-counting scaler to generate the desired baud-rate. The scaler is clocked by the system clock and generates a UART tick each time it underflows. The scaler is reloaded with the value of the UART scaler reload register after each underflow. The resulting UART tick frequency should be 8 times the desired baud-rate. If the EC bit is set, the scaler will be clocked by the LeonPio[3] input rather than the system clock. In this case, the frequency of LeonPio[3] must be less than half the frequency of the system clock.

### 3.3.5.4 Loop back mode

If the LB bit in the UART control register is set, the UART will be in loop back mode. In this mode, the transmitter output is internally connected to the receiver input and the RTSN is connected to the CTSN. It is then possible to perform loop back tests to verify operation of receiver, transmitter and associated software routines. In this mode, the outputs remain in the inactive state, in order to avoid sending out data.

### 3.3.5.5 Interrupt generation

The UART will generate an interrupt under the following conditions: when the transmitter is enabled, the transmitter interrupt is enabled and the transmitter holding register moves from full to empty; when the receiver is enabled, the receiver interrupt is enabled and the receiver holding register moves from empty to full; when the receiver is enabled, the receiver interrupt is enabled and a character with either parity, framing, break or overrun error is received.

### 3.3.5.6 UART registers

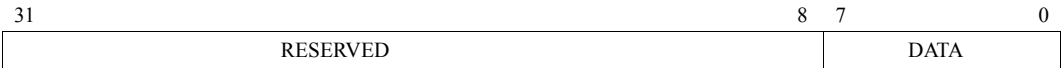
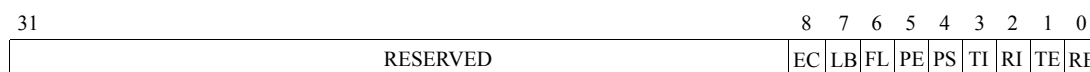


Figure 3-24. UART data register

- [7:0] : Receiver holding register (read access)
- [7:0] : Transmitter holding register (write access)
- [31:8]: Reserved. No effect when written to. Undefined when read.



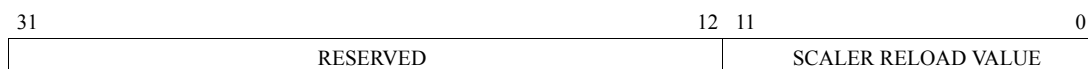
**Figure 3-25. UART control register**

- 0: Receiver enable (RE) - if set, enables the receiver.
- 1: Transmitter enable (TE) - if set, enables the transmitter.
- 2: Receiver interrupt enable (RI) - if set, enables generation of receiver interrupt.
- 3: Transmitter interrupt enable (TI) - if set, enables generation of transmitter interrupt.
- 4: Parity select (PS) - selects parity polarity (0 = even parity, 1 = odd parity)
- 5: Parity enable (PE) - if set, enables parity generation and checking.
- 6: Flow control (FL) - if set, enables flow control using CTS/RTS.
- 7: Loop back (LB) - if set, loop back mode will be enabled.
- 8: External Clock (EC) - if set, the UART scaler will be clocked by LeonPio[3]
- [31:9]: Reserved. No effect when written to. Undefined when read.



**Figure 3-26. UART status register**

- 0: Data ready (DR) - indicates that new data is available in the receiver holding register.
- 1: Transmitter shift register empty (TS) - indicates that the transmitter shift register is empty.
- 2: Transmitter hold register empty (TH) - indicates that the transmitter hold register is empty.
- 3: Break received (BR) - indicates that a BREAK has been received.
- 4: Overrun (OV) - indicates that one or more character have been lost due to overrun.
- 5: Parity error (PE) - indicates that a parity error was detected.
- 6: Framing error (FE) - indicates that a framing error was detected.
- [31:7]: Reserved. No effect when written to. Undefined when read.



**Figure 3-27. UART scaler reload register**

- [31:12]: Reserved. No effect when written to. Undefined when read.





EN - Enable. If set, the corresponding interrupt will be enabled, otherwise it will be masked.

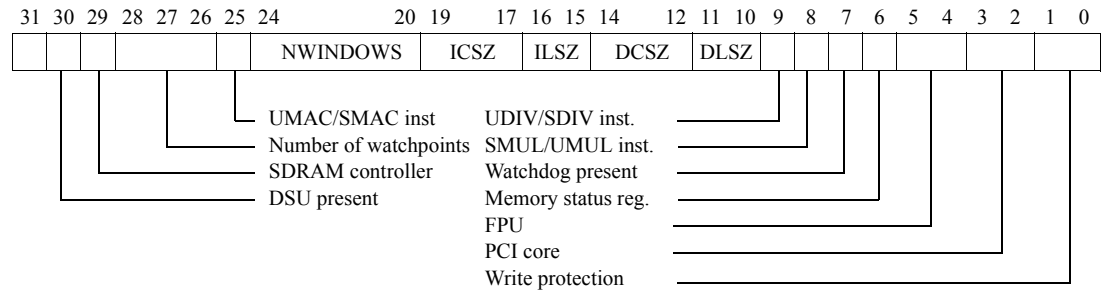
To save pins, I/O pins are shared with other functions according to the table below:

I/O port	Function	Type	Description	Enabling condition
LeonPio[15]	TXD1	Output	UART1 transmitter data	UART1 transmitter enabled
LeonPio[14]	RXD1	Input	UART1 receiver data	-
LeonPio[13]	RTS1	Output	UART1 request-to-send	UART1 flow-control enabled
LeonPio[12]	CTS1	Input	UART1 clear-to-send	-
LeonPio[11]	TXD2	Output	UART2 transmitter data	UART2 transmitter enabled
LeonPio[10]	RXD2	Input	UART2 receiver data	-
LeonPio[9]	RTS2	Output	UART2 request-to-send	UART2 flow-control enabled
LeonPio[8]	CTS2	Input	UART2 clear-to-send	-
LeonPio[3]	UART clock	Input	Use as alternative UART clock	-
LeonPio[2]	Prom EDAC	Input	Defines prom edac at boot time	-
LeonPio[1:0]	Prom width	Input	Defines prom width at boot time	-

**Table 3-9. UART/IO port usage**

### 3.3.7 LEON configuration register

Since LEON is extensively configurable, the LEON configuration register (read-only) is used to indicate which options were enabled in the design. For each option present, the corresponding register bit is hardwired to '1'.



**Figure 3-31. LEON configuration register**

- [31]: Reserved. No effect when written to. Undefined when read.
- [30]: Debug support unit (0=disabled, 1=present)
- [29]: SDRAM controller present (0=disabled, 1=present)
- [28:26]: Number of implemented watchpoints (0 - 4)
- [25]: UMAC/SMAC instruction implemented
- [24:20]: Number of register windows. The implemented number of SPARC register windows - 1.
- [19:17]: Instruction cache size. The size (in kbytes) of the instruction cache. Cache size =  $2^{\text{ICSZ}}$ .
- [16:15]: Instruction cache line size. The line size (in 32-bit words) of each line. Line size =  $2^{\text{ILSZ}}$ .
- [14:12]: Data cache size. The size (in kbytes) of the data cache. Cache size =  $2^{\text{DCSZ}}$ .
- [11:10]: Data cache line size. The line size (in 32-bit words) of each line. Line size =  $2^{\text{DLSZ}}$ .
- [9]: UDIV/SDIV instruction implemented

- [8]: UMUL/SMUL instruction implemented
- [7]: Watchdog implemented
- [6]: Memory status and failing address register present
- [5:4]: FPU type (00 = none, 01=Meiko)
- [3:2]: PCI core type (00=none, 01=InSilicon, 10=ESA, 11=other)
- [1:0]: Write protection type (00=none, 01=standard)

### 3.3.8 Power-down

The processor can be powered-down by writing (an arbitrary) value to the power-down register. Power-down mode will be entered on the next load or store instruction. To enter power-down mode immediately, a store to the power-down register should be performed followed by a 'dummy' load. During power-down mode, the integer unit will effectively be halted. The power-down mode will be terminated (and the integer unit re-enabled) when an unmasked interrupt with higher level than the current processor interrupt level (PIL) becomes pending. All other functions and peripherals operate as nominal during the power-down mode. A suitable power-down routine could be:

```
struct pwd_reg_type { volatile int pwd; };

power_down()
{
    struct pwd_reg_type *lreg = (struct pwd_reg_type *) 0x80000018;
    while (1) lreg->pwd = lreg->pwd;
}

In assembly, a suitable sequence could be:
power_down:
    set 0x80000000, %13
    st  %g0, [%13 + 0x18]
    ba  power_down
    ld  [%13 + 0x18], %g0
```

### 3.3.9 AHB status register

Any access triggering an error response on the AHB bus will be registered in two registers; AHB failing address register and AHB status register. The failing address register will store the address of the access while the AHB status register will store the access and error types. The registers are updated when an error occur, and the EV (error valid) is not set. When the EV bit is set, interrupt 1 is generated to inform the processor about the error. After an error, the EV bit has to be reset by software.

The following figure shows the layout of the AHB status register.



**Figure 3-32. AHB status register**

- [9]: EE - EDAC correctable error. Set when a correctable EDAC error is detected.
- [8]: EV - error valid. Set when an error occurred.
- [7]: RW - Read/Write. This bit is set if the failed access was a read cycle, otherwise it is cleared.
- [6:3]: HMASTER - AHB master. This field contains the HMASTER[3:0] of the failed access.
- [2:0]: HSIZE - transfer size. This field contains the HSIZE[2:0] of the failed transfer.
- [31:9]: Reserved. No effect when written to. Undefined when read.

## 3.4 External memory access

### 3.4.1 Memory interface

The memory bus provides a direct interface to PROM, memory mapped I/O devices, asynchronous static ram (SRAM). Chip-select decoding is done for two PROM banks, one I/O bank, and four SRAM banks. The figure presented here after shows how the connection to the different device types is made.

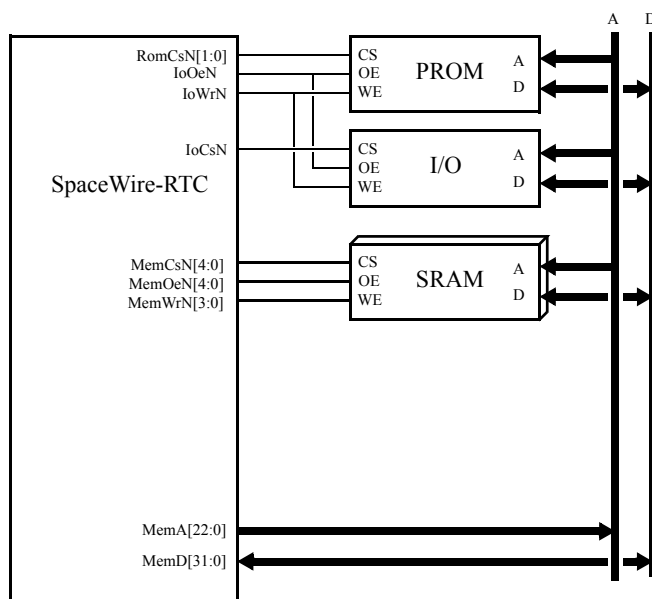


Figure 3-33. Memory device interface

### 3.4.2 Memory controller

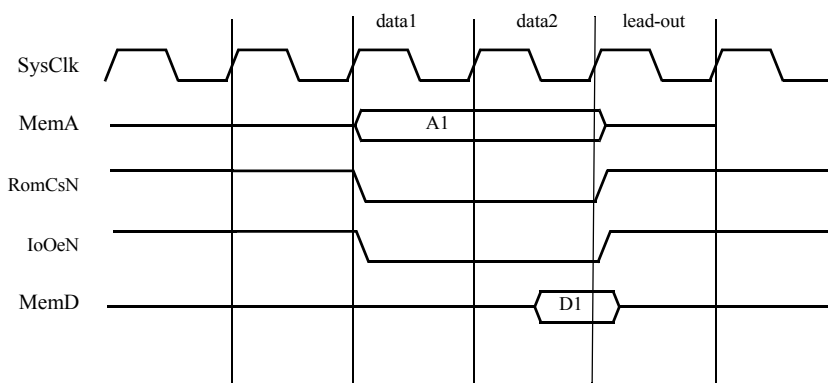
The external memory bus is controlled by a programmable memory controller. The controller acts as a slave on the AHB bus. The function of the memory controller is programmed through memory configuration registers 1, 2 & 3 (MCFG1, MCFG2 & MCFG3) through the APB bus. The memory bus supports various types of devices: prom, sram and local I/O. The memory bus can also be configured in 8-bit mode for applications with low memory and performance demands. The controller decodes a 2 Gbyte address space, divided according to the following table:

Address range	Size	Mapping
0x00000000 - 0x1FFFFFFF	512 M	Prom
0x20000000 - 0x3FFFFFFF	512M	I/O
0x40000000 - 0x7FFFFFFF	1 G	SRAM

Table 3-10. Memory controller address map

### 3.4.3 PROM access

Accesses to PROM have the same timing as RAM accesses, the differences being that PROM cycles can have up to 30 waitstates.

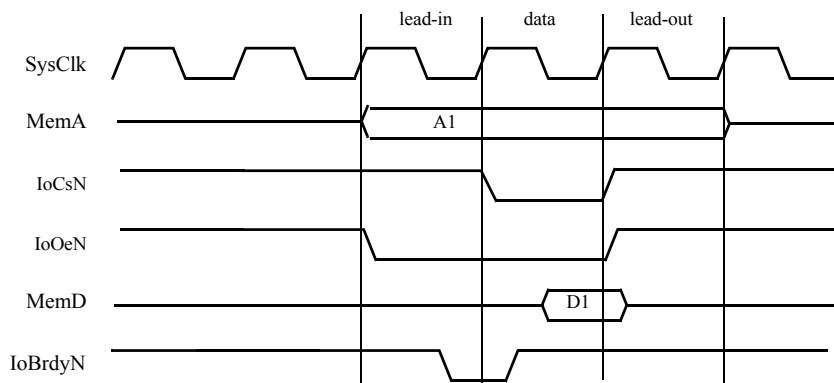


**Figure 3-34. Prom read cycle**

Two PROM chip-select signals are provided, RomCsN[1:0]. RomCsN[0] is asserted when the lower half (0 - 0x10000000) of the PROM area as addressed while RomCsN[1] is asserted for the upper half (0x10000000 - 0x20000000).

### 3.4.4 Memory mapped I/O

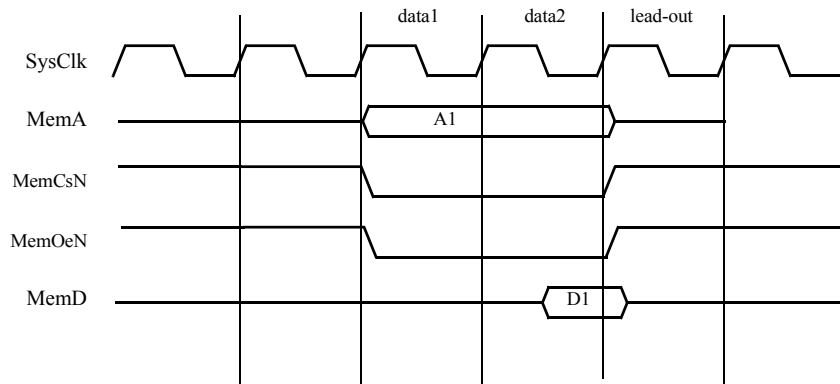
Accesses to I/O have similar timing to ROM/RAM accesses, the differences being that a additional waitstates can be inserted by de-asserting the IoBrdyN signal. The I/O select signal (IoCsN) is delayed one clock to provide stable address before IoCsN is asserted.



**Figure 3-35. I/O read cycle**

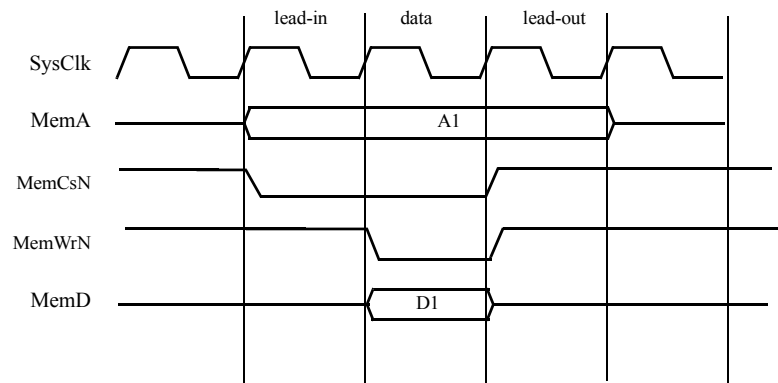
### 3.4.5 SRAM access

The SRAM area can be up to 32 MByte, divided on up to four RAM banks. The size of banks 1-4 (MemCsN[3:0]) is programmed in the RAM bank-size field (MCFG2[12:9]) and can be set in binary steps from 8 kbyte to 32 Mbyte. A read access to SRAM consists of two data cycles and between zero and three waitstates. On non-consecutive accesses, a lead-out cycle is added after a read cycle to prevent bus contention due to slow turn-off time of memories or I/O devices. Next figure presents the basic read cycle waveform (zero waitstate).



**Figure 3-36. Static ram read cycle (0-waitstate)**

For read accesses to MemCsN[3:0], a separate output enable signal (MemOeN[n]) is provided for each RAM bank and only asserted when that bank is selected. A write access is similar to the read access but takes a minimum of three cycles:



**Figure 3-37. Static ram write cycle**

Each byte lane has an individual write strobe to allow efficient byte and half-word writes. If the memory uses a common write strobe for the full 32-bit data, the read-modify-write bit MCFG2 should be set to enable read-modify-write cycles for sub-word writes.

### 3.4.6 Burst cycles

To improve the bandwidth of the memory bus, accesses to consecutive addresses can be performed in burst mode. Burst transfers will be generated when the memory controller is accessed using an AHB burst request. These includes instruction cache-line fills, double loads and double stores. The timing of a burst cycle is identical to the programmed basic cycle with the exception that during read cycles, the lead-out cycle will only occurs after the last transfer.

### 3.4.7 8-bit PROM and SRAM access

To support applications with low memory and performance requirements efficiently, it is not necessary to always have full 32-bit memory banks. The SRAM and PROM areas can be individually configured for 8-bit operation by programming the ROM and RAM size fields in the memory configuration registers. Since read access to memory is always done on 32-bit word basis, read access to 8-bit memory will

be transformed in a burst of four read cycles. During writes, only the necessary bytes will be written. here is an interface example with 8-bit PROM and 8-bit SRAM.

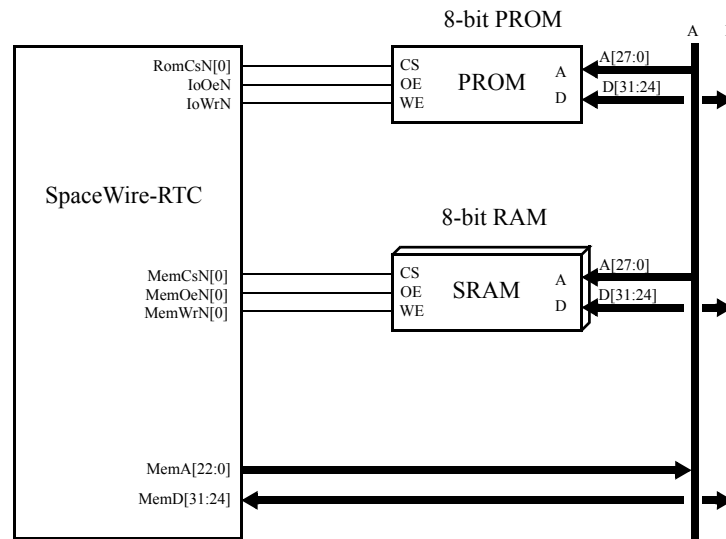


Figure 3-38. 8-bit memory interface example

### 3.4.8 8-bit I/O access

Similar to the PROM/RAM areas, the I/O area can also be configured to 8-bit mode. However, the I/O device will NOT be accessed by multiple 8 bits accesses as the memory areas, but only with one single access just as in 32-bit mode. To accesses an I/O device on a 8-bit bus LDUB/STB instructions should be used.

### 3.4.9 Memory EDAC

The memory controller in LEON2-FT is provided with an EDAC that can correct one error and detect two errors in a 32-bit word. For each word, a 7-bit checksum is generated according to the equations below. Correction is done on-the-fly and no timing penalty occurs during correction. If an un-correctable error (double-error) is detected, an memory exception is signalled to the IU. If a correctable error occurs, no exception is generated but the event is registered in the failing address and memory status register and interrupt 1 is generated. The interrupt can then be attached to a low priority interrupt handler that scrubs the failing memory location. The EDAC can be used during access to PROM or RAM areas by setting the corresponding EDAC enable bits in the Error control register (see below). The equations below show how the EDAC checkbits are generated:

$$CB0 = D0 \wedge D4 \wedge D6 \wedge D7 \wedge D8 \wedge D9 \wedge D11 \wedge D14 \wedge D17 \wedge D18 \wedge D19 \wedge D21 \wedge D26 \wedge D28 \wedge D29 \wedge D31$$

$$CB1 = D0 \wedge D1 \wedge D2 \wedge D4 \wedge D6 \wedge D8 \wedge D10 \wedge D12 \wedge D16 \wedge D17 \wedge D18 \wedge D20 \wedge D22 \wedge D24 \wedge D26 \wedge D28$$

$$CB2 = D0 \wedge D3 \wedge D4 \wedge D7 \wedge D9 \wedge D10 \wedge D13 \wedge D15 \wedge D16 \wedge D19 \wedge D20 \wedge D23 \wedge D25 \wedge D26 \wedge D29 \wedge D31$$

$$CB3 = D0 \wedge D1 \wedge D5 \wedge D6 \wedge D7 \wedge D11 \wedge D12 \wedge D13 \wedge D16 \wedge D17 \wedge D21 \wedge D22 \wedge D23 \wedge D27 \wedge D28 \wedge D29$$

$$CB4 = D2 \wedge D3 \wedge D4 \wedge D5 \wedge D6 \wedge D7 \wedge D14 \wedge D15 \wedge D18 \wedge D19 \wedge D20 \wedge D21 \wedge D22 \wedge D23 \wedge D30 \wedge D31$$

$$CB5 = D8 \wedge D9 \wedge D10 \wedge D11 \wedge D12 \wedge D13 \wedge D14 \wedge D15 \wedge D24 \wedge D25 \wedge D26 \wedge D27 \wedge D28 \wedge D29 \wedge D30 \wedge D31$$

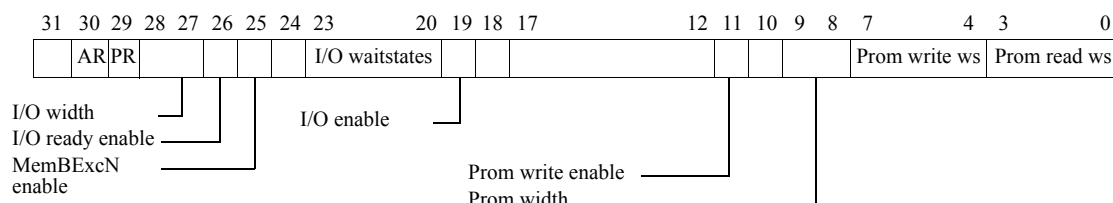
$$CB6 = D0 \wedge D1 \wedge D2 \wedge D3 \wedge D4 \wedge D5 \wedge D6 \wedge D7 \wedge D24 \wedge D25 \wedge D26 \wedge D27 \wedge D28 \wedge D29 \wedge D30 \wedge D31$$

If the memory is configured in 8-bit mode, the EDAC checkbit bus (CB[7:0]) is not used but it is still possible to use EDAC protection. Data is always accessed as words (4 bytes at a time) and the corresponding checkbits are located at the address acquired by inverting the word address (address[27:2]) and using it as a byte address. The same chip-select is kept active. A word written as four bytes to addresses 0, 1, 2, 3 will have its checkbits at address 0x0FFFFFFF, addresses 4, 5, 6, 7 at 0x0FFFFFFE and so on. All the bits up to the maximum banksize will be inverted while the same chip-select is always asserted. This way all the banksize can be supported and no memory will be unused (except for a maximum of 4 B in the gap between the data and checkbit area). The 8-bit mode applies to RAM and PROM. Only byte-writes should be performed to ROM with EDAC enabled. In this case, only the corresponding byte will be written.

The operation of the EDAC can be tested through the Error control register (see below). If the WB (write bypass) bit is set, the value in the TCB field will replace the normal checkbits during memory write cycles. If the RB (read bypass) is set, the memory checkbits of the loaded data will be stored in the TCB field during memory read cycles. NOTE: when the EDAC is enabled, the RMW bit in memory configuration register 2 must be set.

### 3.4.10 Memory configuration register 1 (MCFG1)

Memory configuration register 1 is used to program the timing of rom and local I/O accesses.



**Figure 3-39. Memory configuration register 1**

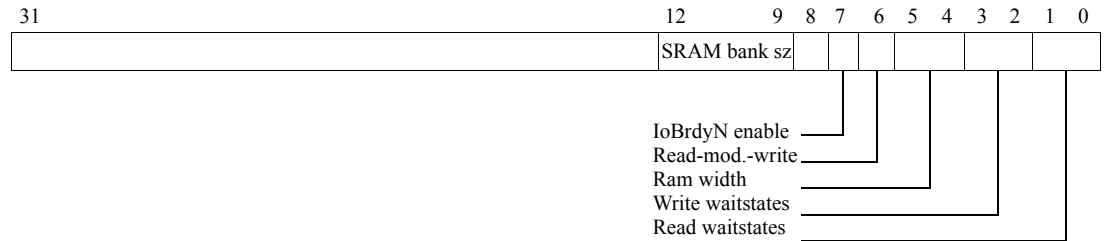
- [3:0]: Prom read waitstates. Defines the number of waitstates during prom read cycles ("0000"=0, "0001"=2, "0010"=4, ... "1111"=30).
- [7:4]: Prom write waitstates. Defines the number of waitstates during prom write cycles ("0000"=0, "0001"=2, "0010"=4, ... "1111"=30).
- [9:8]: Prom width. Defines the data width of the prom area ("00"=8, "10"=32).
- [10]: Reserved
- [11]: Prom write enable. If set, enables write cycles to the prom area.
- [17:12]: Reserved
- [19]: I/O enable. If set, the access to the memory bus I/O area are enabled.
- [23:20]: I/O waitstates. Defines the number of waitstates during I/O accesses ("0000"=0, "0001"=1, "0010"=2, ..., "1111"=15).
- [25]: Bus error (MemBExcN) enable.
- [26]: Bus ready (IoBrdyN) enable.
- [28:27]: I/O bus width. Defines the data width of the I/O area ("00"=8, "10"=32).
- [29]: Asynchronous bus ready (ABRDYN). If set, the IoBrdyN input can be asserted without relation to the system clock. Reset to '0' at power-up.
- [30]: PROM area bus ready enable (PBRDYN). If set, a PROM access will be extended until IoBrdyN is asserted. Reset to '0' at power-up.
- [18], [24], [31]: Reserved. No effect when written to. Undefined when read.

During power-up, the prom width (bits [9:8]) are set with value on LeonPio[1:0] inputs. The PROM waitstates fields are set to 30 (maximum). External bus error and bus ready are disabled. All other fields are undefined.

### 3.4.11 Memory configuration register 2 (MCFG2)

Memory configuration register 2 is used to control the timing of the SRAM.



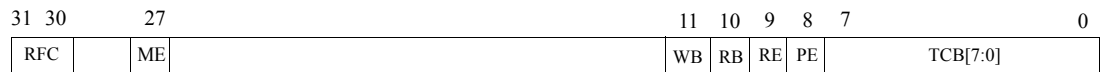


**Figure 3-40. Memory configuration register 2**

- [1:0]: Ram read waitstates. Defines the number of waitstates during ram read cycles ("00"=0, "01"=1, "10"=2, "11"=3).
- [3:2]: Ram write waitstates. Defines the number of waitstates during ram write cycles ("00"=0, "01"=1, "10"=2, "11"=3).
- [5:4]: Ram with. Defines the data with of the ram area ("00"=8, "1X"= 32).
- [6]: Read-modify-write. Enable read-modify-write cycles on sub-word writes to 32-bit areas with common write strobe (no byte write strobe).
- [7]: Bus ready enable. If set, will enable IoBrdyN for ram area. Unused.
- [12:9]: Ram bank size. Defines the size of each ram bank ("0000"=8 Kbyte, "0001"=16 Kbyte... "1111"=256 Mbyte).
- [8], [31:13]: Reserved. No effect when written to. Undefined when read.

### 3.4.12 Memory configuration register 3 (MCFG3)

MCFG3 contains the control and monitor the memory EDAC. It also contains the configuration of the register file EDAC.



**Figure 3-41. Memory configuration register 3**

- [31:30]: Regfile check bits (RFC) - Indicates how many checkbits are used for the register file (00=none, 01=1, 10=2, 11=7 (EDAC)). Fixed.
- [29:28]: Reserved. No effect when written to. Undefined when read.
- [27]: Memory EDAC (ME) - Indicates if a memory EDAC is present
- [26:12]: Unused. No effect when written to. Undefined when read.
- [11]: WB - EDAC diagnostic write bypass
- [10]: RB - EDAC diagnostic read bypass
- [9]: RAM EDAC enable (RE) - Enable EDAC checking of the RAM area
- [8]: PROM EDAC enable (PE) - Enable EDAC checking of the PROM area. At reset, this bit is initialised with the value of LeonPio[2]
- [7:0]: TCB - Test checkbits. This field replaces the normal checkbits during store cycles when WB is set. TCB is also loaded with the memory checkbits during load cycles when RB is set.

### 3.4.13 Write protection

Write protection is provided to protect the RAM area against accidental over-writing. It is implemented with two methods: the address/mask method as implemented in the original LEON2 model, and an extended version using start/end addressing.

#### 3.4.13.1 Address/mask write protection

The address/mask write protection is implemented with two block protect units capable of disabling or enabling write access to a binary aligned memory block in the range of 32 Kbyte - 1 Gbyte. Each block protect unit is controlled through a control register (figure 44). The

units operate as follows: on each write access to RAM, address bits (29:15) are xored with the tag field in the control register, and anded with the mask field. A write protection hit is generated if the result is zero, and the corresponding unit is enabled in block protection mode (BP = 1) or if the result is not zero and the unit is enabled in segment mode (BP = 0).

31	30	29		15	14		0
EN	BP		TAG[14:0]			MASK[14:0]	

**Figure 3-42. Write protection register 1 & 2**

- [14:0] Address mask (MASK) - this field contains the address mask
- [29:15] Address tag (TAG) - this field is compared against address(29:15)
- [30] Block protect (BP) - if set, selects block protect mode
- [31] Enable (EN) - if set, enables the write protect unit

### 3.4.13.2 Start/end address write protection

The start/end address write protect scheme contains two identical units that compare the AHB write address against a start and an end address. If operated in block protect mode (BP = 1) and the AHB write address is equal or higher than the start address and lower or equal to the end address, a write protect hit is generated. If operated in segment mode (BP = 0), a write protect hit is generated when the write address is lower than the START address, or higher than the END address.

31	29			1	0
00		START1 [29:2]		BP	0
00		END1 [29:2]		US	SU
00		START2 [29:2]		BP	0
00		END2 [29:2]		US	SU

**Figure 3-43. Start/end address Write protection registers**

- [31:30]: Reserved. No effect when written to. Undefined when read.
- START [29:2] Contains the first address in the protected block
- END [29:2] Contains the last address in the protected block
- BP - Block protect. If set, selects block protect mode
- US - User mode. If set, write protection is enabled for user-mode accesses
- SU - Supervisor mode. If set, write protection is enabled for supervisor-mode access.

The start address is calculated as  $0x40000000 + \text{START} \times 4$ . The end address is calculated as  $0x40000000 + \text{END} \times 4$ .

### 3.4.13.3 Generation of write protection

The results from the two write protection schemes is combined together according to the following scheme:

If all enabled units operate in block protect mode, then a write protect error will be generated if any of the enabled units signal a write protection hit.

If at least one of the enabled units operates in segment mode, then a write protect error will be generated only if all units operating in segment mode signal a write protection hit.

A write protection error will result in that the AHB write cycle is ended with an AHB error response and the data is not written to the memory.

The ROM area can be write protected by clearing the write enable bit MCFG1.

### 3.4.14 Using IoBrdyN

The IoBrdyN signal can be used to stretch access cycles to the I/O area. The accesses will always have at least the pre-programmed number of waitstates as defined in memory configuration registers 1 & 2, but will be further stretched until IoBrdyN is asserted. IoBrdyN should be asserted in the cycle preceding the last one.

If bit 29 in memory configuration register 1 is not set, then BRDYN is sampled synchronously on the rising edge of the system clock and should be asserted in the cycle preceding the last one. If bit 29 is set, the BRDYN can be asserted asynchronously with the system clock. In this case, the read data must be kept stable until the de-assertion of IoOeN.

The use of IoBrdyN can be enabled separately for the I/O area.

### 3.4.15 Access errors

An access error can be signalled by asserting the MemBExcN signal, which is sampled together with the data. If the usage of MemBExcN is enabled in memory configuration register 1, an error response will be generated on the internal AMBA bus. MemBExcN can be enabled or disabled through memory configuration register 1, and is active for all areas (PROM, I/O and RAM).

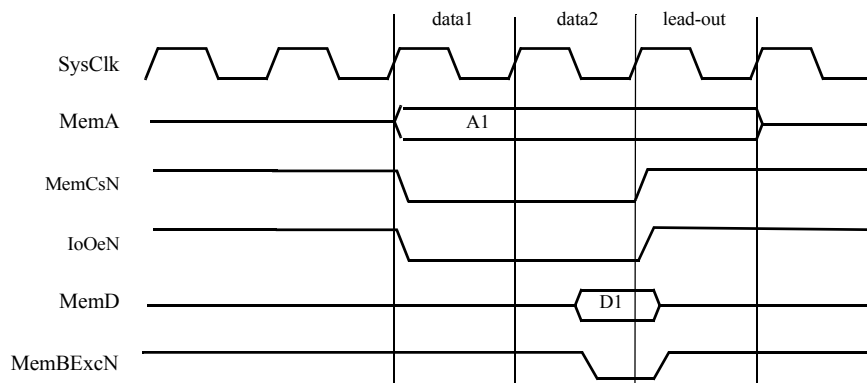
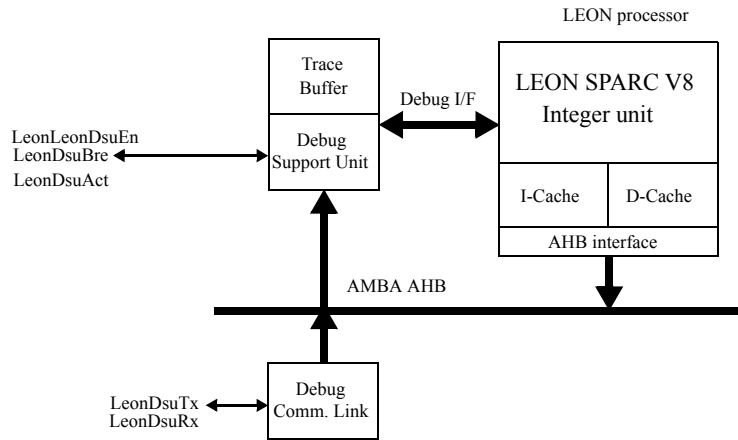


Figure 3-44. Read cycle with MemBExcN

## 3.5 Hardware debug support

### 3.5.1 Overview

The LEON processor includes hardware debug support to aid software debugging on target hardware. The support is provided through two modules: a debug support unit (DSU) and a debug communication link (DCL). The DSU can put the processor in debug mode, allowing read/write access to all processor registers and cache memories. The DSU also contains a trace buffer which stores executed instructions and/or data transfers on the AMBA AHB bus. The debug communications link implements a simple read/write protocol and uses standard asynchronous UART communications (RS232C).



**Figure 3-45. Debug support unit and comm. link**

## 3.5.2 Debug support unit

### 3.5.2.1 Overview

The debug support unit is used to control the trace buffer and the processor debug mode. The DSU is attached to the AHB bus as slave, occupying a 2 Mbyte address space. Through this address space, any AHB master can access the processor registers and the contents of the trace buffer. The DSU control registers can be accessed at any time, while the processor registers and caches can only be accessed when the processor has entered debug mode. The trace buffer can be accessed only when tracing is disabled/completed. In debug mode, the processor pipeline is held and the processor state can be accessed by the DSU. Entering the debug mode can occur on the following events:

- executing a breakpoint instruction (ta 1)
- integer unit hardware breakpoint/watchpoint hit (trap 0xb)
- rising edge of the external break signal (LeonDsuBre)
- setting the break-now (BN) bit in the DSU control register
- a trap that would cause the processor to enter error mode
- occurrence of any, or a selection of traps as defined in the DSU control register
- after a single-step operation
- DSU breakpoint hit

The debug mode can only be entered when the debug support unit is enabled through an external pin (LeonLeonDsuEn). When the debug mode is entered, the following actions are taken:

- PC and nPC are saved in temporary registers (accessible by the debug unit)
- an output signal (LeonDsuAct) is asserted to indicate the debug state
- the timer unit is (optionally) stopped to freeze the LEON timers and watchdog

The instruction that caused the processor to enter debug mode is not executed, and the processor state is kept unmodified. Execution is resumed by clearing the BN bit in the DSU control register or by de-asserting LeonDsuEn. The timer unit will be re-enabled and execution will continue from the saved PC and nPC. Debug mode can also be entered after the processor has entered error mode, for instance when an application has terminated and halted the processor. The error mode can be reset and the processor restarted at any address.

### 3.5.2.2 Trace buffer

The trace buffer consists of a circular buffer that stores executed instructions and/or AHB data transfers. A 30-bit counter is also provided and stored in the trace as time tag. The trace buffer operation is controlled through the DSU control register and the Trace buffer control register (see below). When the processor enters debug mode, tracing is suspended. The size of the trace buffer is 512 lines.

The trace buffer is 128 bits wide, the information stored is indicated in the following tables:

Bits	Name	Definition
127	AHB breakpoint hit	Set to '1' if a DSU AHB breakpoint hit occurred.
126	-	Unused
125:9 6	Time tag	The value of the time tag counter
95:92	IRL	Processor interrupt request input
91:88	PIL	Processor interrupt level (psr.pil)
87:80	Trap type	Processor trap type (psr.tt)
79	Hwrite	AHB HWRITE
78:77	Htrans	AHB HTRANS
76:74	Hsize	AHB HSIZE
73:71	Hburst	AHB HBURST
70:67	Hmaster	AHB HMASTER
66	Hmastlock	AHB HMASTLOCK
65:64	Hresp	AHB HRESP
63:32	Load/Store data	AHB HRDATA or HWDATA
31:0	Load/Store address	AHB HADDR

**Table 3-11. Trace buffer data allocation, AHB tracing mode**

Bits	Name	Definition
127	Instruction breakpoint hit	Set to '1' if a DSU instruction breakpoint hit occurred.
126	Multi-cycle instruction	Set to '1' on the second and third instance of a multi-cycle instruction (LDD, ST or FPOP)
125:9 6	Time tag	The value of the time tag counter
95:64	Load/Store parameters	Instruction result, Store address or Store data
63:34	Program counter	Program counter (2 lsb bits removed since they are always zero)
33	Instruction trap	Set to '1' if traced instruction trapped
32	Processor error mode	Set to '1' if the traced instruction caused processor error mode
31:0	Opcode	Instruction opcode

**Table 3-12. Trace buffer data allocation, Instruction tracing mode**

During instruction tracing, one instruction is stored per line in the trace buffer with the exception of multi-cycle instructions. Multi-cycle instructions are entered two or three times in the trace buffer. For store instructions, bits [63:32] correspond to the store address on the first entry and to the stored data on the second entry (and third in case of STD). Bit 126 is set on the second and third entry to indicate this. A double load (LDD) is entered twice in the trace buffer, with bits [63:32] containing the loaded data. Multiply and divide instructions are entered twice, but only the last entry contains the result. Bit 126 is set for the second entry. For FPU operation producing a double-precision result, the first entry puts the MSB 32 bits of the results in bit [63:32] while the second entry puts the LSB 32 bits in this field. When a trace is frozen, interrupt 11 is generated.

The DSU time tag counter is incremented each clock as long as the processor is running. The counter is stopped when the processor enters debug mode, and restarted when execution is resumed.



**Figure 3-46. Time tag counter**

[31:30]: Reserved. No effect when written to. Undefined when read.

The trace buffer control register contains two counters that contain the next address of the trace buffer to be written. Since the buffer is circular, it actually points to the oldest entry in the buffer. The counters are automatically incremented after each stored trace entry.



**Figure 3-47. Trace buffer control register**

[31:27]: Reserved. No effect when written to. Undefined when read.

[11:0] : Instruction trace index counter

[23:12] : AHB trace index counter

[24] : Trace instruction enable

[25] : Trace AHB enable

[26] : AHB trace buffer freeze. If set, the AHB trace buffer will be frozen when the processor enters debug mode

When both instructions and AHB transfers are traced ('mixed mode tracing'), the buffer is divided on two halves. Instructions are stored in the lower half and AHB transfers in the upper half of the buffer. The MSB bit of the AHB index counter is then automatically kept high, while the MSB of the instruction index counter is kept low. When the AF bit in the trace control register is set, AHB tracing is stopped when the processor is in debug mode. When AF is cleared, tracing continues until the AHB trace enable bits are cleared.

### 3.5.2.3 DSU memory map

DSU memory map can be seen in the table below.

Address	Register
0x90000000	DSU control register
0x90000004	Trace buffer control register
0x90000008	Time tag counter
0x90000010	AHB break address 1
0x90000014	AHB mask 1
0x90000018	AHB break address 2
0x9000001C	AHB mask 2
0x90010000 - 0x90020000	Trace buffer
..0	Trace bits 127 - 96
...4	Trace bits 95 - 64
...8	Trace bits 63 - 32
...C	Trace bits 31 - 0
0x90020000 - 0x90040000	IU/FPU register file
0x90080000 - 0x90100000	IU special purpose registers

Address	Register
0x90080000	Y register
0x90080004	PSR register
0x90080008	WIM register
0x9008000C	TBR register
0x90080010	PC register
0x90080014	NPC register
0x90080018	FSR register
0x9008001C	DSU trap register
0x90080040 - 0x9008007C	ASR16 - ASR31 (when implemented)
0x90100000 - 0x90140000	Instruction cache tags
0x90140000 - 0x90180000	Instruction cache data
0x90180000 - 0x901C0000	Data cache tags
0x901C0000 - 0x90200000	Data cache data

**Table 3-13. DSU address space**

The addresses of the IU/FPU registers depends on how many register windows has been implemented and if and FPU is present. The registers can be accessed at the following addresses (NWINDOVS = number of SPARC register windows):

- %on:  $0x90020000 + (((psr.cwp * 64) + 32 + n) \bmod (NWINDOVS * 64))$
- %ln:  $0x90020000 + (((psr.cwp * 64) + 64 + n) \bmod (NWINDOVS * 64))$
- %in:  $0x90020000 + (((psr.cwp * 64) + 96 + n) \bmod (NWINDOVS * 64))$
- %gn:  $0x90020000 + (NWINDOVS * 64) + 128$  (FPU present)
- %fn:  $0x90020000 + (NWINDOVS * 64)$  (Meiko)

### 3.5.2.4 DSU control register

The DSU is controlled by the DSU control register:

31	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DCNT		RE	DR	LR	SS	PE	EE	EB	DM	DE	BZ	BX	BB	BN	BS	BW	BE	FT	BT	DM	TE

**Figure 3-48. DSU control register**

0:Trace enable (TE). Enables the trace buffer.

1:Delay counter mode (DM). In mixed tracing mode, setting this bit will cause the delay counter to decrement on AHB traces. If reset, the delay counter will decrement on instruction traces.

2:Break on trace (BT) - if set, will generate a DSU break condition on trace freeze.

3:Freeze timers (FT) - if set, the scaler in the LEON timer unit will be stopped during debug mode to preserve the time for the software application.

4:Break on error (BE) - if set, will force the processor to debug mode when the processor would have entered error condition (trap in trap).

5:Break on IU watchpoint - if set, debug mode will be forced on a IU watchpoint (trap 0xb).

6:Break on S/W breakpoint (BS) - if set, debug mode will be forced when an breakpoint instruction (ta 1) is executed.

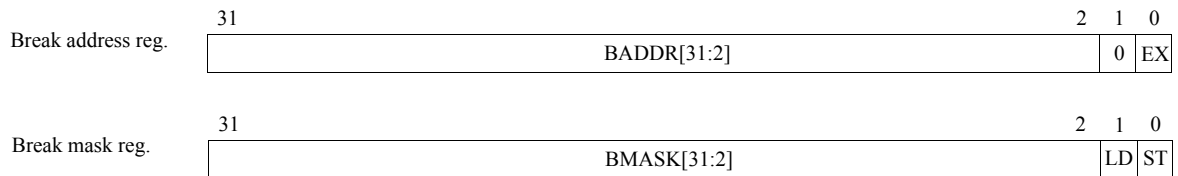
7:Break now (BN) -Force processor into debug mode. If cleared, the processor will resume execution.

8:Break on DSU breakpoint (BD) - if set, will force the processor to debug mode when an DSU breakpoint is hit.

- 9:Break on trap (BX) - if set, will force the processor into debug mode when any trap occurs.
- 10:Break on error traps (BZ) - if set, will force the processor into debug mode on all *except* the following traps: privileged\_instruction, fpu\_disabled, window\_overflow, window\_underflow, asynchronous\_interrupt, ticc\_trap.
- 11:Delay counter enable (DE) - if set, the trace buffer delay counter will decrement for each stored trace. This bit is set automatically when an DSU breakpoint is hit and the delay counter is not equal to zero.
- 12:Debug mode (DM). Indicates when the processor has entered debug mode (read-only).
- 13:EB - value of the external LeonDsuBre signal (read-only)
- 14:EE - value of the external LeonDsuEn signal (read-only)
- 15: Processor error mode (PE) - returns '1' on read when processor is in error mode, else '0'.
- 16:Single step (SS) - if set, the processor will execute one instruction and the return to debug mode.
- 17: Link response (LR) - is set, the DSU communication link will send a response word after AHB transfer.
- 18:Debug mode response (DR) - if set, the DSU communication link will send a response word when the processor enters debug mode.
- 19: Reset error mode (RE) - if set, will clear the error mode in the processor.
- 31:20 Trace buffer delay counter (DCNT). Note that the number of bits actually implemented depends on the size of the trace buffer.

### 3.5.2.5 DSU breakpoint registers

The DSU contains two breakpoint registers for matching either AHB addresses or executed processor instructions. A breakpoint hit is typically used to freeze the trace buffer, but can also put the processor in debug mode. Freezing can be delayed by programming the DCNT field in the DSU control register to a non-zero value. In this case, the DCNT value will be decremented for each additional trace until it reaches zero, after which the trace buffer is frozen. If the BT bit in the DSU control register is set, the DSU will force the processor into debug mode when the trace buffer is frozen. Note that due to pipeline delays, up to 4 additional instruction can be executed before the processor is placed in debug mode. A mask register is associated with each breakpoint, allowing breaking on a block of addresses. Only address bits with the corresponding mask bit set to '1' are compared during breakpoint detection. To break on executed instructions, the EX bit should be set. To break on AHB load or store accesses, the LD and/or ST bits should be set.



**Figure 3-49. DSU breakpoint registers**

- BADDR : breakpoint address (bits 31:2)
- EX : break on instruction
- BMASK : Breakpoint mask
- LD : break on data load address
- ST : beak on data store address

### 3.5.2.6 DSU trap register

The DSU trap register is a read-only register that indicates which SPARC trap type that caused the processor to enter debug mode. When debug mode is force by setting the BN bit in the DSU control register, the trap type will be 0xb (hardware watchpoint trap).



**Figure 3-50. DSU trap register**

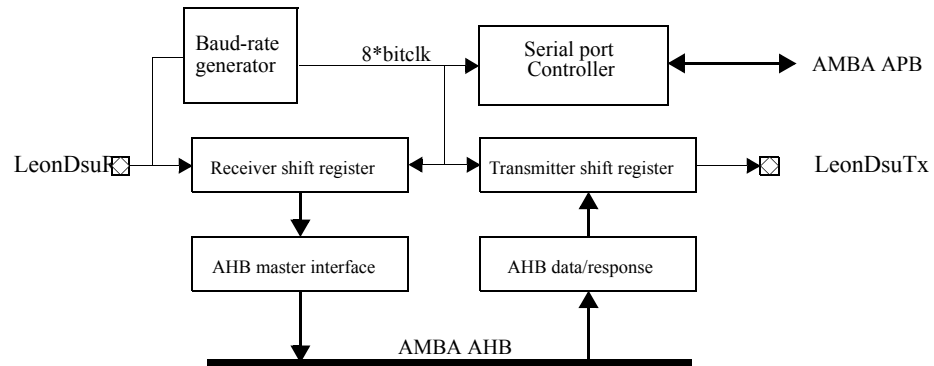


[31:12]: Reserved. No effect when written to. Undefined when read.  
 [11:4] : 8-bit SPARC trap type  
 [12] : Error mode (EM). Set if the trap would have cause the processor to enter error mode.  
 [3:0]: Reserved. No effect when written to. Undefined when read.

### 3.5.3 DSU communication link

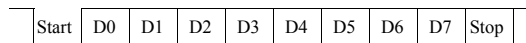
#### 3.5.3.1 Operation

The DSU communication link consists of a UART connected to the AHB bus as a master. A simple communication protocol is supported to transmit access parameters and data. A link command consist of a control byte, followed by a 32-bit address, followed by optional write data. If the LR bit in the DSU control register is set, a response byte will be sent after each AHB transfer. If the LR bit is not set, a write access does not return any response, while a read access only returns the read data.

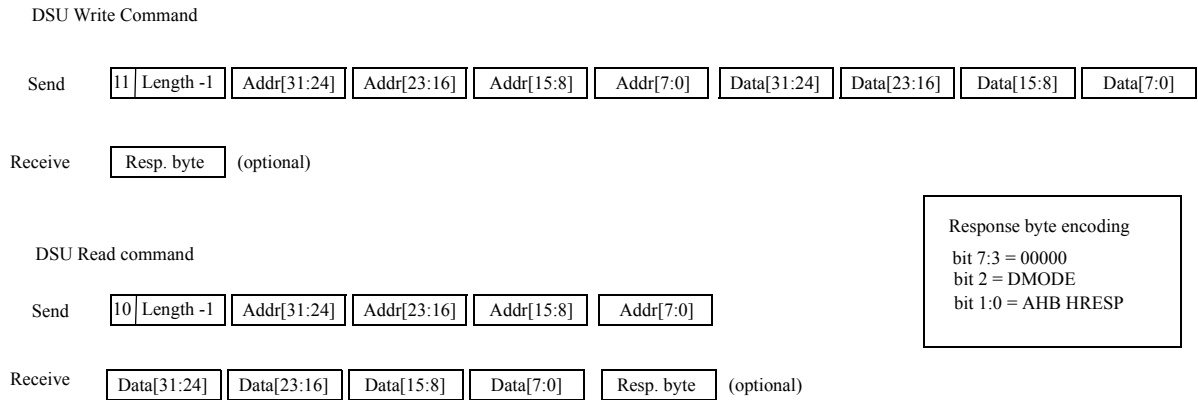


**Figure 3-51. DSU communication link block diagram**

Data is sent on 8-bit basis. Through the communication link, a read or write transfer can be generated to any address on the AHB bus.



**Figure 3-52. DSU UART data frame**

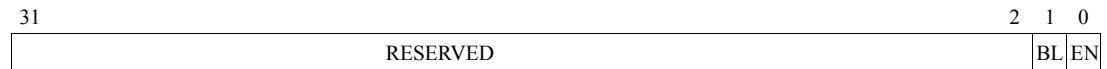


**Figure 3-53. DSU Communication link commands**

A response byte can optionally be sent when the processor goes from execution mode to debug mode. Block transfers can be performed by setting the length field to  $n-1$ , where  $n$  denotes the number of transferred words. For write accesses, the control byte and address is sent once, followed by the number of data words to be written. The address is automatically incremented after each data word. For read accesses, the control byte and address is sent once and the corresponding number of data words is returned.

The UART receiver is implemented with same glitch filtering as the nominal UARTs.

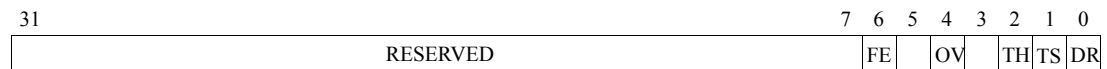
### 3.5.3.2 DSU UART control register



**Figure 3-54. UART control register**

- 31:2: Reserved. No effect when written to. Undefined when read.
- 0: Receiver enable (RE) - if set, enables both the transmitter and receiver.
- 1: Baud rate locked (BL) - is automatically set when the baud rate is locked.

### 3.5.3.3 DSU UART status register



**Figure 3-55. UART status register**

- 0: Data ready (DR) - indicates that new data has been received and not yet read-out by the AHB master interface.
- 1: Transmitter shift register empty (TS) - indicates that the transmitter shift register is empty.
- 2: Transmitter hold register empty (TH) - indicates that the transmitter hold register is empty.
- 3: Reserved. No effect when written to. Undefined when read.
- 4: Overrun (OV) - indicates that one or more character have been lost due to overrun.
- 5: Reserved. No effect when written to. Undefined when read.
- 6: Framing error (FE) - indicates that a framing error was detected.

31:7: Reserved. No effect when written to. Undefined when read.

### 3.5.3.4 Baud rate generation

The UART contains a 18-bit down-counting scaler to generate the desired baud-rate. The scaler is clocked by the system clock and generates a UART tick each time it underflows. The scaler is reloaded with the value of the UART scaler reload register after each underflow. The resulting UART tick frequency should be 8 times the desired baud-rate.

If not programmed by software, the baud rate will be automatically be discovered. This is done by searching for the shortest period between two falling edges of the received data (corresponding to two bit periods). When three identical two-bit periods has been found, the corresponding scaler reload value is latched into the reload register, and the BL bit is set in the UART control register. If the BL bit is reset by software, the baud rate discovery process is restarted. The baud-rate discovery is also restarted when a 'break' or framing error is detected by the receiver, allowing to change to baudrate from the external transmitter. For proper baudrate detection, the value 0x55 should be transmitted to the receiver after reset or after sending break.

The best scaler value for manually programming the baudrate can be calculated as follows:

$$\text{scaler} = (((\text{system\_clk} * 10) / (\text{baudrate} * 8)) - 5) / 10$$

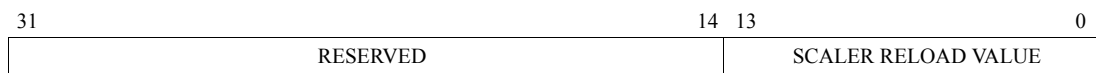


Figure 3-56. DSU UART scaler reload register

[31:14]: Reserved. No effect when written to. Undefined when read.

## 3.5.4 Common operations

### 3.5.4.1 Instruction breakpoints

Instruction breakpoints can be inserted by writing the breakpoint instruction (ta 1) to the desired memory address (software breakpoint) or using any of the four integer unit hardware breakpoints. Since cache snooping is only done on the data cache, the instruction cache must be flushed after the insertion or removal of software breakpoints. To minimize the influence on execution, it is enough to clear the corresponding instruction cache tag valid bit (which is accessible through the DSU).

The two DSU hardware breakpoints should only be used to freeze the trace buffer, and not for software debugging since there is a 4-instruction delay from the breakpoint hit before the processor enters the debug mode.

### 3.5.4.2 Single stepping

By setting the SS bit and clearing the BN bit in the DSU control register, the processor will resume execution for one instruction and then automatically return to debug mode.

### 3.5.4.3 Booting from DSU

By asserting LeonDsuEn and LeonDsuBre at reset time, the processor will directly enter debug mode without executing any instructions. The system can then be initialised from the communication link, and applications can be downloaded and debugged. Additionally, external (flash) prompts for standalone booting can be re-programmed.

## 3.6 Vendor and device id

The AMBA AHB master of the LEON2 caches has vendor id 0x04 (ESA) and device id 0x002.

The AMBA AHB slave of the LEON2 memory controller has vendor id 0x04 (ESA) and device id 0x00F.

The AMBA AHB slave of the LEON2 DSU has vendor id 0x01 (Gaisler Research) and device id 0x002.

The AMBA AHB master of the LEON2 DSU AHB UART has vendor id 0x04 (ESA) and device id 0x013.

The AMBA APB slave of the LEON2 peripherals has vendor id 0x04 (ESA) and device id 0x003.

## 4. ON-CHIP MEMORY

### 4.1 Overview

The On-Chip Memory is implemented with the FTAHBRAM core. The FTAHBRAM is a version of the normal AHBRAM core with added Error Detection And Correction (EDAC). One error is corrected and two errors are detected, which is done by using a (32, 7) BCH code. Configuration is possible through an APB interface. Some of the features available are: single error counter, diagnostic reads and writes and autoscrubbing (automatic correction of single errors during reads). Figure below shows a block diagram of the internals of the RAM.

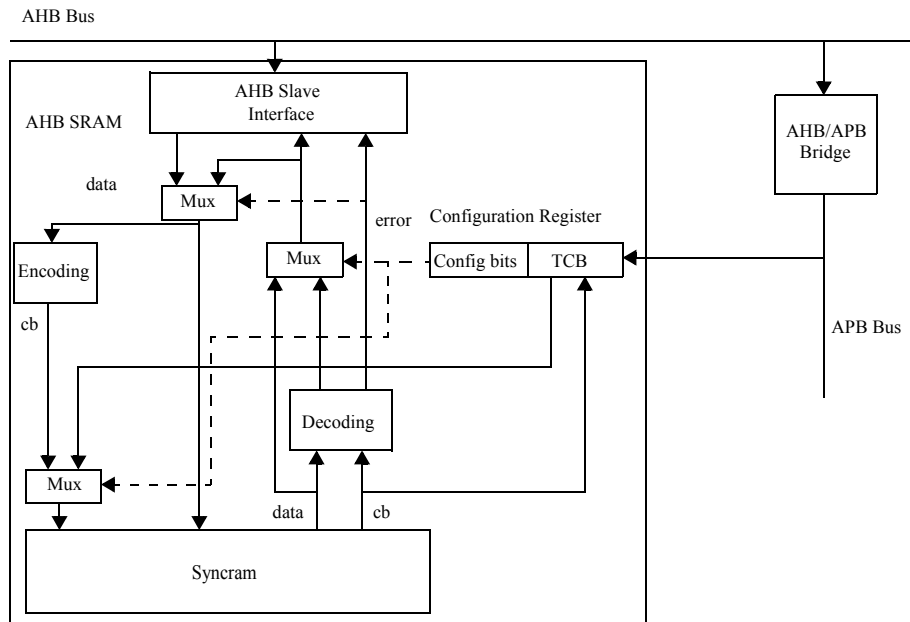


Figure 4-1. The FT AHB RAM block diagram

### 4.2 Operation

The on-chip fault tolerant RAM is accessed through an AHB slave interface.

Run-time configuration is done by writing to a configuration register accessed through an APB interface. The fields of the configuration register are shown in detail later in this section.

The following can be configured during run-time: EDAC can be enabled and disabled. Read and write diagnostics can be controlled through separate bits. The single error counter can be reset.

If EDAC is disabled (EN bit in configuration register set to 0) write data is passed directly to the RAM area and read data will appear on the AHB bus immediately after it arrives from memory. If EDAC is enabled write data is passed to an encoder which outputs a 7-bit checksum. The checksum is stored together with the data in memory and the whole operation is performed without any added waitstates. This applies to word stores (32-bit). To ensure correct checksum, RAM locations containing data prior to enabling the EDAC, must first be refreshed by reading and writing a complete word before enabling the EDAC. If a byte or halfword store is performed, the whole word to which the byte or halfword belongs must first be read from memory (read - modify - write). A new checksum is calculated when the new data is placed in the word and both data and checksum are stored in memory. This is done with 1 - 2 additional waitstates compared to the non EDAC case. Note that autoscrubbing is only performed on read/write access to a memory location, thus it is recommended to read in-frequently accessed memory periodically in order to reduce accumulation of SEU bit errors.

Reads with EDAC disabled are performed with 0 or 1 waitstates while there could also be 2 waitstates when EDAC is enabled. There is no difference between word and subword reads. Next table shows a summary of the number of waitstates for the different operations with and without EDAC.

Operation	Waitstates with EDAC Disabled	Waitstates with EDAC Enabled
<i>Read</i>	0 - 1	0 - 2
<i>Word write</i>	0	0
<i>Subword write</i>	0	1 - 2

**Table 4-1. Summary of the number of waitstates for the different operations for the AHB RAM.**

When EDAC is used, the data is decoded the first cycle after it arrives from the memory and appears on the bus the next cycle if no uncorrectable error is detected. The decoding is done by comparing the stored checksum with a new one which is calculated from the stored data. This decoding is also done during the read phase for a subword write. A so-called syndrome is generated from the comparison between the checksum and it determines the number of errors that occurred. One error is automatically corrected and this situation is not visible on the bus. Two or more detected errors cannot be corrected so the operation is aborted and the required two cycle error response is given on the AHB bus (see the AMBA manual for more details). If no errors are detected data is passed through the decoder unaltered.

As mentioned earlier the AHB RAM provides read and write diagnostics when EDAC is enabled. When write diagnostics are enabled, the calculated checksum is not stored in memory during the write phase. Instead, the TCB field from the configuration register is used. In the same manner, if read diagnostics are enabled, the stored checksum from memory is stored in the TCB field during a read (and also during a subword write). This way, the EDAC functionality can be tested during run-time. Note that checkbits are stored in TCB during reads and subword writes even if a multiple error is detected.

An additional feature is the single error counter. A single error counter (SEC) field is present in the configuration register. It is incremented each time a single databit error is encountered (reads or subword writes). The number of bits of this counter is 8. It is accessed via the configuration register. Each bit can be reset to zero by writing a one to it. The counter saturates at the value 255.

Autoscrubbing is an error handling feature and cannot be controlled through the configuration register. Every single error encountered during a read results in the word being written back with the error corrected and new checkbits generated. It is not visible externally except for that it can generate an extra waitstate. This happens if the read is followed by an odd numbered read in a burst sequence of reads or if a subword write follows. These situations are very rare during normal operation so the total timing impact is negligible.

### 4.3 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x50.

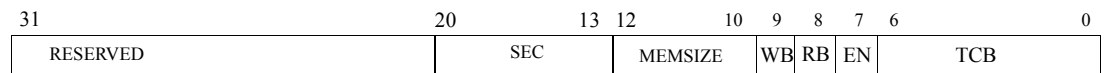
### 4.4 Registers

Next table shows the FTAHB RAM registers.

Register	Address
Configuration Register	0x80010000

**Table 4-2. FT AHB RAM unit registers**

The following figure shows the register bit fields. All fields except TCB are initialised at reset. The EDAC is initially disabled (EN = 0), which also applies to diagnostics (RB and WB are zero). Additionally, if the single error counter is enabled, the error counter (SEC) is set to zero and interrupts are disabled (IE = 0).



**Figure 4-2. Configuration register bit fields**

[31: 21] - Reserved. Write access has no effect. Read data is undefined.

[20: 13] - Single error counter. Incremented each time a single error is corrected (includes errors on checkbits). Each bit can be set to zero by writing a one to it.

[12 : 10] - Log2 of the current memory size

[9] - Write Bypass (WB) : When set, the TCB field is stored as check bits when a write is performed to the memory.

[8] - Read Bypass (RB) : When set during a read or subword write, the check bits loaded from memory are stored in the TCB field.

[7] - EDAC Enable : When set, the EDAC is used otherwise it is bypassed during read and write operations.

[6:0] - Test Check Bits (TCB) : Used as checkbits when the WB bit is set during writes and loaded with the check bits during a read operation when the RB bit is set.

## 5. FIFO INTERFACE

### 5.1 Overview

The FIFO interface is assumed to operate in an AMBA bus system where both the AMBA AHB bus and the APB bus are present. The AMBA APB bus is used for configuration, control and status handling. The AMBA AHB bus is used for retrieving and storing FIFO data in memory external to the FIFO interface. This memory can be located on-chip or external to the chip.

The FIFO interface supports transmission and reception of blocks of data by use of circular buffers located in memory external to the core. Separate transmit and receive buffers are assumed. Reception and transmission of data can be ongoing simultaneously.

After a data transfer has been set up via the AMBA APB interface, the DMA controller initiates a burst of read accesses on the AMBA AHB bus to fetch data from memory that are performed by the AHB master. The data are then written to the external FIFO. When a programmable amount of data has been transmitted, the DMA controller issues an interrupt.

After reception has been set up via the AMBA APB interface, data are read from the external FIFO. To store data to memory, the DMA controller initiates a burst of write accesses on the AMBA AHB bus that are performed by the AHB master. When a programmable amount of data has been received, the DMA controller issues an interrupt.

The block diagram shows a possible usage of the FIFO interface.

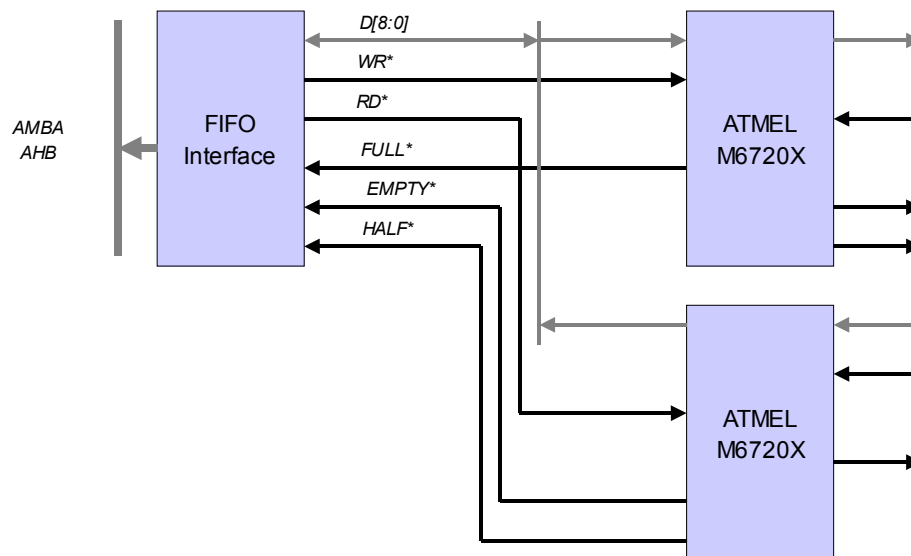


Figure 5-1. Block diagram of the GRFIFO environment.

#### 5.1.1 Function

The core implements the following functions:

- data transmission to external FIFO
- circular transmit buffer
- direct memory access for transmitter
- data reception from external FIFO
- circular receive buffer
- direct memory access for receiver
- automatic 8- and 16-bit data width conversion
- general purpose input output

### 5.1.2 Transmission

Data to be transferred via the FIFO interface are fetched via the AMBA AHB master interface from on-chip or off-chip memory. This is performed by means of direct memory access (DMA), implementing a circular transmit buffer in the memory. The transmit channel is programmable via the AMBA APB slave interface, which is also used for the monitoring of the FIFO and DMA status.

The transmit channel is programmed in terms of a base address and size of the circular transmit buffer. The outgoing data are stored in the circular transmit buffer by the system. A write address pointer register is then set by the system to indicate the last byte written to the circular transmit buffer. An interrupt address pointer register is used by the system to specify a location in the circular transmit buffer from which a data read should cause an interrupt to be generated.

The FIFO interface automatically indicates with a read address pointer register the location of the last fetched byte from the circular transmit buffer. Read accesses are performed as incremental bursts, except when close to the location specified by the interrupt pointer register at which point the last bytes might be fetched by means of single accesses.

Data transferred via the FIFO interface can be either 8- or 16-bit wide. The handling of the transmit channel is however the same. All transfers performed by the AMBA AHB master are 32-bit word based. No byte or half-word transfers are performed.

To handle the 8- and 16-bit FIFO data width, a 32-bit read access might carry less than four valid bytes. In such a case, the remaining bytes are ignored. When additional data are available in the circular transmit buffer, the previously fetched bytes will be re-read together with the newly written bytes to form the 32-bit data. Only the new bytes will be transmitted to the FIFO, not to transmit the same byte more than once. The aforementioned write address pointer indicates what bytes are valid.

An interrupt is generated when the circular transmit buffer is empty. The status of the external FIFO is observed via the AMBA APB slave interface, indicating Full Flag and Half-Full Flag.

### 5.1.3 Reception

Data received via the FIFO interface are stored via the AMBA AHB master interface to on-chip or off-chip memory. This is performed by means of direct memory access (DMA), implementing a circular receive buffer in the memory. The receive channel is programmable via the AMBA APB slave interface, which is also used for the monitoring of the FIFO and DMA status.

The receive channel is programmed in terms of a base address and size of the circular receive buffer. The incoming data are stored in the circular receive buffer. The interface automatically indicates with a write address pointer register the location of the last stored byte. A read address pointer register is used by the system to indicate the last byte read from the circular receive buffer. An interrupt address pointer register is used by the system to specify a location in the circular receive buffer to which a data write should cause an interrupt to be generated.

Write accesses are performed as incremental bursts, except when close to the location specified by the interrupt pointer register at which point the last bytes might be stored by means of single accesses.

Data transferred via the FIFO interface can be either 8- or 16-bit wide. The handling of the receive channel is however the same. All transfers performed by the AMBA AHB master are 32-bit word based. No byte or half-word transfers are performed.

To handle the 8- and 16-bit FIFO data width, a 32-bit write access might carry less than four valid bytes. In such a case, the remaining bytes will all be zero. When additional data are received from the FIFO interface, the previously stored bytes will be re-written together with the newly received bytes to form the 32-bit data. In this way, the previously written bytes are never overwritten. The aforementioned write address pointer indicates what bytes are valid.

An interrupt is generated when the circular receive buffer is full. If more FIFO data are available, they will not be moved to the circular receive buffer. The status of the external FIFO is observed via the AMBA APB slave interface, indicating Empty Flag and Half-Full Flag.

### 5.1.4 General purpose input output

Data input and output signals unused by the FIFO interface can be used as general purpose input output, providing 0, 8 or 16 individually programmable channels.

### 5.1.5 Interfaces

The core provides the following external and internal interfaces:

- FIFO interface
- AMBA AHB master interface, with sideband signals as per [GLRIB] including:
  - cachability information
  - interrupt bus
  - configuration information



- diagnostic information
- AMBA APB slave interface, with sideband signals as per [GLRIB] including:
  - interrupt bus
  - configuration information
  - diagnostic information

The interface is intended to be used with the following FIFO devices from ATMEL:

Name:	Type:	
M67204H	4K x 9 FIFO	ESA/SCC 9301/049, SMD/5962-89568
M67206H	16K x 9 FIFO	ESA/SCC 9301/048, SMD/5962-93177
M672061H	16K x 9 FIFO	ESA/SCC 9301/048, SMD/5962-93177

## 5.2 Interface

The external interface supports one or more FIFO devices for data output (transmission) and/or one or more FIFO devices for data input (reception). The external interface supports FIFO devices with 8- and 16-bit data width. Note that one device is used when 8-bit and two devices are used when 16-bit data width is needed. The data width is programmable. Note that this is performed commonly for both directions.

The external interface supports one parity bit over every 8 data bits. Note that there can be up to two parity bits in either direction. The parity is programmable in terms of odd or even parity. Note that odd parity is defined as an odd number of logical ones in the data bits and parity bit. Note that even parity is defined as an even number of logical ones in the data bits and parity bit. Parity is generated for write accesses to the external FIFO devices. Parity is checked for read accesses from the external FIFO devices and a parity failure results in an internal interrupt.

The external interface provides a Write Enable output signal. The external interface provides a Read Enable output signal. The timing of the access towards the FIFO devices is programmable in terms of wait states based on system clock periods.

The external interface provides an Empty Flag input signal, which is used for flow-control during the reading of data from the external FIFO, not reading any data while the external FIFO is empty. Note that the Empty Flag is sampled at the end of the read access to determine if the FIFO is empty. To determine when the FIFO is not empty, the Empty Flag is re-synchronized with Clk.

The external interface provides a Full Flag input signal, which is used for flow-control during the writing of data to the external FIFO, not writing any data while the external FIFO is full. Note that the Full Flag is sampled at the end of the write access to determine if the FIFO is full. To determine when the FIFO is not full, the Full Flag is re-synchronized with Clk.

The external interface provides a Half-Full Flag input signal, which is used as status information only.

The data input and output signals are possible to use as general purpose input output channels. This need is only realized when the data signals are not used by the FIFO interface. Each general purpose input output channel is individually programmed as input or output. The default reset configuration for each general purpose input output channel is as input. The default reset value each general purpose input output channel is logical zero. Note that protection toward spurious pulse commands during power up shall be mitigated as far as possible by means of I/O cell selection from the target technology.

5.3 Waveforms

The following figures show read and write accesses to the FIFO with 0 and 4 wait states, respectively.

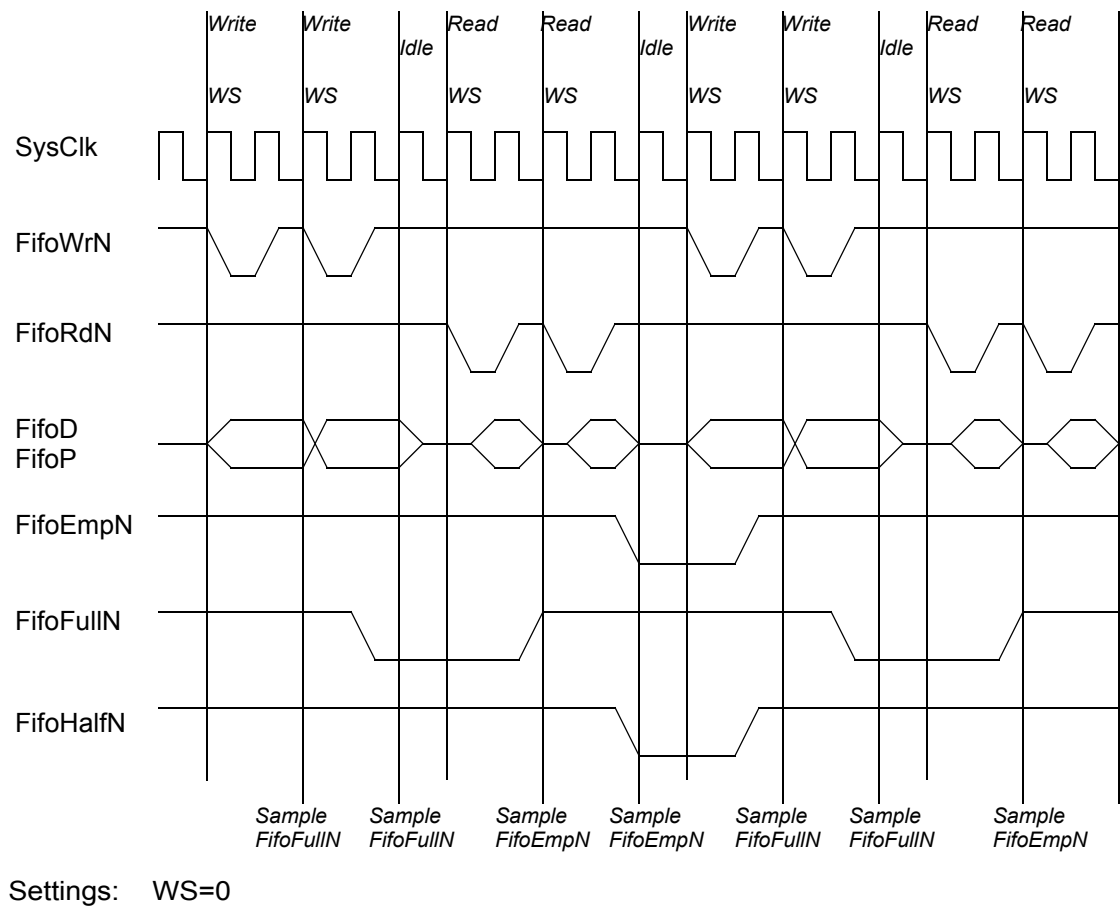
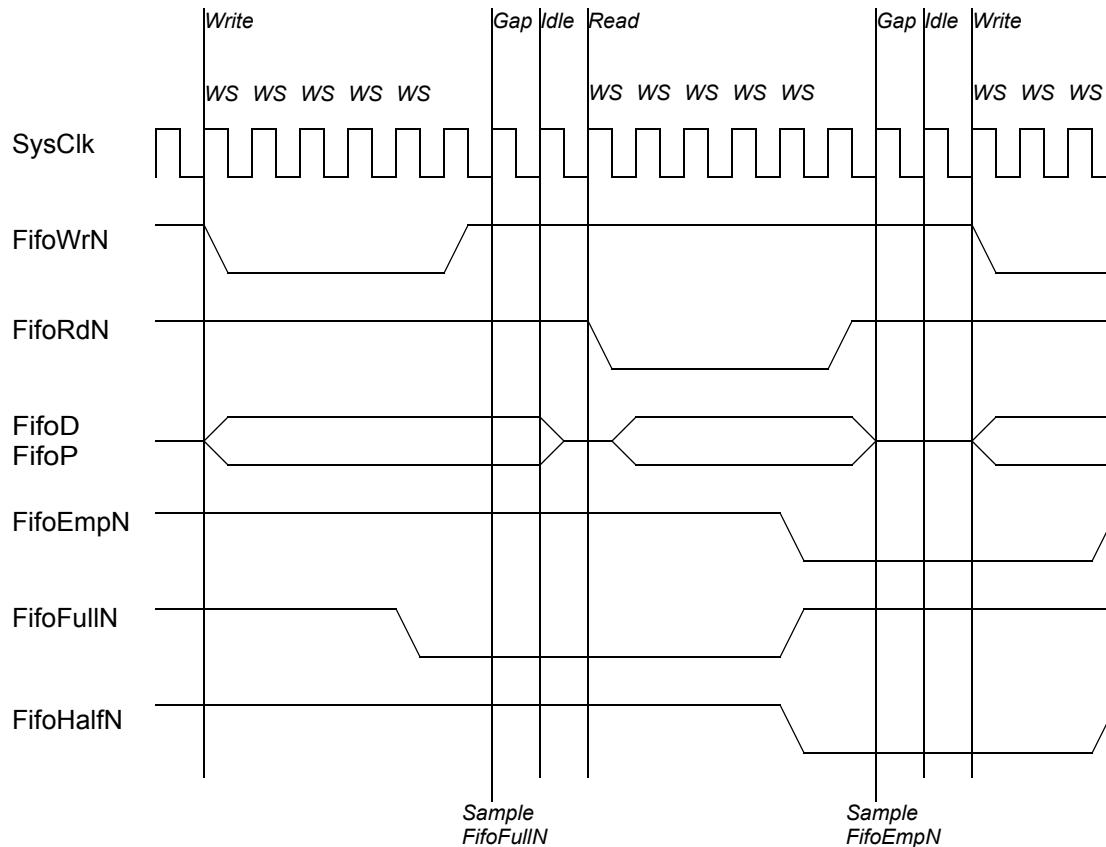


Figure 5-2. FIFO read and write access waveform, 0 wait states (WS)



Settings: WS=4 (with additional gap between accesses)

**Figure 5-3. FIFO read and write access waveform, 4 wait states (WS)**

## 5.4 Transmission

The transmit channel is defined by the following parameters:

- base address
- buffer size
- write pointer
- read pointer

The transmit channel can be enabled or disabled.

### 5.4.1 Circular buffer

The transmit channel operates on a circular buffer located in memory external to the FIFO controller. The circular buffer can also be used as a straight buffer. The buffer memory is accessed via the AMBA AHB master interface.

The size of the buffer is defined by the FifoTxSIZE.SIZE field, specifying the number of 64 byte blocks that fit in the buffer.

E.g. FifoTxSIZE.SIZE = 1 means 64 bytes fit in the buffer.

Note however that it is not possible to fill the buffer completely, leaving at least one word in the buffer empty. This is to simplify wrap-around condition checking.

E.g. FifoTxSIZE.SIZE = 1 means that 60 bytes fit in the buffer at any given time.

### 5.4.2 Write and read pointers

The write pointer (FifoTxWR.WRITE) indicates the position+1 of the last byte written to the buffer. The write pointer operates on number of bytes, not on absolute or relative addresses.

The read pointer (FifoTxRD.READ) indicates the position+1 of the last byte read from the buffer. The read pointer operates on number of bytes, not on absolute or relative addresses.

The difference between the write and the read pointers is the number of bytes available in the buffer for transmission. The difference is calculated using the buffer size, specified by the FifoTxSIZE.SIZE field, taking wrap around effects of the circular buffer into account.

Examples:

- There are 2 bytes available for transmit when FifoTxSIZE.SIZE=1, FifoTxWR.WRITE=2 and FifoTxRD.READ=0.
- There are 2 bytes available for transmit when FifoTxSIZE.SIZE=1, FifoTxWR.WRITE =0 and FifoTxRD.READ =62.
- There are 2 bytes available for transmit when FifoTxSIZE.SIZE=1, FifoTxWR.WRITE =1 and FifoTxRD.READ =63.
- There are 2 bytes available for transmit when FifoTxSIZE.SIZE=1, FifoTxWR.WRITE =5 and FifoTxRD.READ =3.

When a byte has been successfully written to the FIFO, the read pointer (FifoTxRD.READ) is automatically incremented, taking wrap around effects of the circular buffer into account. Whenever the write pointer FifoTxWR.WRITE and read pointer FifoTxRD.READ are equal, there are no bytes available for transmission.

### 5.4.3 Location

The location of the circular buffer is defined by a base address (FifoTxADDR.ADDR), which is an absolute address. The location of a circular buffer is aligned on a 1kbyte address boundary.

### 5.4.4 Transmission procedure

When the channel is enabled (FifoTxCTRL.ENABLE=1), as soon as there is a difference between the write and read pointer, a transmission will be started. Note that the channel should not be enabled if a potential difference between the write and read pointers could be created, to avoid the data transmission to start prematurely.

A data transmission will begin with a fetch of the data from the circular buffer to a local buffer in the FIFO controller. After a successful fetch, a write access will be performed to the FIFO.

The read pointer (FifoTxRD.READ) is automatically incremented after a successful transmission, taking wrap around effects of the circular buffer into account. If there is at least one byte available in the circular buffer, a new fetch will be performed.

If the write and read pointers are equal, no more prefetches and fetches will be performed, and transmission will stop.

Interrupts are provided to aid the user during transmission, as described in detail later in this section. The main interrupts are the TxError, TxEmpty and TxIrq which are issued on the unsuccessful transmission of a byte due to an error condition on the AMBA bus, when all bytes have been transmitted successfully and when a predefined number of bytes have been transmitted successfully.

Note that 32-bit wide read accesses past the address of the last byte or halfword available for transmission can be performed as part of a burst operation, although no read accesses are made beyond the circular buffer size.

All accesses to the AMBA AHB bus are performed as two consecutive 32-bit accesses in a burst, or as a single 32-bit access in case of an AMBA AHB bus error.

### 5.4.5 Straight buffer

It is possible to use the circular buffer as a straight buffer, with a higher granularity than the 1kbyte address boundary limited by the base address (FifoTxADDR.ADDR) field.

While the channel is disabled, the read pointer (FifoTxRD.READ) can be changed to an arbitrary value pointing to the first byte to be transmitted, and the write pointer (FifoTxWR.WRITE) can be changed to an arbitrary value.

When the channel is enabled, the transmission will start from the read pointer and continue to the write pointer.

### 5.4.6 AMBA AHB error

An AHB error response occurring on the AMBA AHB bus while data is being fetched will result in a TxError interrupt.

If the FifoCONF.ABORT bit is set to 0b, the channel causing the AHB error will re-try to read the data being fetched from memory till successful.

If the FifoCONF.ABORT bit is set to 1b, the channel causing the AHB error will be disabled (FifoTxCTRL.ENABLE is cleared automatically to 0 b). The read pointer can be used to determine which data caused the AHB error. The interface will not start any new write accesses to the FIFO. Any ongoing FIFO access will be completed and the FifoTxSTAT.TxOnGoing bit will be cleared. When the channel is re-enabled, the fetch and transmission of data will resume at the position where it was disabled, without losing any data.

#### 5.4.7 Enable and disable

When an enabled transmit channel is disabled (FifoTxCTRL.ENABLE=0b), the interface will not start any new read accesses to the circular buffer by means of DMA over the AMBA AHB bus. No new write accesses to the FIFO will be started. Any ongoing FIFO access will be completed. If the data is written successfully, the read pointer (FifoTxRD.READ) is automatically incremented and the FifoTxSTAT.TxOnGoing bit will be cleared. Any associated interrupts will be generated.

Any other fetched or pre-fetched data from the circular buffer which is temporarily stored in the the local buffer will be discarded, and will be fetched again when the transmit channel is re-enabled.

The progress of the any ongoing access can be observed via the FifoTxSTAT.TxOnGoing bit. The FifoTxSTAT.TxOnGoing must be 0b before the channel can be re-configured safely (i.e. changing address, size or read/write pointers). It is also possible to wait for the TxEmpty interrupt described hereafter.

The channel can be re-enabled again without the need to re-configure the address, size and pointers. No data transmission is started while the channel is not enabled.

#### 5.4.8 Interrupts

During transmission several interrupts can be generated:

- TxEmpty:Successful transmission of all data in buffer
- TxIrq:Successful transmission of a predefined number of data
- TxError:AHB access error during transmission

The TxEmpty and TxIrq interrupts are only generated as the result of a successful data transmission, after the FifoTxRD.READ pointer has been incremented.

### 5.5 Reception

The receive channel is defined by the following parameters:

- base address
- buffer size
- write pointer
- read pointer

The receive channel can be enabled or disabled.

#### 5.5.1 Circular buffer

The receive channel operates on a circular buffer located in memory external to the FIFO controller. The circular buffer can also be used as a straight buffer. The buffer memory is accessed via the AMBA AHB master interface.

The size of the buffer is defined by the FifoRxSIZE.SIZE field, specifying the number 64 byte blocks that fit in the buffer.

E.g. FifoRxSIZE.SIZE=1 means 64 bytes fit in the buffer.

Note however that it is not possible for the hardware to fill the buffer completely, leaving at least two words in the buffer empty. This is to simplify wrap-around condition checking.

E.g. FifoRxSIZE.SIZE=1 means that 56 bytes fit in the buffer at any given time.

#### 5.5.2 Write and read pointers

The write pointer (FifoRxWR.WRITE) indicates the position+1 of the last byte written to the buffer. The write pointer operates on number of bytes, not on absolute or relative addresses.

The read pointer (FifoRxRD.READ) indicates the position+1 of the last byte read from the buffer. The read pointer operates on number of bytes, not on absolute or relative addresses.

The difference between the write and the read pointers is the number of bytes available in the buffer for reception. The difference is calculated using the buffer size, specified by the `FifoRxSIZE.SIZE` field, taking wrap around effects of the circular buffer into account.

Examples:

- There are 2 bytes available for read-out when `FifoRxSIZE.SIZE=1`, `FifoRxWR.WRITE =2` and `FifoRxRD.READ=0`.
- There are 2 bytes available for read-out when `FifoRxSIZE.SIZE=1`, `FifoRxWR.WRITE =0` and `FifoRxRD.READ=62`.
- There are 2 bytes available for read-out when `FifoRxSIZE.SIZE=1`, `FifoRxWR.WRITE =1` and `FifoRxRD.READ=63`.
- There are 2 bytes available for read-out when `FifoRxSIZE.SIZE=1`, `FifoRxWR.WRITE =5` and `FifoRxRD.READ=3`.

When a byte has been successfully received and stored, the write pointer (`FifoRxWR.WRITE`) is automatically incremented, taking wrap around effects of the circular buffer into account.

### 5.5.3 Location

The location of the circular buffer is defined by a base address (`FifoRxADDR.ADDR`), which is an absolute address. The location of a circular buffer is aligned on a 1kbyte address boundary.

### 5.5.4 Reception procedure

When the channel is enabled (`FifoRxCTRL.ENABLE=1`), and there is space available for data in the circular buffer (as defined by the write and read pointer), a read access will be started towards the FIFO, and then an AMBA AHB store access will be started. The received data will be temporarily stored in a local store-buffer in the FIFO controller. Note that the channel should not be enabled until the write and read pointers are configured, to avoid the data reception to start prematurely.

After a datum has been successfully stored the FIFO controller is ready to receive new data. The write pointer (`FifoRxWR.WRITE`) is automatically incremented, taking wrap around effects of the circular buffer into account.

Interrupts are provided to aid the user during reception, as described in detail later in this section. The main interrupts are the `RxError`, `RxParity`, `RxFull` and `RxIrq` which are issued on the unsuccessful reception of data due to an AMBA AHB error or parity error, when the buffer has been successfully filled and when a predefined number of data have been received successfully.

All accesses to the AMBA AHB bus are performed as two consecutive 32-bit accesses in a burst, or as a single 32-bit access in case of an AMBA AHB bus error.

### 5.5.5 Straight buffer

It is possible to use the circular buffer as a straight buffer, with a higher granularity than the 1kbyte address boundary limited by the base address (`FifoRxADDR.ADDR`) field.

While the channel is disabled, the write pointer (`FifoRxWR.WRITE`) can be changed to an arbitrary value pointing to the first data to be received, and the read pointer (`FifoRxRD.READ`) can be changed to an arbitrary value.

When the channel is enabled, the reception will start from the write pointer and continue to the read pointer.

### 5.5.6 AMBA AHB error

An AHB error response occurring on the AMBA AHB bus while data is being stored will result in an `RxError` interrupt.

If the `FifoCONF.ABORT` bit is set to 0b, the channel causing the AHB error will retry to store the received data till successful.

If the `FifoCONF.ABORT` bit is set to 1b, the channel causing the AHB error will be disabled (`FifoRxCTRL.ENABLE` is cleared automatically to 0b). The write pointer can be used to determine which address caused the AHB error. The interface will not start any new read accesses to the FIFO. Any ongoing FIFO access will be completed and the data will be stored in the local receive buffer. The `FifoRxSTAT.ONGOING` bit will be cleared. When the receive channel is re-enabled, the reception and storage of data will resume at the position where it was disabled, without losing any data.

### 5.5.7 Enable and disable

When an enabled receive channel is disabled (`FifoRxCTRL.ENABLE=0b`), any ongoing data storage on the AHB bus will not be aborted, and no new storage will be started. If the data is stored successfully, the write pointer (`FifoRxWR.WRITE`) is automatically incremented. Any associated interrupts will be generated. The interface will not start any new read accesses to the FIFO. Any ongoing FIFO access will be completed.

The channel can be re-enabled again without the need to re-configure the address, size and pointers. No data reception is performed while the channel is not enabled.

The progress of the any ongoing access can be observed via the FifoRxSTAT.ONGOING bit. Note that the there might be data left in the local store-buffer in the FIFO controller. This can be observed via the FifoRxSTAT.RxByteCntr field. The data will not be lost if the channel is not reconfigured before re-enabled.

To empty this data from the local store-buffer to the external memory, the channel needs to be renabled. By setting the FifoRxIRQ.IRQ field to match the value of the FifoRxWR.WRITE field plus the value of the FifoRxSTAT.RxByteCntr field, an emptying to the external memory is forced of any data temporarily stored in the local store-buffer. Note however that additional data could be received in the local store-buffer when the channel is re-enabled.

The FifoRxSTAT.ONGOING must be 0b before the channel can be re-configured safely (i.e. changing address, size or read/write pointers).

### 5.5.8 Interrupts

During reception several interrupts can be generated:

- RxFull:Successful reception of all data possible to store in buffer
- RxIrq:Successful reception of a predefined number of data
- RxError:AHB access error during reception
- RxParity:Parity error during reception

The RxFull and RxIrq interrupts are only generated as the result of a successful data reception, after the FifoRxWR.WRITE pointer has been incremented.

## 5.6 Operation

### 5.6.1 Global reset and enable

When the FifoCTRL.RESET bit is set to 1b, a reset of the core is performed. The reset clears all the register fields to their default values. Any ongoing data transfers will be aborted.

### 5.6.2 Interrupt

Seven interrupts are implemented by the FIFO interface:

Name:	Description:
TxIrq	Successful transmission of block of data
TxEmpty	Circular transmission buffer empty
TxError	AMBA AHB access error during transmission
RxIrq	Successful reception of block of data
RxFull	Circular reception buffer full
RxError	AMBA AHB access error during reception
RxParity	Parity error during reception

### 5.6.3 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x35.

## 5.7 Registers

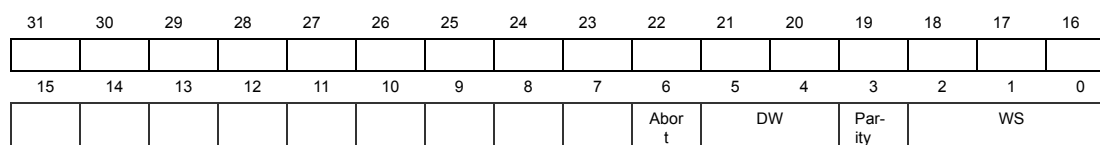
The GRFIFO is programmed through registers mapped into APB address space. Any blank register is considered as reserved and has no effect when written to, and returns undefined data when read.

Register	Address
Configuration Register	0x80050000
Control Register	0x80050008
Transmit Channel Control Register	0x80050020

Register	Address
Transmit Channel Status Register	0x80050024
Transmit Channel Address Register	0x80050028
Transmit Channel Size Register	0x8005002C
Transmit Channel Write Register	0x80050030
Transmit Channel Read Register	0x80050034
Transmit Channel Interrupt Register	0x80050038
Receive Channel Control Register	0x80050040
Receive Channel Status Register	0x80050044
Receive Channel Address Register	0x80050048
Receive Channel Size Register	0x8005004C
Receive Channel Write Register	0x80050050
Receive Channel Read Register	0x80050054
Receive Channel Interrupt Register	0x80050058
Data Input Register	0x80050060
Data Output Register	0x80050064
Data Direction Register	0x80050068

**Table 5-1. GRFIFO registers**

### 5.7.1 Configuration Register[FifoCONF]R/W



**Figure 5-4. Configuration Register**

- 31:7    RESERVED        No affect when written to. Undefined when read.
- 6:     ABORT            Abort transfer on AHB ERROR
- 5-4:   DW               Data width:
  - 00b = none
  - 01b = 8 bitFIFO.Dout[7:0], FIFOI.Din[7:0]
  - 10b = 16 bitFIFO.Dout[15:0],FIFOI.Din[15:0]
  - 11b = spare/none
- 3:     PARITY            Parity type:
  - 0b = even
  - 1b = odd
- 2-0:   WS                Number of wait states, 0 to 7

All bits are cleared to 0 at reset.



Note that the transmit or receive channel active during the AMBA AHB error is disabled if the ABORT bit is set to 1b. Note that all accesses on the affected channel will be disabled after an AMBA AHB error occurs while the ABORT bit is set to 1b. The accesses will be disabled until the affected channel is re-enabled setting the FifoTxCTRL.ENABLE or FifoRxCTRL.ENABLE bit, respectively.

Note that a wait states corresponds to an additional clock cycle added to the period when the read or write strobe is asserted. The default asserted width is one clock period for the read or write strobe when WS=0. Note that an idle gap of one clock cycle is always inserted between read and write accesses, with neither the read nor the write strobe being asserted.

Note that an additional gap of one clock cycle with the read or write strobe de-asserted is inserted between two accesses when WS is equal to or larger than 100b.

## 5.7.2 Control Register[FifoCTRL]R/W

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

**Figure 5-5. Control Register**

- 31:2: RESERVED No affect when written to. Undefined when read.
- 1: RESET Reset complete FIFO interface, all registers
- 0: RESERVED No affect when written to. Undefined when read.

All bits are cleared to 0 at reset.

Note that RESET is read back as 0b.

## 5.7.3 Transmit Channel Control Register[FifoTxCTRL]R/W

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

**Figure 5-6. Transmit Channel Control Register**

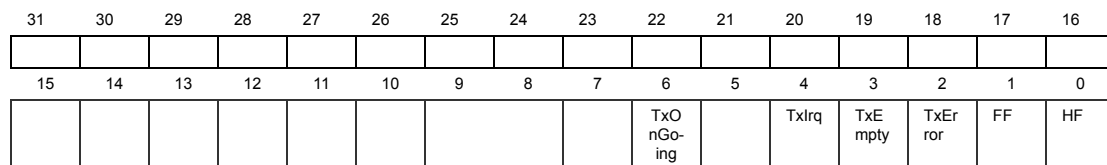
- 31:1: RESERVED No affect when written to. Undefined when read.
- 0: ENABLE Enable channel

All bits are cleared to 0 at reset.

Note that in the case of an AHB bus error during an access while fetching transmit data, and the FifoConf.ABORT bit is 1b, then the ENABLE bit will be reset automatically.

At the time the ENABLE is cleared to 0b, any ongoing data writes to the FIFO are not aborted.

### 5.7.4 Transmit Channel Status Register[FifoTxSTAT]R



**Figure 5-7. Transmit Channel Status Register**

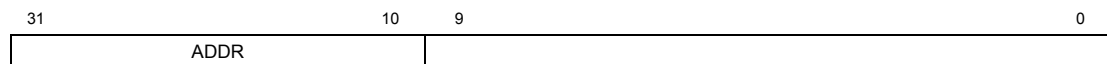
31:7:	RESERVED	No affect when written to. Undefined when read.
6:	TxOnGoing	Access ongoing
5:	RESERVED	No affect when written to. Undefined when read.
4:	TxIrq	Successful transmission of block of data
3:	TxEmply	Transmission buffer has been emptied
2:	TxEr ror	AMB AHB access error during transmission
1:	FF	FIFO Full Flag
0:	HF	FIFO Half-Full Flag

All bits are cleared to 0 at reset.

The following sticky status bits are cleared when the register has been read:

- TxIrq, TxEmply and TxError.

### 5.7.5 Transmit Channel Address Register[FifoTxADDR]R/W

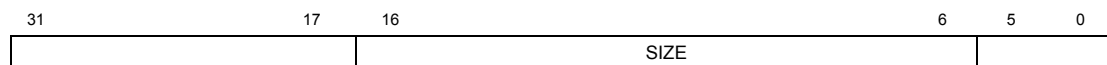


**Figure 5-8. Transmit Channel Address Register**

31-10:	ADDR	Base address for circular buffer
9:0	RESERVED	No affect when written to. Undefined when read.

All bits are cleared to 0 at reset.

### 5.7.6 Transmit Channel Size Register[FifoTxSIZE]R/W



**Figure 5-9. Transmit Channel Size Register**

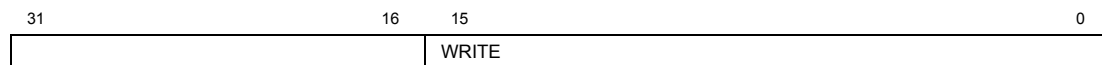
31-17:	RESERVED	No affect when written to. Undefined when read.
16-6:	SIZE	Size of circular buffer, in number of 64 byte blocks
5-0:	RESERVED	No affect when written to. Undefined when read.

All bits are cleared to 0 at reset.

Valid SIZE values are 0, and between 1 and 1024. Note that the resulting behavior of invalid SIZE values is undefined.

Note that only  $\text{SIZE} \times 64 - 4$  bytes can be stored simultaneously in the buffer. This is to simplify wrap-around condition checking.

### 5.7.7 Transmit Channel Write Register[FifoTxWR]R/W



**Figure 5-10. Transmit Channel Write Register**

31-16: RESERVED No affect when written to. Undefined when read.

15-0: WRITE Pointer to last written byte + 1

All bits are cleared to 0 at reset.

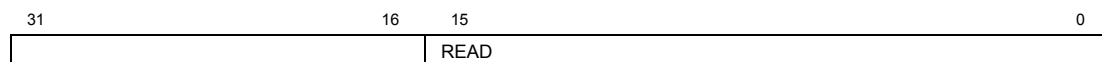
The WRITE field is written to in order to initiate a transfer, indicating the position +1 of the last byte to transmit.

Note that it is not possible to fill the buffer. There is always one word position in buffer unused. Software is responsible for not over-writing the buffer on wrap around (i.e. setting  $\text{WRITE} = \text{READ}$ ).

Note that the LSB may be ignored for 16-bit wide FIFO devices.

The field is implemented as relative to the buffer base address (scaled with the SIZE field).

### 5.7.8 Transmit Channel Read Register[FifoTxRD]R/W



**Figure 5-11. Transmit Channel Read Register**

31-16: RESERVED No affect when written to. Undefined when read.

15-0: READ Pointer to last read byte + 1

All bits are cleared to 0 at reset.

The READ field is written to automatically when a transfer has been completed successfully, indicating the position +1 of the last byte transmitted.

Note that the READ field can be used to read out the progress of a transfer.

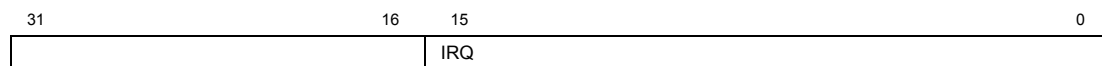
Note that the READ field can be written to in order to set up the starting point of a transfer. This should only be done while the transmit channel is not enabled.

Note that the READ field can be automatically incremented even if the transmit channel has been disabled, since the last requested transfer is not aborted until completed.

Note that the LSB may be ignored for 16-bit wide FIFO devices.

The field is implemented as relative to the buffer base address (scaled with the SIZE field).

## 5.7.9 Transmit Channel Interrupt Register[FifoTxIRQ]R/W



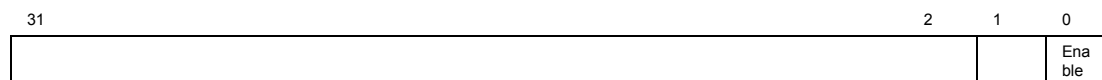
**Figure 5-12. Transmit Channel Interrupt Register**

- 31-16: RESERVED No affect when written to. Undefined when read.  
 15-0: IRQ Pointer+1 to a byte address from which the read of transmitted data shall result in an interrupt  
 All bits are cleared to 0 at reset.

Note that this indicates that a programmed amount of data has been sent. Note that the LSB may be ignored for 16-bit wide FIFO devices.

The field is implemented as relative to the buffer base address (scaled with the SIZE field).

## 5.7.10 Receive Channel Control Register[FifoRxCTRL]R/W



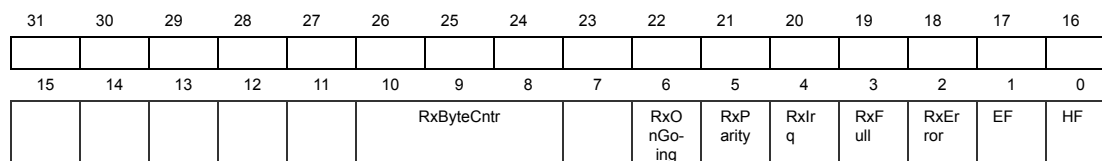
**Figure 5-13. Receive Channel Control Register**

- 31-1: RESERVED No affect when written to. Undefined when read.  
 0: ENABLE Enable channel  
 All bits are cleared to 0 at reset.

Note that in the case an AHB bus error occurs during an access while storing receive data, and the FifoConf.ABORT bit is 1b, then the ENABLE bit will be reset automatically.

At the time the ENABLE is cleared to 0b, any ongoing data reads from the FIFO are not aborted.

## 5.7.11 Receive Channel Status Register[FifoRxSTAT]R



**Figure 5-14. Receive Channel Status Register**

- 31-11: RESERVED No affect when written to. Undefined when read.  
 10-8: RxByteCntr Number of bytes in local buffer  
 6: RxOnGoing Access ongoing  
 5: RxParity Parity error during reception  
 4: RxIrq Successful reception of block of data  
 3: RxFull Reception buffer has been filled

2:	RxError	AMB AHB access error during reception
1:	EF	FIFO Empty Flag
0:	HF	FIFO Half-Full Flag

All bits are cleared to 0 at reset.

The following sticky status bits are cleared when the register has been read:

- RxParity, RxIrq, RxFull and RxError.

The circular buffer is considered as full when there are two words or less left in the buffer.

#### 5.7.12 Receive Channel Address Register[FifoRxADDR]R/W

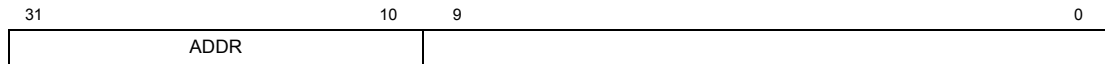


Figure 5-15. Receive Channel Address Register

31-10:	ADDR	Base address for circular buffer
9-0:	RESERVED	No affect when written to. Undefined when read.

All bits are cleared to 0 at reset.

#### 5.7.13 Receive Channel Size Register[FifoRxSIZE]R/W

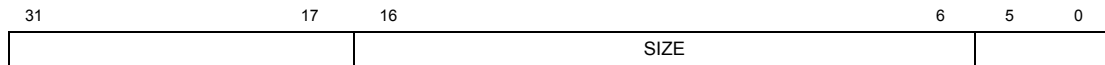


Figure 5-16. Receive Channel Size Register

31-17:	RESERVED	No affect when written to. Undefined when read.
16-6:	SIZE	Size of circular buffer, in number of 64 byte blocks
5-0:	RESERVED	No affect when written to. Undefined when read.

All bits are cleared to 0 at reset.

Valid SIZE values are 0, and between 1 and 1024. Note that the resulting behavior of invalid SIZE values is undefined.

Note that only SIZE\*64-8 bytes can be stored simultaneously in the buffer. This is to simplify wrap-around condition checking.

#### 5.7.14 Receive Channel Write Register[FifoRxWR]R/W

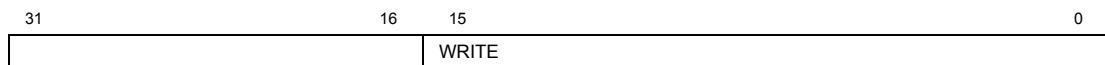


Figure 5-17. Receive Channel Write Register

31-16:	RESERVED	No affect when written to. Undefined when read.
15-0:	WRITE	Pointer to last written byte +1

All bits are cleared to 0 at reset.

The field is implemented as relative to the buffer base address (scaled with SIZE field).The WRITE field is written to automatically when a transfer has been completed successfully, indicating the position +1 of the last byte received.

Note that the WRITE field can be used to read out the progress of a transfer.

Note that the WRITE field can be written to in order to set up the starting point of a transfer. This should only be done while the transmit channel is not enabled.

Note that the LSB may be ignored for 16-bit wide FIFO devices.

5.7.15 Receive Channel Read Register[FifoRxRD]R/W

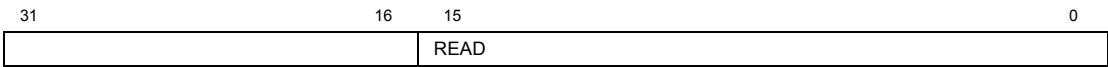


Figure 5-18. Receive Channel Read Register

- 31-16: RESERVED No affect when written to. Undefined when read.
  - 15-0: READ Pointer to last read byte +1
- All bits are cleared to 0 at reset.

The field is implemented as relative to the buffer base address (scaled with SIZE field).The READ field is written to in order to release the receive buffer, indicating the position +1 of the last byte that has been read out.

Note that it is not possible to fill the buffer. There are always at least two word positions unused in the buffer. Software is responsible for not over-reading the buffer on wrap around (i.e. setting WRITE=READ).

Note that the LSB may be ignored for 16-bit wide FIFO devices

5.7.16 Receive Channel Interrupt Register[FifoRxIRQ]R/W

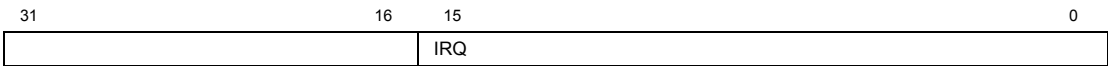


Figure 5-19. Receive Channel Interrupt Register

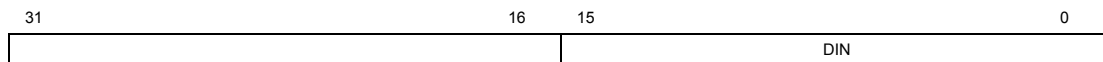
- 31-16: RESERVED No affect when written to. Undefined when read.
  - 15-0: IRQ Pointer+1 to a byte address to which the write of received data shall result in an interrupt
- All bits are cleared to 0 at reset.

Note that this indicates that a programmed amount of data has been received. The field is implemented as relative to the buffer base address (scaled with SIZE field).

Note that the LSB may be ignored for 16-bit wide FIFO devices.

Note that by setting the IRQ field to match the value of the Receive Channel Write Register.WRITE field plus the value of the Receive Channel Status Register.RxByteCnt field, an emptying to the external memory is forced of any data temporarily stored in the local buffer.

### 5.7.17 Data Input Register[FifoDIN]R

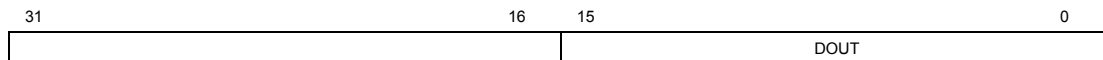


**Figure 5-20. Data Input Register**

31-16: RESERVED      No affect when written to. Undefined when read.  
 15-0: DIN              Input data *FIFOI.Din[15:0]*  
 All bits are cleared to 0 at reset.

Note that only the part of FIFOI.Din[15:0] not used by the FIFO can be used as general purpose input output, see FifoCONF.DW.  
 Note that only bits dwidth-1 to 0 are implemented.

### 5.7.18 Data Output Register[FifoDOUT]R/W

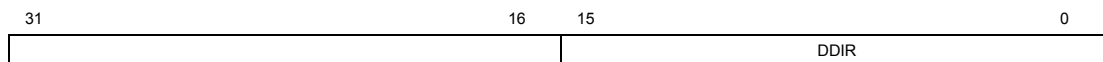


**Figure 5-21. Data Output Register**

31-16: RESERVED      No affect when written to. Undefined when read.  
 15-0: DOUT             Output data *FIFOO.Dout[15:0]*  
 All bits are cleared to 0 at reset.

Note that only the part of FIFOO.Dout[15:0] not used by the FIFO can be used as general purpose input output, see FifoCONF.DW.  
 Note that only bits dwidth-1 to 0 are implemented.

### 5.7.19 Data Register[FifoDDIR]R/W



**Figure 5-22. Data Direction Register**

31-16: RESERVED      No affect when written to. Undefined when read.  
 15-0: DDIR             Direction: *FIFOO.Dout[15:0]*  
          0b = input = high impedance,  
          1b = output = driven  
 All bits are cleared to 0 at reset.

Note that only the part of FIFOO.Dout[15:0] not used by the FIFO can be used as general purpose input output, see FifoCONF.DW.  
 Note that only bits dwidth-1 to 0 are implemented.

## 6. ADC / DAC INTERFACE

### 6.1 Overview

The block diagram shows a partitioning of the combined analogue-to-digital converter (ADC) and digital-to-analogue converter (DAC) interface.

The combined analogue-to-digital converter (ADC) and digital-to-analogue converter (DAC) interface is assumed to operate in an AMBA bus system where an APB bus is present. The AMBA APB bus is used for data access, control and status handling.

The ADC/DAC interface provides a combined signal interface to parallel ADC and DAC devices. The two interfaces are merged both at the pin/pad level as well as at the interface towards the AMBA bus. The interface supports simultaneously one ADC device and one DAC device in parallel.

Address and data signals unused by the ADC and the DAC can be used for general purpose input output, providing 0, 8, 16 or 24 channels.

The ADC interface supports 8 and 16 bit data widths. It provides chip select, read/convert and ready signals. The timing is programmable. It also provides an 8-bit address output. The ADC conversion can be initiated either via the AMBA interface or by internal or external triggers. An interrupt is generated when a conversion is completed.

The DAC interface supports 8 and 16 bit data widths. It provides a write strobe signal. The timing is programmable. It also provides an 8-bit address output. The DAC conversion is initiated via the AMBA interface. An interrupt is generated when a conversion is completed.

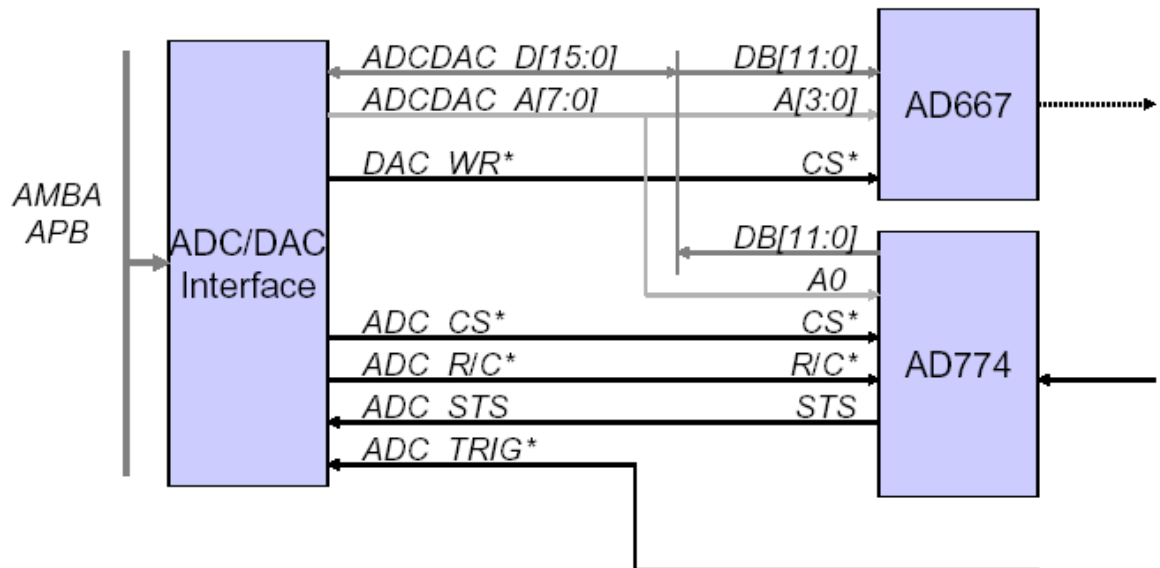


Figure 6-1. Block diagram of the GRADCDAC environment

#### 6.1.1 Function

The core implements the following functions:

- ADC interface conversion:
  - ready feed-back, or
  - timed open-loop
- DAC interface conversion:
  - timed open-loop
- General purpose input output:
  - unused data bus, and



- unused address bus
- Status and monitoring:
  - on-going conversion
  - completed conversion
  - timed-out conversion

Note that it is not possible to perform ADC and DAC conversions simultaneously. On only one conversion can be performed at a time.

### 6.1.2 Interfaces

The core provides the following external and internal interfaces:

- Combined ADC/DAC interface
  - programmable timing
  - programmable signal polarity
  - programmable conversion modes
- AMBA APB slave interface

The ADC interface is intended for amongst others the following devices:

Name:	Width:	Type:
AD574	12-bit	R/C*, CE, CS*, RDY*, tri-state
AD674	12-bit	R/C*, CE, CS*, RDY*, tri-state
AD774	12-bit	R/C*, CE, CS*, RDY*, tri-state
AD670	8-bit	R/W*, CE*, CS*, RDY, tri-state
AD571	10-bit	Blank/Convert*, RDY*, tri-state
AD1671	12-bit	Encode, RDY*, non-tri-state
LTC1414	14-bit	Convert*, RDY, non-tri-state

The DAC interface is intended for amongst others the following devices:

Name:	Width:	Type:
AD561	10-bit	Parallel-Data-in-Analogue-out
AD565	12-bit	Parallel-Data-in-Analogue-out
AD667	12-bit	Parallel-Data-in-Analogue-out, CS*
AD767	12-bit	Parallel-Data-in-Analogue-out, CS*
DAC08	8-bit	Parallel-Data-in-Analogue-out

## 6.2 Operation

### 6.2.1 Interfaces

The internal interface on the on-chip bus towards the core is a common AMBA APB slave for data access, configuration and status monitoring, used by both the ADC interface and the DAC interface.

The ADC address output and the DAC address output signals are shared on the external interface. The address signals are possible to use as general purpose input output channels. This is only realized when the address signals are not used by either ADC or DAC.

The ADC data input and the DAC data output signals are shared on the external interface. The data input and output signals are possible to use as general purpose input output channels. This is only realized when the data signals are not used by either ADC or DAC.

Each general purpose input output channel shall be individually programmed as input or output. This applies to both the address bus and the data bus. The default reset configuration for each general purpose input output channel is as input. The default reset value each general purpose input output channel is logical zero.

Note that protection toward spurious pulse commands during power up shall be mitigated as far as possible by means of I/O cell selection from the target technology.

## 6.2.2 Analogue to digital conversion

The ADC interface supports 8 and 16 bit wide input data.

The ADC interface provides an 8-bit address output, shared with the DAC interface. Note that the address timing is independent of the acquisition timing.

The ADC interface shall provide the following control signals:

- Chip Select
- Read/Convert
- Ready

The timing of the control signals is made up of two phases:

- Start Conversion
- Read Result

The Start Conversion phase is initiated by one of the following sources, provided that no other conversion is ongoing:

- Event on one of three separate trigger inputs
- Write access to the AMBA APB slave interface

Note that the trigger inputs can be connected to internal or external sources to the ASIC incorporating the core. Note that any of the trigger inputs can be connected to a timer to facilitate cyclic acquisition. The selection of the trigger source is programmable. The trigger inputs is programmable in terms of: Rising edge or Falling edge. Triggering events are ignored if ADC or DAC conversion is in progress.

The transition between the two phases is controlled by the Ready signal. The Ready input signal is programmable in terms of: Rising edge or Falling edge. The Ready input signaling is protected by means of a programmable time-out period, to assure that every conversion terminates. It is also possible to perform an ADC conversion without the use of the Ready signal, by means of a programmable conversion time duration. This can be seen as an open-loop conversion.

The Chip Select output signal is programmable in terms of:

- Polarity
- Number of assertions during a conversion, either
  - Pulsed once during Start Conversion phase only,
  - Pulsed once during Start Conversion phase and once during Read Result phase, or
  - Asserted at the beginning of the Start Conversion phase and de-asserted at the end of the Read Result phase

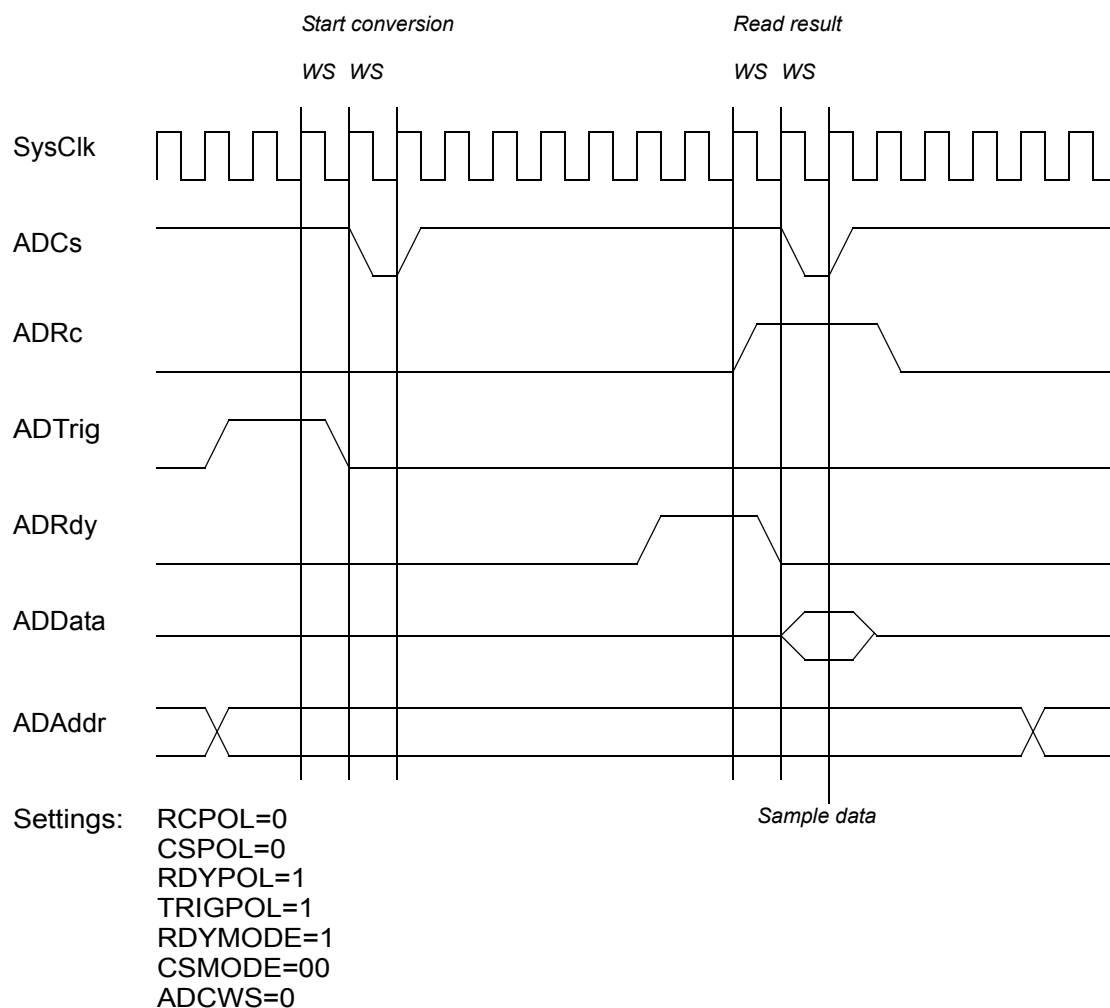
The duration of the asserted period is programmable, in terms of system clock periods, for the Chip Select signal when pulsed in either of two phases.

The Read/Convert signal is de-asserted during the Start Conversion phase, and asserted during the Read Result phase.

The Read/Convert output signal is programmable in terms of: Polarity. The setup timing from Read/Convert signal being asserted till the Chip Select signal is asserted is programmable, in terms of system clock periods. Note that the programming of Chip Select and Read/Convert timing is implemented as a common parameter.

At the end of the Read Result phase, an interrupt is generated, indicating that data is ready for readout via the AMBA APB slave interface. The status of an on-going conversion is possible to read out via the AMBA APB slave interface. The result of a conversion is read out via the AMBA APB slave interface. Note that this is independent of what trigger started the conversion.

An ADC conversion is non-interruptible. It is possible to perform at least 1000 conversions per second.



**Figure 6-2. Analogue to digital conversion waveform, 0 wait states (WS)**

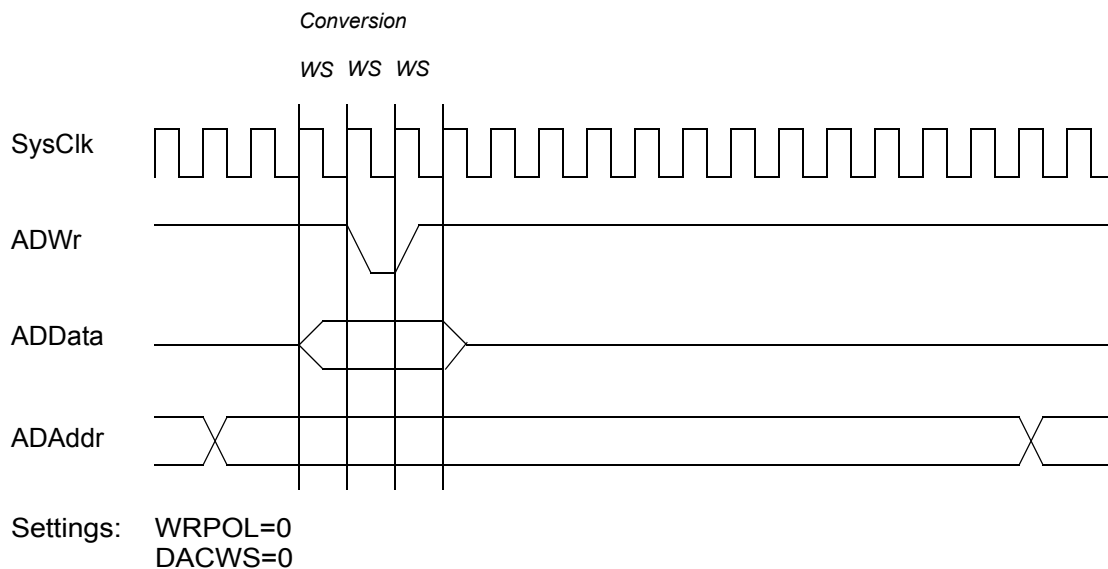
### 6.2.3 Digital to analogue conversion

The DAC interface supports 8 and 16 bit wide output data. The data output signal is driven during the conversion and is placed in high impedance state after the conversion.

The DAC interface provides an 8-bit address output, shared with the ADC interface. Note that the address timing is independent of the acquisition timing.

The DAC interface provides the following control signal: Write Strobe. Note that the Write Strobe signal can also be used as a chip select signal. The Write Strobe output signal is programmable in terms of: Polarity. The Write Strobe signal is asserted during the conversion. The duration of the asserted period of the Write Strobe is programmable in terms of system clock periods.

At the end the conversion, an interrupt is generated. The status of an on-going conversion is possible to read out via the AMBA APB slave interface. A DAC conversion is non-interruptible.



**Figure 6-3. Digital to analogue conversion waveform, 0 wait states (WS)**

## 6.2.4 Interrupt

Two interrupts are implemented by the ADC/DAC interface:

Name:	Description:
ADC	ADC conversion ready
DAC	DAC conversion ready

## 6.2.5 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x36.

## 6.3 Registers

The GRADCDAC is programmed through registers mapped into APB address space.

Register	Address
Configuration Register	0x80040000
Status Register	0x80040004
ADC Data Input Register	0x80040010
DAC Data Output Register	0x80040014
Address Input Register	0x80040020
Address Output Register	0x80040024
Address Direction Register	0x80040028
Data Input Register	0x80040030
Data Output Register	0x80040034
Data Direction Register	0x80040038

**Table 6-1. GRADCDAC registers**

Any blank register is considered as reserved and has no effect when written to, and returns undefined data when read.

### 6.3.1 Configuration Register[ADCONF]R/W

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
								DACWS					WRPOL	DACDW	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADCWS						RCPOL	CSMODE	CSPOL	RDYMODE	RDYPOL	TRIGPOL	TRIGMODE		ADCDW	

**Figure 6-4. Configuration register**

- 31-24: RESERVED      No affect when written to. Undefined when read.
- 23-19: DACWS      Number of DAC wait states, 0 to 31 [5 bits]
- 18: WRPOL      Polarity of DAC write strobe:
  - 0b = active low
  - 1b = active high
- 17-16: DACDW      DAC data width
  - 00b = none
  - 01b = 8 bitADData[7:0]
  - 10b = 16 bitADData[15:0]
  - 11b = none/spare
- 15-11: ADCWS      Number of ADC wait states, 0 to 31 [5 bits]
- 10: RCPOL      Polarity of ADC read convert:
  - 0b = active low read
  - 1b = active high read
- 9-8: CSMODE      Mode of ADC chip select:
  - 00b = asserted during conversion and read phases
  - 01b = asserted during conversion phase
  - 10b = asserted during read phase
  - 11b = asserted continuously during both phases
- 7: CSPOL      Polarity of ADC chip select:
  - 0b = active low
  - 1b = active high
- 6: RDYMODE      Mode of ADC ready:
  - 0b = unused, i.e. open-loop
  - 1b = used, with time-out
- 5: RDYPOL      Polarity of ADC ready:
  - 0b = falling edge
  - 1b = rising edge
- 4: TRIGPOL      Polarity of ADC triggers:
  - 0b = falling edge
  - 1b = rising edge
- 3-2: TRIGMODE      ADC trigger source:
  - 00b = none

- 01b = ADTrig
- 10b = 32-bit Timer 1
- 11b = 32-bit Timer 2
- 1-0: ADCDW ADC data width:
  - 00b = none
  - 01b = 8 bitADData[7:0]
  - 10b = 16 bitADData[15:0]
  - 11b = none/spare

The ADCDW field defines what part of ADData[15:0] is read by the ADC. The DACDW field defines what part of ADData[15:0] is written by the DAC. Parts of the data input/output signals used neither by ADC nor by DAC are available for the general purpose input output functionality. Note that an ADC conversion can be initiated by means of a write access via the AMBA APB slave interface, thus not requiring an explicit ADC trigger source setting.

The ADCONF.ADCWS field defines the number of system clock periods, ranging from 1 to 32, for the following timing relationships between the ADC control signals:

- ADRC stable before ADCS period
- ADCS asserted period, when pulsed
- ADTrig[2:0] event until ADCS asserted period
- Time-out period for ADRdy:  $2048 * (1 + \text{ADCONF.ADCWS})$
- Open-loop conversion timing:  $512 * (1 + \text{ADCONF.ADCWS})$

The ADCONF.DACWS field defines the number of system clock periods, ranging from 1 to 32, for the following timing relationships between the DAC control signals:

- ADData[15:0] stable before ADWR period
- ADWR asserted period
- ADData[15:0] stable after ADWR period

### 6.3.2 Status Register[ADSTAT]R

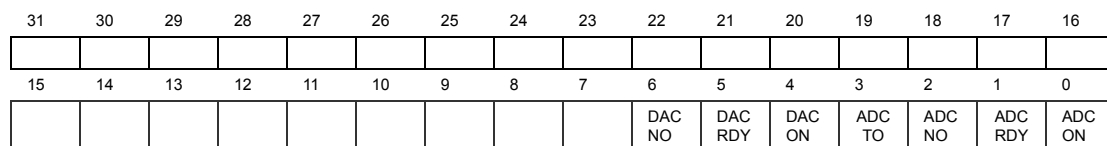


Figure 6-5. Status register

- |       |          |  |
|-------|----------|--|
| 31-7: | RESERVED | No affect when written to. Undefined when read.                        |
| 6:    | DACNO    | DAC conversion request rejected (due to ongoing DAC or ADC conversion) |
| 5:    | DACRDY   | DAC conversion completed   |
| 4:    | DACON    | DAC conversion ongoing   |
| 3:    | ADCTO    | ADC conversion timeout   |
| 2:    | ADCNO    | ADC conversion request rejected (due to ongoing ADC or DAC conversion) |
| 1:    | ADCRDY   | ADC conversion completed   |
| 0:    | ADCON    | ADC conversion ongoing   |

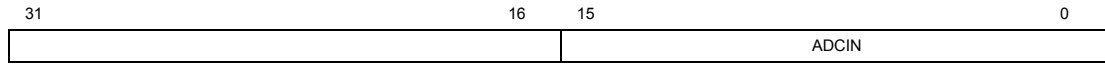
When the register is read, the following sticky bit fields are cleared:

- DACNO, DACRDY,

- ADCTO, ADCNO, and ADCRDY.

Note that the status bits can be used for monitoring the progress of a conversion or to ascertain that the interface is free for usage.

### 6.3.3 ADC Data Input Register[ADIN] R/W



**Figure 6-6. ADC Data Input Register**

31-16:	RESERVED	No affect when written to. Undefined when read.
15-0:	ADCIN	ADC input data <i>ADData[15:0]</i>

A write access to the register initiates an analogue to digital conversion, provided that no other ADC or DAC conversion is ongoing (otherwise the request is rejected).

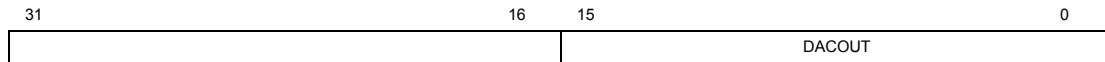
A read access that occurs before an ADC conversion has been completed returns the result from a previous conversion.

Note that any data can be written and that it cannot be read back, since not relevant to the initiation of the conversion.

Note that only the part of *ADData[15:0]* that is specified by means of bit *ADCONF.ADCDW* is used by the ADC. The rest of the bits are read as zeros.

Note that only bits 15 to 0 are implemented.

### 6.3.4 DAC Data Output Register[ADOUT]R/W



**Figure 6-7. DAC Data Output Register**

31-16:	RESERVED	No affect when written to. Undefined when read.
15-0:	DACOUT	DAC output data <i>ADData[15:0]</i>

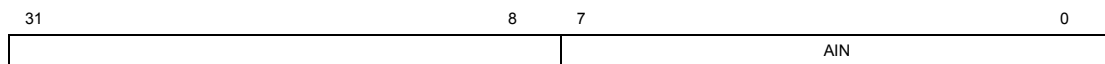
A write access to the register initiates a digital to analogue conversion, provided that no other DAC or ADC conversion is ongoing (otherwise the request is rejected).

Note that only the part of *ADData[15:0]* that is specified by means of *ADCONF.DACDW* is driven by the DAC. The rest of the bits are not driven by the DAC during a conversion.

Note that only the part of *ADData[15:0]* which is specified by means of *ADCONF.DACDW* can be read back, whilst the rest of the bits are read as zeros.

Note that only bits 15 to 0 are implemented.

### 6.3.5 Address Input Register[ADAIN]R

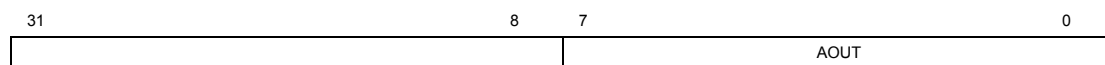


**Figure 6-8. Address Input Register**

31-8: RESERVED No affect when written to. Undefined when read.  
 7-0: AIN Input address *ADAddr[7:0]*  
 All bits are cleared to 0 at reset.

Note that only bits 7 to 0 are implemented.

### 6.3.6 Address Output Register[ADAOUT]R/W

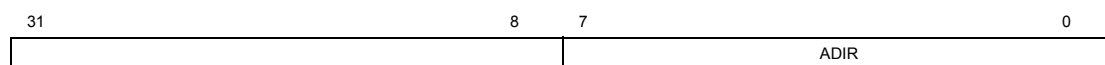


**Figure 6-9. Address Output Register**

31-8: RESERVED No affect when written to. Undefined when read.  
 7-0: AOUT Output address *ADAddr[7:0]*  
 All bits are cleared to 0 at reset.

Note that only bits 7 to 0 are implemented.

### 6.3.7 Address Direction Register[ADADIR]R/W

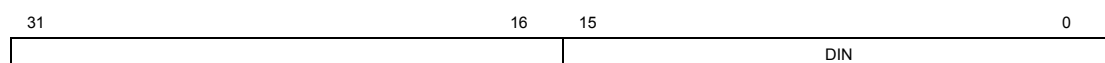


**Figure 6-10. Address Direction Register**

31-8: RESERVED No affect when written to. Undefined when read.  
 7-0: ADIR Direction: *ADAddr[7:0]*  
     0b = input = high impedance,  
     1b = output = driven  
 All bits are cleared to 0 at reset.

Note that only bits 7 to 0 are implemented.

### 6.3.8 Data Input Register[ADDIN]R



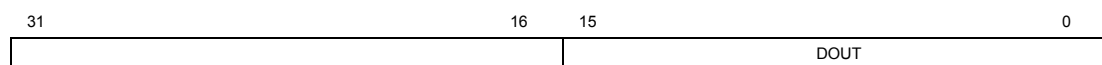
**Figure 6-11. Data Input Register**



31-16: RESERVED      No affect when written to. Undefined when read.  
 15-0: DIN              Input data      *ADData[15:0]*  
 All bits are cleared to 0 at reset.

Note that only the part of *ADData[15:0]* not used by the ADC can be used as general purpose input output, see *ADCONF.ADCDW*.  
 Note that only bits 15 to 0 are implemented.

### 6.3.9 Data Output Register[*ADDOUT*]R/W

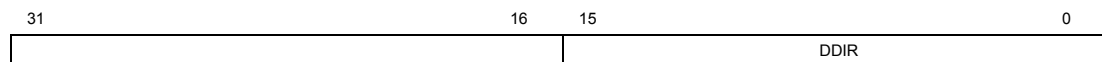


**Figure 6-12. Data Output Register**

31-16: RESERVED      No affect when written to. Undefined when read.  
 15-0: DOUT              Output data      *ADData[15:0]*  
 All bits are cleared to 0 at reset.

Note that only the part of *ADData[15:0]* neither used by the DAC nor the ADC can be used as general purpose input output, see *ADCONF.DACDW* and *ADCONF.ADCDW*.  
 Note that only bits 15 to 0 are implemented.

### 6.3.10 Data Register[*ADDDIR*]R/W



**Figure 6-13. Data Direction Register**

31-16: RESERVED      No affect when written to. Undefined when read.  
 15-0: DDIR              Direction:      *ADData[15:0]*  
     0b = input = high impedance,  
     1b = output = driven  
 All bits are cleared to 0 at reset.

Note that only the part of *ADData[15:0]* not used by the DAC can be used as general purpose input output, see *ADCONF.DACDW*.  
 Note that only bits 15 to 0 are implemented.

## 7. 32-BIT TIMERS

### 7.1 Overview

The Modular Timer Unit implements one prescaler and two decrementing timers. The unit is capable of asserting interrupt on when timer(s) underflow. Interrupt is configured to be separate for each timer.

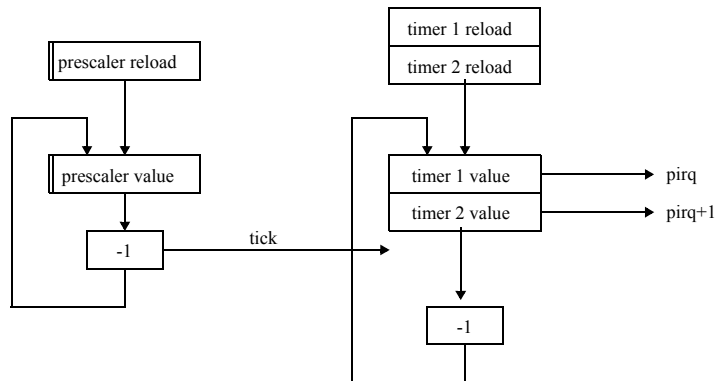


Figure 7-1. General Purpose Timer Unit block diagram

### 7.2 Operation

The prescaler is clocked by the system clock and decremented on each clock cycle. When the prescaler underflows, it is reloaded from the prescaler reload register and a timer tick is generated. Timers share the decrements to save area.

The operation of each timer is controlled through its control register. A timer is enabled by setting the enable bit in the control register. The timer value is then decremented on each prescaler tick. When a timer underflows, it will automatically be reloaded with the value of the corresponding timer reload register if the restart bit in the control register is set, otherwise it will stop at -1 and reset the enable bit.

Since configured to signal interrupt for each timer the timer unit will signal an interrupt on appropriate line when a timer underflows (if the interrupt enable bit for the current timer is set). The interrupt pending bit in the control register of the underflow timer will be set and remain set until cleared by writing '0'.

To minimize complexity, timers share the same decrements. This means that the minimum allowed prescaler division factor is 3 (reload register = 2) where 2 is the number of implemented timers.

By setting the chain bit in the control register timer  $n$  can be chained with preceding timer  $n-1$ . Decrementing timer  $n$  will start when timer  $n-1$  underflows.

Each timer can be reloaded with the value in its reload register at any time by writing a 'one' to the load bit in the control register.

### 7.3 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x38.

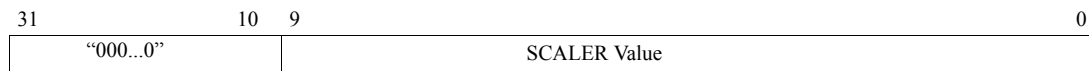
## 7.4 Registers

Here is the list of registers implemented in the timer unit registers.

Register	Address
Scaler value	0x80030000
Scaler reload value	0x80030004
Configuration register	0x80030008
Timer latch configuration register	0x8003000C
Timer 1 counter value register	0x80030010
Timer 1 reload value register	0x80030014
Timer 1 control register	0x80030018
Timer 1 latch register	0x8003001C
Timer 2 counter value register	0x80030020
Timer 2 reload value register	0x80030024
Timer 2 control register	0x80030028
Timer 2 latch register	0x8003002C

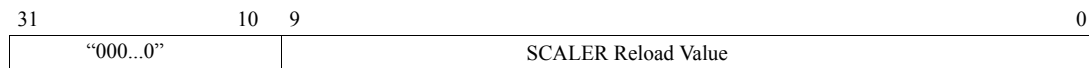
**Table 7-1. GRTIMER unit registers**

The following figures show the layout of the timer unit registers. Any blank register is considered as reserved and has no effect when written to, and returns undefined data when read.



**Figure 7-2. Scaler value**

[31:10] Reserved      No effect when written to. Undefined when read.



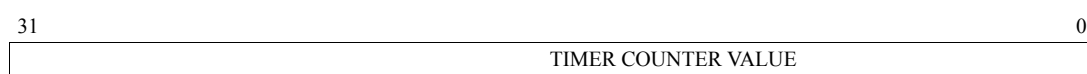
**Figure 7-3. Scaler reload value**

[31:10] Reserved      No effect when written to. Undefined when read.



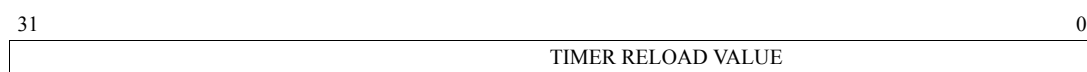
**Figure 7-4. GRTIMER Configuration register**

[31:12]	Reserved.	No effect when written to. Undefined when read.
[11]	Enable latching (EL).	If set, on the next matching interrupt, the latches will be loaded with the corresponding timer values. The bit is then automatically cleared, not to load a timer value until set again.
[10]	Enable external clock source (EE).	If set the prescaler is clocked from the external clock source.
[9]	Disable timer freeze (DF)	If set the timer unit cannot be freed, otherwise DSU freezes the timer unit.
[8]	Separate interrupts (SI)	Reads '1' if the timer unit generates separate interrupts for each timer, otherwise '0'. Read-only.
[7:3]	APB Interrupt	Timer $n$ will drive APB Interrupt $IRQ+n$ . Read-only.
[2:0]	Number of implemented timers.	Read-only.



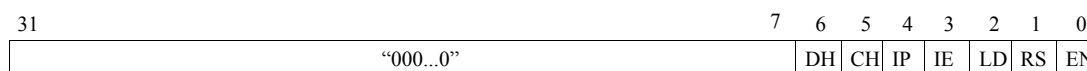
**Figure 7-5. Timer counter value registers**

[31:0] Timer Counter value. Decrementd by 1 for each prescaler tick.



**Figure 7-6. Timer reload value registers**

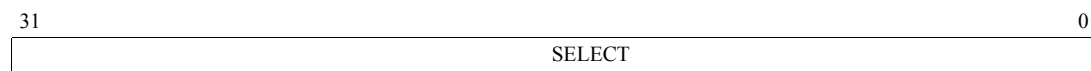
[31:0]	Timer Reload value.	This value is loaded into the timer counter value register when '1' is written to load bit in the timers control register.
--------	---------------------	--



**Figure 7-7. Timer control registers**

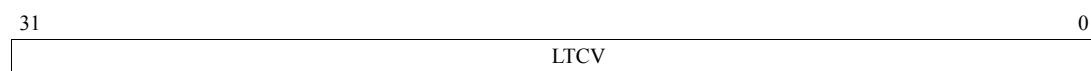
[31:7]	Reserved.	No effect when written to. Undefined when read.
[6]	Debug Halt (DH)	Value of GPTI.DHALT signal which is used to freeze counters (e.g. when a system is in debug mode). Read-only.
[5]	Chain (CH)	Chain with preceding timer. If set for timer $n$ , decrementing timer $n$ begins when timer $(n-1)$ underflows.
[4]	Interrupt Pending (IP)	Sets when an interrupt is signalled. Remains '1' until cleared by writing '0' to this bit.
[3]	Interrupt Enable (IE)	If set the timer signals interrupt when it underflows.
[2]	Load (LD)	Load value from the timer reload register to the timer counter value register.

[1]	Restart (RS)	If set the value from the timer reload register is loaded to the timer counter value register and decrementing the timer is restarted.
[0]	Enable (EN)	Enable the timer.



**Figure 7-8. Timer latch configuration register**

[31:0]	Specifies what bits of the AMBA APB interrupt bus shall cause the Timer Latch Register to latch the timer values.
--------	---



**Figure 7-9. Timer latch register**

[31:0]	Latched Timer Counter Value (LTCV).	Value latch from corresponding timer
--------	-------------------------------------	--------------------------------------

## 8. 24-BIT GENERAL PURPOSE INPUT OUTPUT

### 8.1 Overview

The General Purpose Input Output interface is assumed to operate in an AMBA bus system where the APB bus is present. The AMBA APB bus is used for control and status handling. The General Purpose Input Output interface provides a configurable number of channels. Each channel is individually programmed as input or output. Additionally, a configurable number of the channels are also programmable as pulse command outputs. The default reset configuration for each channel is as input. The default reset value each channel is logical zero.

The pulse command outputs have a common counter for establishing the pulse command length. The pulse command length defines the logical one (active) part of the pulse. It is possible to select which of the channels shall generate a pulse command. The pulse command outputs are generated simultaneously in phase with each other, and with the same length (or duration). It is not possible to generate pulse commands out of phase with each other. Each channel can generate a separate internal interrupt. The interrupt are individually programmed as enabled or disabled, as active high or low level sensitive, or as rising or falling edge sensitive.

#### 8.1.1 Function

The core implements the following functions:

- Input and input interrupts
- Output and output pulse commands
- Status and monitoring

#### 8.1.2 Interfaces

The core provides the following external and internal interfaces:

- Discrete input and output interface
- AMBA APB slave interface, with sideband signals as per [GRLIB] including:
  - interrupt bus
  - configuration information
  - diagnostic information

#### 8.1.3 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x37.

### 8.2 Registers

The GRPULSE is programmed through registers mapped into APB address space.

Register	Address
Input Register	0x80020000
Output Register	0x80020004
Direction Register	0x80020008
Interrupt Mask Register	0x8002000C
Interrupt Polarity Register	0x80020010
Interrupt Edge Register	0x80020014
Pulse Register	0x80020018
Pulse Counter Register	0x8002001C

Table 8-1. GRPULSE registers

Any blank register is considered as reserved and has no effect when written to, and returns undefined data when read.

8.2.1 Input Register[GpioIN]R

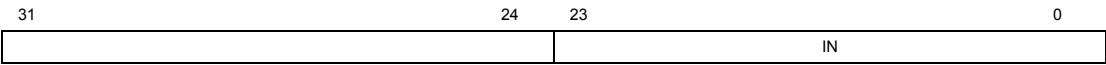


Figure 8-1. Input Register

- 31-24 Reserved. No effect when written to. Undefined when read.
- 23-0 IN Input Data

Note that only bits 23 to 0 are implemented.

8.2.2 Output Register[GpioOUT]R/W

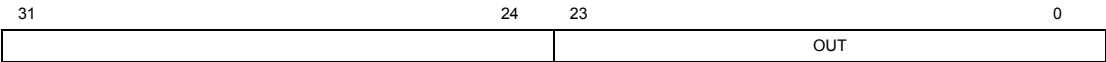


Figure 8-2. Output Register

- 31-24 Reserved. No effect when written to. Undefined when read.
- 23-0 OUT Output Data

All bits are cleared to 0 at reset.

Note that only bits 23 to 0 are implemented.

8.2.3 Direction Register[GpioDIR]R/W

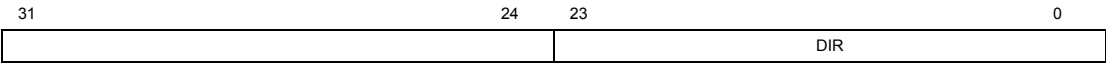


Figure 8-3. Direction Register

- 31-24: Reserved. No effect when written to. Undefined when read.
- 23-0: DIR Direction:

0b=input,  
1b=output

All bits are cleared to 0 at reset.

Note that only bits 23 to 0 are implemented.

8.2.4 Pulse Register[GpioPULSE]R/W

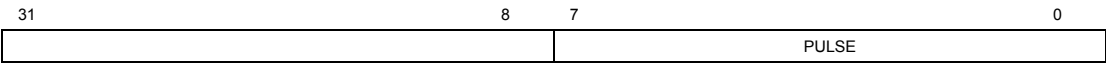


Figure 8-4. Pulse Register

- 31-24 Reserved. No effect when written to. Undefined when read.

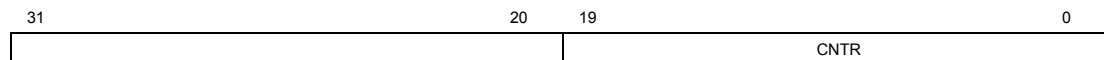
23-0 PULSE Pulse enable:  
0b=output,  
1b=pulse command output

All bits are cleared to 0 at reset.

Only channels configured as outputs are possible to enable as command pulse outputs.

Note that only bits 7 to 0 are implemented.

### 8.2.5 Pulse Counter Register[GpioCNTR]R/W



**Figure 8-5. Pulse Counter Register**

31-24 Reserved. No effect when written to. Undefined when read.

23-0 CNTR Pulse counter value

All bits are cleared to 0 at reset.

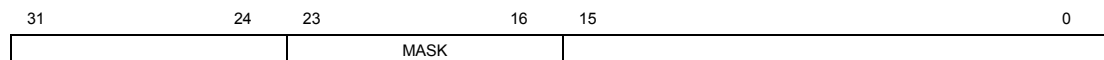
The pulse counter is decremented each clock period, and does not wrap after reaching zero.

Command pulse channels, with the corresponding output data and pulse enable bits set, are asserted while the pulse counter is greater than zero.

- Setting CNTR to 0 does not give a pulse.
- Setting CNTR to 1 does give a pulse with of 1 Clk period.
- Setting CNTR to 255 does give a pulse with of 255 Clk periods.

Note that only bits 19 to 0 are implemented.

### 8.2.6 Interrupt Mask Register[GpioMASK]R/W



**Figure 8-6. Interrupt Mask Register**

31-24 Reserved. No effect when written to. Undefined when read.

23-16 MASK Interrupt enable,  
0b=disable,  
1b=enable

15-0 Reserved. No effect when written to. Undefined when read.

All bits are cleared to 0 at reset.

Note that only bits 23 to 16 are implemented and are mapped on interrupts 31 to 24.



8.2.7 Interrupt Polarity Register[GpioPOL]R/W

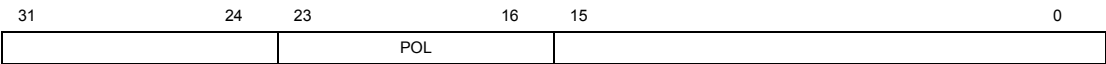


Figure 8-7. Interrupt Polarity Register

- 31-24 Reserved. No effect when written to. Undefined when read.
  - 23-16 POL Interrupt polarity, 0b=active low or falling edge, 1b=active high or rising edge
  - 15-0 Reserved. No effect when written to. Undefined when read.
- All bits are cleared to 0 at reset.

Note that only bits 23 to 16 are implemented and are mapped on interrupts 31 to 24.

8.2.8 Interrupt Edge Register[GpioEDGE]R/W

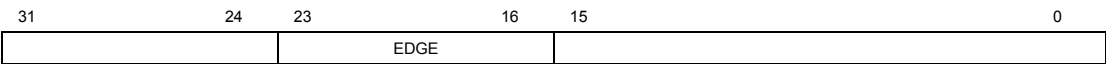


Figure 8-8. Interrupt Edge Register

- 31-24 Reserved. No effect when written to. Undefined when read.
  - 23-16 EDGE) Interrupt edge or level, 0b=level, 1b=edge
  - 15-0 Reserved. No effect when written to. Undefined when read.
- All bits are cleared to 0 at reset.

Note that only bits 23 to 16 are implemented and are mapped on interrupts 31 to 24.

Note that the secondary interrupt controller uses edge detection. The Interrupt Edge Register must therefore only be programmed for edge detection, not for level, to ensure that multiple interrupts can be detected from the same source.

## 9. CAN INTERFACE

### 9.1 Overview

The CAN controller is assumed to operate in an AMBA bus system where both the AMBA AHB bus and the APB bus are present. The AMBA APB bus is used for configuration, control and status handling. The AMBA AHB bus is used for retrieving and storing CAN messages in memory external to the CAN controller. This memory can be located on-chip, as shown in the block diagram, or external to the chip. The CAN protocol is based on the ESA HurriCANE CAN Controller core.

The CAN controller supports transmission and reception of sets of messages by use of circular buffers located in memory external to the core. Separate transmit and receive buffers are assumed. Reception and transmission of sets of messages can be ongoing simultaneously.

After a set of message transfers has been set up via the AMBA APB interface the DMA controller initiates a burst of read accesses on the AMBA AHB bus to fetch messages from memory, which are performed by the AHB master. The messages are then transmitted by the ESA HurriCANE CAN Controller. When a programmable number of messages have been transmitted, the DMA controller issues an interrupt.

After the reception has been set up via the AMBA APB interface, messages are received by the ESA HurriCANE CAN Controller. To store messages to memory, the DMA controller initiates a burst of write accesses on the AMBA AHB bus, which are performed by the AHB master. When a programmable number of messages have been received, the DMA controller issues an interrupt.

The CAN controller can detect a SYNC message and generate an interrupt. The SYNC message identifier is programmable via the AMBA APB interface. The CAN controller supports the draft ECSS high-resolution time distribution protocol.

The CAN controller can transmit and receive messages on either of two CAN busses, but only on one at a time. The selection is programmable via the AMBA APB interface. Note that it is not possible to receive a CAN message while transmitting one.

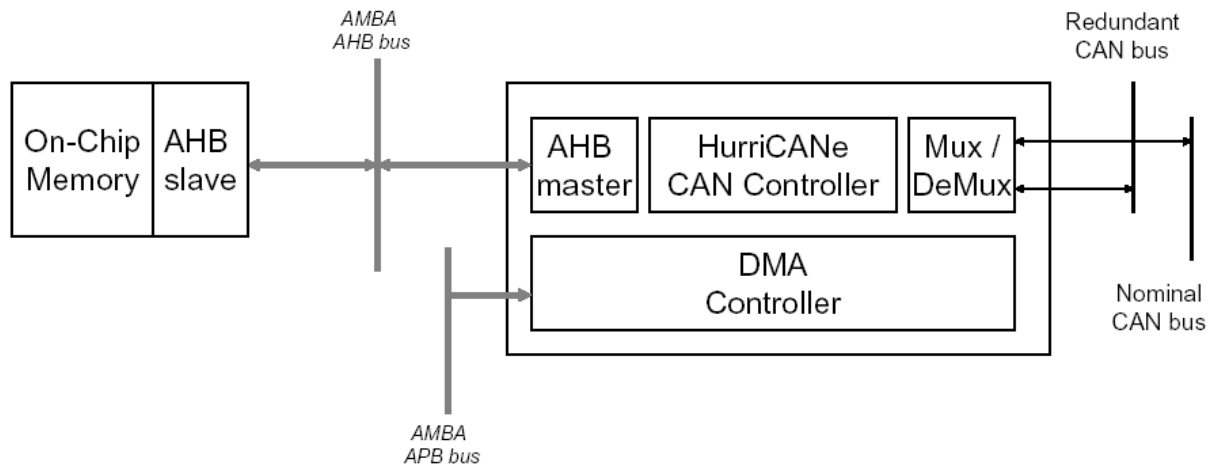


Figure 9-1. Block diagram of the internal structure of the GRHCAN.

#### 9.1.1 Function

The core implements the following functions:

- CAN protocol
- Message transmission
- Message filtering and reception
- SYNC message reception
- Status and monitoring
- Interrupt generation
- Redundancy selection

### 9.1.2 Interfaces

The core provides the following external and internal interfaces:

- CAN interface
- AMBA AHB master interface, with sideband signals as per [GRLIB] including:
  - cacheability information
  - interrupt bus
  - configuration information
  - diagnostic information
- AMBA APB slave interface, with sideband signals as per [GRLIB] including:
  - interrupt bus
  - configuration information
  - diagnostic information

### 9.1.3 Hierarchy

The CAN controller core can be partitioned in the following hierarchical elements:

- ESA HurriCANE CAN Controller
- Redundancy Multiplexer / De-multiplexer
- Direct Memory Access controller
- AMBA APB slave
- AMBA AHB master

## 9.2 Interface

The external interface towards the CAN bus features two redundant pairs of transmit output and receive input (i.e. 0 and 1). The active pair (i.e. 0 or 1) is bselectable by means of a configuration register bit. Note that all reception and transmission is made over the active pair.

For each pair, there is one enable output (i.e. 0 and 1), each being individually programmable. Note that the enable outputs can be used for enabling an external physical driver. Note that both pairs can be enabled simultaneously. Note that the polarity for the enable/inhibit inputs on physical interface drivers differs, thus the meaning of the enable output is undefined.

Redundancy is implemented by means of Selective Bus Access, as specified in [CANWG]. Note that the active pair selection above provides means to meet this requirement.

## 9.3 Protocol

The CAN protocol is based on the ESA HurriCANE CAN bus controller core described in [CANESA]. The CAN controller complies with [CANSTD], except for the overload frame generation. Note that the ESA HurriCANE CAN bus controller core does not implement overload frame generation. Version 5.1, dated 18 May 2005, has been used. No other CAN protocol capabilities than those implement by the ESA HurriCANE CAN bus controller core are provided.

The Remote Frame Response function implemented by the ESA HurriCANE CAN bus controller is no implemented. The remote frame response is instead envisaged to be implemented by means of processor support and software.

Note that there are three different CAN types generally defined:

- 2.0A, which considers 29 bit ID messages as an error
- 2.0B Passive, which ignores 29 bit ID messages
- 2.0B Active, which handles 11 and 29 bit ID messages

Only 2.0B Active is implemented.

## 9.4 Status and monitoring

The CAN interface provides status and monitoring functionalities, including:

- Transmitter active indicator

- Bus-Off condition indicator
- Error-Passive condition indicator
- Over-run indicator
- 8-bit Transmission error counter
- 8-bit Reception error counter

The status is available via a register and is stored in a circular buffer for each received message.

## 9.5 Transmission

The transmit channel is defined by the following parameters:

- base address
- buffer size
- write pointer
- read pointer

The transmit channel can be enabled or disabled.

### 9.5.1 Circular buffer

The transmit channel operates on a circular buffer located in memory external to the CAN controller. The circular buffer can also be used as a straight buffer. The buffer memory is accessed via the AMBA AHB master interface.

Each CAN message occupies 4 consecutive 32-bit words in memory. Each CAN message is aligned to 4 words address boundaries (i.e. the 4 least significant byte address bits are zero for the first word in a CAN message).

The size of the buffer is defined by the CanTxSIZE.SIZE field, specifying the number of CAN messages \* 4 that fit in the buffer.

E.g. CanTxSIZE.SIZE =2 means 8 CAN messages fit in the buffer.

Note however that it is not possible to fill the buffer completely, leaving at least one message position in the buffer empty. This is to simplify wrap-around condition checking.

E.g. CanTxSIZE.SIZE =2 means that 7 CAN messages fit in the buffer at any given time.

### 9.5.2 Write and read pointers

The write pointer (CanTxWR.WRITE) indicates the position+1 of the last CAN message written to the buffer. The write pointer operates on number of CAN messages, not on absolute or relative addresses.

The read pointer (CanTxRD.READ) indicates the position+1 of the last CAN message read from the buffer. The read pointer operates on number of CAN messages, not on absolute or relative addresses.

The difference between the write and the read pointers is the number of CAN messages available in the buffer for transmission. The difference is calculated using the buffer size, specified by the CanTxSIZE.SIZE field, taking wrap around effects of the circular buffer into account.

Examples:

- There are 2 CAN messages available for transmit when CanTxSIZE.SIZE=2, CanTxWR.WRITE=2 and CanTxRD.READ=0.
- There are 2 CAN messages available for transmit when CanTxSIZE.SIZE=2, CanTxWR.WRITE =0 and CanTxRD.READ =6.
- There are 2 CAN messages available for transmit when CanTxSIZE.SIZE=2, CanTxWR.WRITE =1 and CanTxRD.READ =7.
- There are 2 CAN messages available for transmit when CanTxSIZE.SIZE=2, CanTxWR.WRITE =5 and CanTxRD.READ =3.

When a CAN message has been successfully transmitted, the read pointer (CanTxRD.READ) is automatically incremented, taking wrap around effects of the circular buffer into account. Whenever the write pointer CanTxWR.WRITE and read pointer CanTxRD.READ are equal, there are no CAN messages available for transmission.

### 9.5.3 Location

The location of the circular buffer is defined by a base address (CanTxADDR.ADDR), which is an absolute address. The location of a circular buffer is aligned on a 1kbyte address boundary.

### 9.5.4 Transmission procedure

When the channel is enabled (`CanTxCTRL.ENABLE=1`), as soon as there is a difference between the write and read pointer, a message transmission will be started. Note that the channel should not be enabled if a potential difference between the write and read pointers could be created, to avoid the message transmission to start prematurely.

A message transmission will begin with a fetch of the complete CAN message from the circular buffer to a local fetch-buffer in the CAN controller. After a successful data fetch, a transmission request will be forwarded to the HurriCANE codec. If there is at least an additional CAN message available in the circular buffer, a prefetch of this CAN message from the circular buffer to a local prefetch-buffer in the CAN controller will be performed. The CAN controller can thus hold two CAN messages for transmission: one in the fetch buffer, which is fed to the HurriCANE codec, and one in the prefetch buffer.

After a message has been successfully transmitted, the prefetch-buffer contents are moved to the fetch buffer (provided that there is message ready). The read pointer (`CanTxRD.READ`) is automatically incremented after a successful transmission, i.e. after the fetch-buffer contents have been transmitted, taking wrap around effects of the circular buffer into account. If there is at least an additional CAN message available in the circular buffer, a new prefetch will be performed.

If the write and read pointers are equal, no more prefetches and fetches will be performed, and transmission will stop.

If the single shot mode is enabled for the transmit channel (`CanTxCTRL.SINGLE=1`), any message for which the arbitration is lost, or failed for some other reason, will lead to the disabling of the channel (`CanTxCTRL.ENABLE=0`), and the message will not be put up for re-arbitration.

Interrupts are provided to aid the user during transmission, as described in detail later in this section. The main interrupts are the Tx, TxEmpty and TxIrq which are issued on the successful transmission of a message, when all messages have been transmitted successfully and when a predefined number of messages have been transmitted successfully. The TxLoss interrupt is issued whenever transmission arbitration has been lost, could also be caused by a communications error. The TxSync interrupt is issued when a message matching the SYNC Code Filter Register.SYNC and SYNC Mask Filter Register.MASK registers is successfully transmitted. Additional interrupts are provided to signal error conditions on the CAN bus and AMBA bus.

### 9.5.5 Straight buffer

It is possible to use the circular buffer as a straight buffer, with a higher granularity than the 1kbyte address boundary limited by the base address (`CanTxADDR.ADDR`) field.

While the channel is disabled, the read pointer (`CanTxRD.READ`) can be changed to an arbitrary value pointing to the first message to be transmitted, and the write pointer (`CanTxWR.WRITE`) can be changed to an arbitrary value.

When the channel is enabled, the transmission will start from the read pointer and continue to the write pointer.

### 9.5.6 AMBA AHB error

Definition:

- a message fetch occurs when no other messages is being transmitted
- a message prefetch occurs when a previously fetched message is being transmitted
- the local fetch buffer holds the message being fetched
- the local prefetch buffer holds the message being prefetched
- the local fetch buffer holds the message being transmitted by the HurriCANE codec
- a successfully prefetched message is copied from the local prefetch buffer to the local fetch buffer when that buffer is freed after a successful transmission.

An AHB error response occurring on the AMBA AHB bus while a CAN message is being fetched will result in a TxAHBErr interrupt.

If the `CanCONF.ABORT` bit is set to 0b, the channel causing the AHB error will skip the message being fetched from memory and will increment the read pointer. No message will be transmitted.

If the `CanCONF.ABORT` bit is set to 1b, the channel causing the AHB error will be disabled (`CanTxCTRL.ENABLE` is cleared automatically to 0 b). The read pointer can be used to determine which message caused the AHB error. Note that it could be any of the four word accesses required to read a message that caused the AHB error.

If the `CanCONF.ABORT` bit is set to 1b, all accesses to the AMBA AHB bus will be disabled after an AMBA AHB error occurs, as indicated by the `CanSTAT.AHBErr` bit being 1b. The accesses will be disabled until the `CanSTAT` register is read, and automatically clearing bit `CanSTAT.AHBErr`.

An AHB error response occurring on the AMBA AHB bus while a CAN message is being prefetched will not cause an interrupt, but will stop the ongoing prefetch and further prefetch will be prevented temporarily. The ongoing transmission of a CAN message from the fetch buffer will not be affected. When the fetch buffer is freed after a successful transmission, a new fetch will be initiated, and if this fetch results in an AHB error response occurring on the AMBA AHB bus, this will be handled as for the case above. If no AHB error occurs, prefetch will be allowed again.

### 9.5.7 Enable and disable

When an enabled transmit channel is disabled (CanTxCTRL.ENABLE=0b), any ongoing CAN message transfer request will not be aborted until a CAN bus arbitration is lost or the message has been sent successfully. If the message is sent successfully, the read pointer (CanTxRD.READ) is automatically incremented. Any associated interrupts will be generated.

The progress of the any ongoing access can be observed via the CanTxCTRL.ONGOING bit. The CanTxCTRL.ONGOING must be 0b before the channel can be re-configured safely (i.e. changing address, size or read/write pointers). It is also possible to wait for the Tx and TxLoss interrupts described hereafter.

The channel can be re-enabled again without the need to reconfigure the address, size and pointers.

Priority inversion is handled by disabling the transmitting channel, i.e. setting CanTxCTRL.ENABLE=0 b as described above, and observing the progress, i.e. reading via the CanTxCTRL.ONGOING bit as described above. When the transmit channel is disabled, it can be re-configured and a higher priority message can be transmitted. Note that the single shot mode does not require the channel to be disabled, but the progress should still be observed as above.

No message transmission is started while the channel is not enabled.

### 9.5.8 Interrupts

During transmission several interrupts can be generated:

- TxLoss:Message arbitration lost for transmit (could be caused by communications error, as indicated by other interrupts as well)
- TxErrCnt:Error counter incremented for transmit
- TxSync:Synchronization message transmitted
- Tx:Successful transmission of one message
- TxEmpty:Successful transmission of all messages in buffer
- TxIrq:Successful transmission of a predefined number of messages
- TxAHBErr:AHB access error during transmission
- Off:Bus-off condition
- Pass:Error-passive condition

The Tx, TxEmpty and TxIrq interrupts are only generated as the result of a successful message transmission, after the CanTxRD.READ pointer has been incremented.

## 9.6 Reception

The receive channel is defined by the following parameters:

- base address
- buffer size
- write pointer
- read pointer

The receive channel can be enabled or disabled.

### 9.6.1 Circular buffer

The receive channel operates on a circular buffer located in memory external to the CAN controller. The circular buffer can also be used as a straight buffer. The buffer memory is accessed via the AMBA AHB master interface.

Each CAN message occupies 4 consecutive 32-bit words in memory. Each CAN message is aligned to 4 words address boundaries (i.e. the 4 least significant byte address bits are zero for the first word in a CAN message).

The size of the buffer is defined by the CanRxSIZE.SIZE field, specifying the number of CAN messages \* 4 that fit in the buffer.

E.g. CanRxSIZE.SIZE=2 means 8 CAN messages fit in the buffer.

Note however that it is not possible to fill the buffer completely, leaving at least one message position in the buffer empty. This is to simplify wrap-around condition checking.

E.g. CanRxSIZE.SIZE=2 means that 7 CAN messages fit in the buffer at any given time.

### 9.6.2 Write and read pointers

The write pointer (CanRxWR.WRITE) indicates the position+1 of the last CAN message written to the buffer. The write pointer operates on number of CAN messages, not on absolute or relative addresses.

The read pointer (CanRxRD.READ) indicates the position+1 of the last CAN message read from the buffer. The read pointer operates on number of CAN messages, not on absolute or relative addresses.

The difference between the write and the read pointers is the number of CAN message positions available in the buffer for reception. The difference is calculated using the buffer size, specified by the CanRxSIZE.SIZE field, taking wrap around effects of the circular buffer into account.

Examples:

- There are 2 CAN messages available for read-out when CanRxSIZE.SIZE=2, CanRxWR.WRITE=2 and CanRxRD.READ=0.
- There are 2 CAN messages available for read-out when CanRxSIZE.SIZE=2, CanRxWR.WRITE =0 and CanRxRD.READ=6.
- There are 2 CAN messages available for read-out when CanRxSIZE.SIZE=2, CanRxWR.WRITE =1 and CanRxRD.READ=7.
- There are 2 CAN messages available for read-out when CanRxSIZE.SIZE=2, CanRxWR.WRITE =5 and CanRxRD.READ=3.

When a CAN message has been successfully received and stored, the write pointer (CanRxWR.WRITE) is automatically incremented, taking wrap around effects of the circular buffer into account. Whenever the read pointer CanRxRD.READ equals (CanRxWR.WRITE+1) modulo (CanRxSIZE.SIZE\*4), there is no space available for receiving another CAN message.

The error behavior of the HurriCANE codec is according to the CAN standard, which applies to the error counter, buss-off condition and error-passive condition.

### 9.6.3 Location

The location of the circular buffer is defined by a base address (CanRxADDR.ADDR), which is an absolute address. The location of a circular buffer is aligned on a 1kbyte address boundary.

### 9.6.4 Reception procedure

When the channel is enabled (CanRxCTRL.ENABLE=1), and there is space available for a message in the circular buffer (as defined by the write and read pointer), as soon as a message is received by the HurriCANE codec, an AMBA AHB store access will be started. The received message will be temporarily stored in a local store-buffer in the CAN controller. Note that the channel should not be enabled until the write and read pointers are configured, to avoid the message reception to start prematurely

After a message has been successfully stored the CAN controller is ready to receive a new message. The write pointer (CanRxWR.WRITE) is automatically incremented, taking wrap around effects of the circular buffer into account.

Interrupts are provided to aid the user during reception, as described in detail later in this section. The main interrupts are the Rx, RxFull and RxIrq which are issued on the successful reception of a message, when the message buffer has been successfully filled and when a predefined number of messages have been received successfully. The RxMiss interrupt is issued whenever a message has been received but does not match a message filtering setting, i.e. neither for the receive channel nor for the SYNC message described hereafter.

The RxSync interrupt is issued when a message matching the SYNC Code Filter Register.SYNC and SYNC Mask Filter Register.MASK registers has been successfully received. Additional interrupts are provided to signal error conditions on the CAN bus and AMBA bus.

### 9.6.5 Straight buffer

It is possible to use the circular buffer as a straight buffer, with a higher granularity than the 1kbyte address boundary limited by the base address (CanRxADDR.ADDR) field.

While the channel is disabled, the write pointer (CanRxWR.WRITE) can be changed to an arbitrary value pointing to the first message to be received, and the read pointer (CanRxRD.READ) can be changed to an arbitrary value.

When the channel is enabled, the reception will start from the write pointer and continue to the read pointer.



### 9.6.6 AMBA AHB error

An AHB error response occurring on the AMBA AHB bus while a CAN message is being stored will result in an RxAHBErr interrupt.

If the CanCONF.ABORT bit is set to 0b, the channel causing the AHB error will skip the received message, not storing it to memory. The write pointer will be incremented.

If the CanCONF.ABORT bit is set to 1b, the channel causing the AHB error will be disabled (CanRxCTRL.ENABLE is cleared automatically to 0b). The write pointer can be used to determine which message caused the AHB error. Note that it could be any of the four word accesses required to write a message that caused the AHB error.

If the CanCONF.ABORT bit is set to 1b, all accesses to the AMBA AHB bus will be disabled after an AMBA AHB error occurs, as indicated by the CanSTAT.AHBErr bit being 1b. The accesses will be disabled until the CanSTAT register is read, and automatically clearing bit CanSTAT.AHBErr.

### 9.6.7 Enable and disable

When an enabled receive channel is disabled (CanRxCTRL.ENABLE=0b), any ongoing CAN message storage on the AHB bus will not be aborted, and no new message storage will be started. Note that only complete messages can be received from the HurriCANE codec. If the message is stored successfully, the write pointer (CanRxWR.WRITE) is automatically incremented. Any associated interrupts will be generated.

The progress of the any ongoing access can be observed via the CanRxCTRL.ONGOING bit. The CanRxCTRL.ONGOING must be 0b before the channel can be re-configured safely (i.e. changing address, size or read/write pointers). It is also possible to wait for the Rx and RxMiss interrupts described hereafter.

The channel can be re-enabled again without the need to reconfigure the address, size and pointers.

No message reception is performed while the channel is not enabled

### 9.6.8 Interrupts

During reception several interrupts can be generated:

- RxMiss:Message filtered away for receive
- RxErrCnt:Error counter incremented for receive
- RxSync:Synchronization message received
- Rx:Successful reception of one message
- RxFull:Successful reception of all messages possible to store in buffer
- RxIrq:Successful reception of a predefined number of messages
- RxAHBErr:AHB access error during reception
- OR:Over-run during reception
- OFF:Bus-off condition
- PASS:Error-passive condition

The Rx, RxFull and RxIrq interrupts are only generated as the result of a successful message reception, after the CanRxWR.WRITE pointer has been incremented.

The OR interrupt is generated when a message is received while a previously received message is still being stored. A full circular buffer will lead to OR interrupts for any subsequently received messages. The last message stored which fills the circular buffer will not generate an OR interrupt. The overrun is reported with the CanSTAT.OR bit, which is cleared when reading the register.

The error behavior of the HurriCANE codec is according to the CAN standard, which applies to the error counter, bus-off condition and error-passive condition.

## 9.7 Global reset and enable

When the CanCTRL.RESET bit is set to 1b, a reset of the core is performed. The reset clears all the register fields to their default values. Any ongoing CAN message transfer request will be aborted, potentially violating the CAN protocol.

When the CanCTRL.ENABLE bit is cleared to 0b, the HurriCANE core is reset and the configuration bits CanCONF.SCALER, CanCONF.PS1, CanCONF.PS2, CanCONF.RSJ and CanCONF.BPR may be modified. When disabled, the CAN controller will be in sleep mode not affecting the CAN bus by only sending recessive bits. Note that the HurriCANE core requires that 10 recessive bits are



received before any reception or transmission can be initiated. This can be caused either by no unit sending on the CAN bus, or by random bits in message transfers.

## 9.8 Interrupt

Three interrupts are implemented by the CAN interface:

Name:	Description:
IRQ	Common output from interrupt handler
TxSYNC	Synchronization message transmitted
RxSYNC	Synchronization message received

## 9.9 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x34.

## 9.10 Registers

The GRHCAN is programmed through registers mapped into APB address space.

Register	Address
Configuration Register	0x80080000
Status Register	0x80080004
Control Register	0x80080008
SYNC Mask Filter Register	0x80080018
SYNC Code Filter Register	0x8008001C
Pending Interrupt Masked Status Register	0x80080100
Pending Interrupt Masked Register	0x80080104
Pending Interrupt Status Register	0x80080108
Pending Interrupt Register	0x8008010C
Interrupt Mask Register	0x80080110
Pending Interrupt Clear Register	0x80080114
Transmit Channel Control Register	0x80080200
Transmit Channel Address Register	0x80080204
Transmit Channel Size Register	0x80080208
Transmit Channel Write Register	0x8008020C
Transmit Channel Read Register	0x80080210
Transmit Channel Interrupt Register	0x80080214
Receive Channel Control Register	0x80080300
Receive Channel Address Register	0x80080304
Receive Channel Size Register	0x80080308
Receive Channel Write Register	0x8008030C
Receive Channel Read Register	0x80080310
Receive Channel Interrupt Register	0x80080314

Register	Address
Receive Channel Mask Register	0x80080318
Receive Channel Code Register	0x8008031C

**Table 9-1. GRHCAN registers**

Any blank register is considered as reserved and has no effect when written to, and returns undefined data when read.

### 9.10.1 Configuration Register[CanCONF]R/W

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SCALER								PS1				PS2			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	RSJ					BPR					Silent	Selection	Enable 1	Enable 0	Abort

**Figure 9-2. Configuration Register**

31-24:	SCALER	Prescaler setting, 8-bit: system clock / (SCALER +1)
23-20:	PS1	Phase Segment 1, 4-bit: (valid range 1 to 15)
19-16:	PS2	Phase Segment 2, 4-bit: (valid range 2 to 8)
14-12:	RSJ	ReSynchronization Jumps, 3-bit: (valid range 1 to 4)
9:8:	BPR	Baud rate, 2-bit:
		00b = system clock / (SCALER +1) / 1
		01b = system clock / (SCALER +1) / 2
		10b = system clock / (SCALER +1) / 4
		11b = system clock / (SCALER +1) / 8
4:	SILENT	Listen only to the CAN bus, send recessive bits.
3:	SELECTION	Selection receiver input and transmitter output:
		Select receive input 0 as active when 0b,
		Select receive input 1 as active when 1b
		Select transmit output 0 as active when 0b,
		Select transmit output 1 as active when 1b
2:	ENABLE1	Set value of output 1 enable
1:	ENABLE0	Set value of output 0 enable
0:	ABORT	Abort transfer on AHB ERROR

All bits are cleared to 0 at reset.

Note that constraints on PS1, PS2 and RSJ are defined as:

- PS1 +1 >= PS2
- PS2 >= RSJ

Note that CAN standard TSEG1 is defined by PS1+1.

Note that CAN standard TSEG2 is defined by PS2.

Note that the SCALER setting defines the CAN time quantum, together with the BPR setting:

$$\text{system clock} / (\text{SCALER}+1) / \text{BPR}$$

where SCALER is in range 0 to 255, and the resulting division factor due to BPR is 1, 2, 4 or 8. Note that the resulting bit rate is:

$$\text{system clock} / (\text{SCALER}+1) / \text{BPR} * (1 + \text{PS1}+1 + \text{PS2})$$

where PS1 is in the range 1 to 15, and PS2 is in the range 2 to 8.

Note that RSJ defines the number of allowed re-synchronization jumps according to the CAN standard, being in the range 1 to 4.

Note that the transmit or receive channel active during the AMBA AHB error is disabled if the ABORT bit is set to 1b. Note that all accesses to the AMBA AHB bus will be disabled after an AMBA AHB error occurs while the ABORT bit is set to 1b. The accesses will be disabled until the CanSTAT register is read.

9.10.2 Status Register[CanSTAT]R

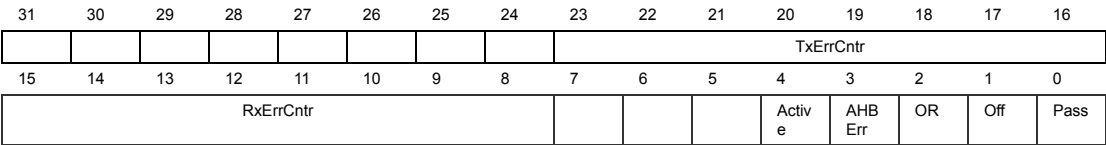


Figure 9-3. Status register

31-24:	Reserved.	No effect when written to. Undefined when read.
23-16:	TxErrCntr	Transmission error counter, 8-bit
15-8:	RxErrCntr	Reception error counter, 8-bit
4:	ACTIVE	Transmission ongoing
3:	AHBErr	AMBA AHB master interface blocked due to previous AHB error
2:	OR	Overflow during reception
1:	OFF	Bus-off condition
0:	PASS	Error-passive condition

All bits are cleared to 0 at reset.

The OR bit is set if a message with a matching ID is received and cannot be stored via the AMBA AHB bus, this can be caused by bandwidth limitations or when the circular buffer for reception is already full.

The OR and AHBErr status bits are cleared when the register has been read.

Note that TxErrCntr and RxErrCntr are defined according to CAN protocol.

Note that the AHBErr bit is only set to 1b if an AMBA AHB error occurs while the CanCONF.ABORT bit is set to 1b.

9.10.3 Control Register[CanCTRL]R/W

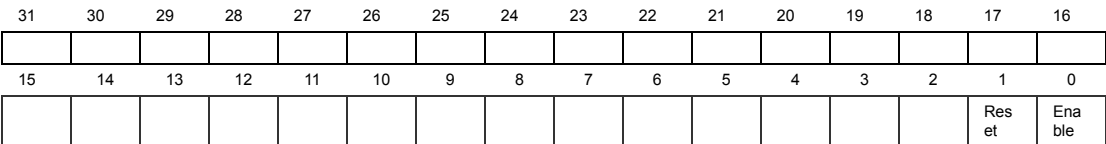


Figure 9-4. Control Register

31-2:	Reserved.	No effect when written to. Undefined when read.
1:	RESET	Reset complete core when 1

0:        ENABLE            Enable HurriCANE controller, when 1. Reset HurriCANE controller, when 0  
All bits are cleared to 0 at reset.

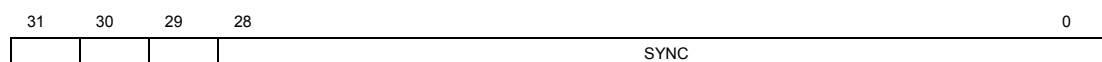
Note that RESET is read back as 0b.

Note that ENABLE should be cleared to 0b while other settings are modified, ensuring that the HurriCANE core is properly synchronized.

Note that when ENABLE is cleared to 0b, the CAN interface is in sleep mode, only outputting recessive bits.

Note that the HurriCANE core requires that 10 recessive bits be received before receive and transmit operations can begin.

#### 9.10.4 SYNC Code Filter Register[CanCODE]R/W



**Figure 9-5. SYNC Code Filter Register**

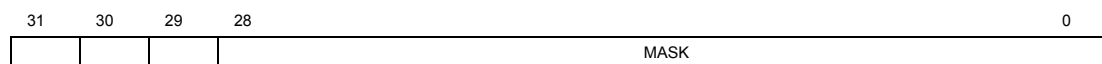
31-29:    Reserved.            No effect when written to. Undefined when read.

28-0:    SYNC                Message Identifier

All bits are cleared to 0 at reset.

Note that Base ID is bits 28 to 18 and Extended ID is bits 17 to 0.

#### 9.10.5 SYNC Mask Filter Register[CanMASK]R/W



**Figure 9-6. SYNC Mask Filter Register**

31-29:    Reserved.            No effect when written to. Undefined when read.

28-0:    MASK                Message Identifier

All bits are set to 1 at reset.

Note that Base ID is bits 28 to 18 and Extended ID is bits 17 to 0.

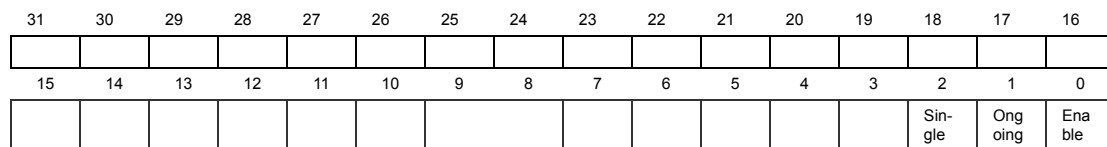
A RxSYNC message ID is matched when

$$((\text{Received-ID XOR CanCODE.SYNC}) \text{ AND CanMASK.MASK}) = 0$$

A TxSYNC message ID is matched when

$$((\text{Transmitted-ID XOR CanCODE.SYNC}) \text{ AND CanMASK.MASK}) = 0$$

### 9.10.6 Transmit Channel Control Register[CanTxCTRL]R/W



**Figure 9-7. Transmit Channel Control Register**

- 31-3: Reserved. No effect when written to. Undefined when read.
  - 2: SINGLE Single shot mode
  - 1: ONGOING Transmission ongoing
  - 0: ENABLE Enable channel
- All bits are cleared to 0 at reset.

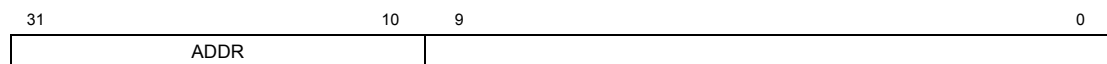
Note that if the SINGLE bit is 1b, the channel is disabled (i.e. the ENABLE bit is cleared to 0b) if the arbitration on the CAN bus is lost.

Note that in the case an AHB bus error occurs during an access while fetching transmit data, and the CanCONF.ABORT bit is 1b, then the ENABLE bit will be reset automatically.

At the time the ENABLE is cleared to 0b, any ongoing message transmission is not aborted, unless the CAN arbitration is lost or communication has failed.

Note that the ONGOING bit being 1b indicates that message transmission is ongoing and that configuration of the channel is not safe.

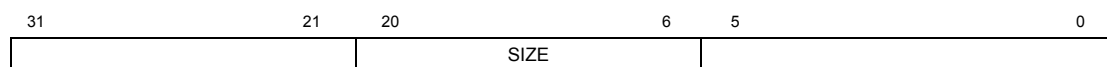
### 9.10.7 Transmit Channel Address Register[CanTxADDR]R/W



**Figure 9-8. Transmit Channel Address Register**

- 31-10: ADDR Base address for circular buffer
  - 9-0: Reserved. No effect when written to. Undefined when read.
- All bits are cleared to 0 at reset.

### 9.10.8 Transmit Channel Size Register[CanTxSIZE]R/W



**Figure 9-9. Transmit Channel Size Register**

- 31-21: Reserved. No effect when written to. Undefined when read.
  - 20-6: SIZE The size of the circular buffer is SIZE\*4 messages  
Valid SIZE values are between 0 and 16384.
  - 5-0: Reserved. No effect when written to. Undefined when read.
- All bits are cleared to 0 at reset.

Note that each message occupies four 32-bit words.

Note that the resulting behavior of invalid SIZE values is undefined.

Note that only (SIZE\*4)-1 messages can be stored simultaneously in the buffer. This is to simplify wrap-around condition checking.

9.10.9 Transmit Channel Write Register[CanTxWR]R/W

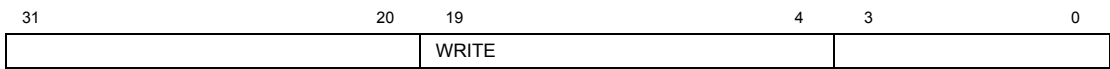


Figure 9-10. Transmit Channel Write Register

- 31-20: Reserved. No effect when written to. Undefined when read.
  - 19-4: WRITE Pointer to last written message +1
  - 3-0: Reserved. No effect when written to. Undefined when read.
- All bits are cleared to 0 at reset.

The WRITE field is written to in order to initiate a transfer, indicating the position +1 of the last message to transmit.

Note that it is not possible to fill the buffer. There is always one message position in buffer unused. Software is responsible for not over-writing the buffer on wrap around (i.e. setting WRITE=READ).

The field is implemented as relative to the buffer base address (scaled with the SIZE field).

9.10.10 Transmit Channel Read Register[CanTxRD]R/W

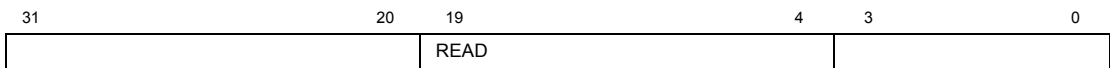


Figure 9-11. Transmit Channel Read Register

- 31-20: Reserved. No effect when written to. Undefined when read.
  - 19-4: READ Pointer to last read message +1
  - 3-0: Reserved. No effect when written to. Undefined when read.
- All bits are cleared to 0 at reset.

The READ field is written to automatically when a transfer has been completed successfully, indicating the position +1 of the last message transmitted.

Note that the READ field can be use to read out the progress of a transfer.

Note that the READ field can be written to in order to set up the starting point of a transfer. This should only be done while the transmit channel is not enabled.

Note that the READ field can be automatically incremented even if the transmit channel has been disabled, since the last requested transfer is not aborted until CAN bus arbitration is lost.

When the Transmit Channel Read Pointer catches up with the Transmit Channel Write Register, an interrupt is generated (TxEmpty). Note that this indicates that all messages in the buffer have been transmitted.

The field is implemented as relative to the buffer base address (scaled with the SIZE field).

9.10.11 Transmit Channel Interrupt Register[CanTxIRQ]R/W

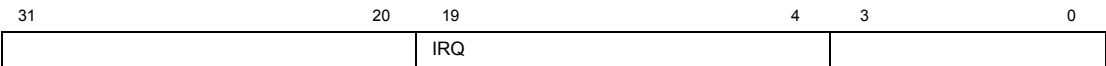


Figure 9-12. Transmit Channel Interrupt Register

- 31-20: Reserved. No effect when written to. Undefined when read.
  - 19-4: IRQ Interrupt is generated when CanTxRD.READ=IRQ, as a consequence of a message transmission
  - 3-0: Reserved. No effect when written to. Undefined when read.
- All bits are cleared to 0 at reset.

Note that this indicates that a programmed number of messages have been transmitted.  
The field is implemented as relative to the buffer base address (scaled with the SIZE field).

9.10.12 Receive Channel Control Register[CanRxCTRL]R/W

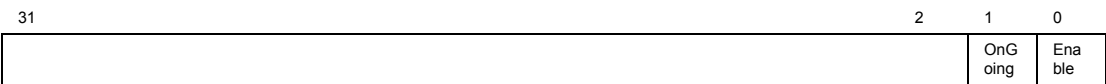


Figure 9-13. Receive Channel Control Register

- 31-2: Reserved. No effect when written to. Undefined when read.
  - 1: ONGOING Reception ongoing (read-only)
  - 0: ENABLE Enable channel
- All bits are cleared to 0 at reset.

Note that in the case an AHB bus error occurs during an access while fetching transmit data, and the CanCONF.ABORT bit is 1b, then the ENALBE bit will be reset automatically.  
At the time the ENABLE is cleared to 0b, any ongoing message reception is not aborted  
Note that the ONGOING bit being 1b indicates that message reception is ongoing and that configuration of the channel is not safe.

9.10.13 Receive Channel Address Register[CanRxADDR]R/W

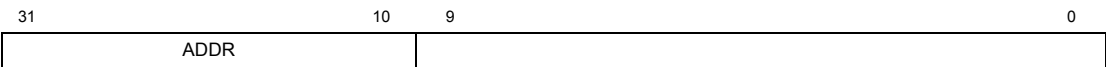
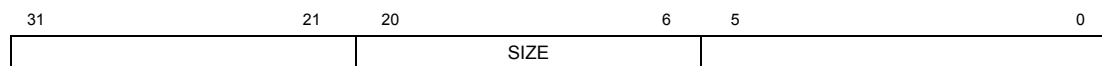


Figure 9-14. Receive Channel Address Register

- 31-10: ADDR Base address for circular buffer
  - 9-0: Reserved. No effect when written to. Undefined when read.
- All bits are cleared to 0 at reset.

#### 9.10.14 Receive Channel Size Register[CanRxSIZE]R/W



**Figure 9-15. Receive Channel Size Register**

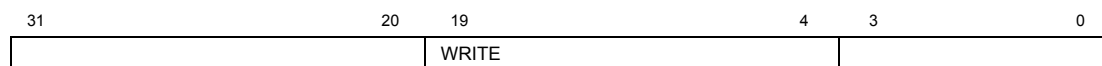
- |        |           |  |
|--------|-----------|--|
| 31-21: | Reserved. | No effect when written to. Undefined when read.  |
| 20-6:  | SIZE      | The size of the circular buffer is SIZE*4 messages<br>Valid SIZE values are between 0 and 16384. |
| 5-0:   | Reserved. | No effect when written to. Undefined when read.  |
- All bits are cleared to 0 at reset.

Note that each message occupies four 32-bit words.

Note that the resulting behavior of invalid SIZE values is undefined.

Note that only (SIZE\*4)-1 messages can be stored simultaneously in the buffer. This is to simplify wrap-around condition checking.

#### 9.10.15 Receive Channel Write Register[CanRxWR]R/W



**Figure 9-16. Receive Channel Write Register**

- |        |           |   |
|--------|-----------|---|
| 31-20: | Reserved. | No effect when written to. Undefined when read. |
| 19-4:  | WRITE     | Pointer to last written message +1              |
| 3-0:   | Reserved. | No effect when written to. Undefined when read. |
- All bits are cleared to 0 at reset.

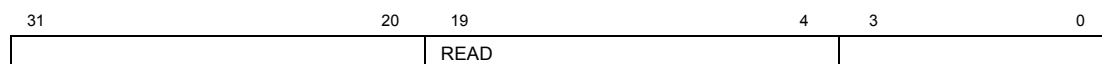
The field is implemented as relative to the buffer base address (scaled with the SIZE field).

The WRITE field is written to automatically when a transfer has been completed successfully, indicating the position +1 of the last message received.

Note that the WRITE field can be use to read out the progress of a transfer.

Note that the WRITE field can be written to in order to set up the starting point of a transfer. This should only be done while the receive channel is not enabled.

#### 9.10.16 Receive Channel Read Register[CanRxRD]R/W



**Figure 9-17. Receive Channel Read Register**

- |        |           |   |
|--------|-----------|---|
| 31-20: | Reserved. | No effect when written to. Undefined when read. |
|--------|-----------|---|



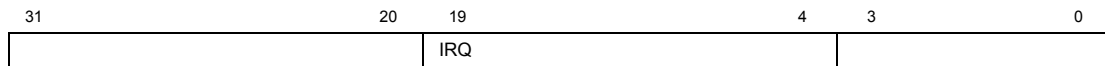
19-4: READ                      Pointer to last read message +1  
 3-0: Reserved.                No effect when written to. Undefined when read.  
 All bits are cleared to 0 at reset.

The field is implemented as relative to the buffer base address (scaled with the SIZE field).

The READ field is written to in order to release the receive buffer, indicating the position +1 of the last message that has been read out.

Note that it is not possible to fill the buffer. There is always one message position in buffer unused. Software is responsible for not over-reading the buffer on wrap around (i.e. setting WRITE=READ).

#### 9.10.17 Receive Channel Interrupt Register[CanRxIRQ]R/W



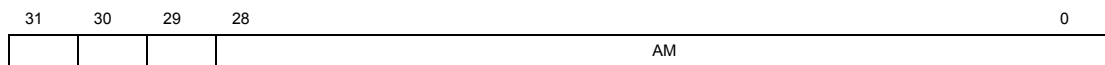
**Figure 9-18. Receive Channel Interrupt Register**

31-20: Reserved.                No effect when written to. Undefined when read.  
 19-4: IRQ                        Interrupt is generated when CanRxWR.WRITE=IRQ, as a consequence of a message reception  
 3-0: Reserved.                No effect when written to. Undefined when read.  
 All bits are cleared to 0 at reset.

Note that this indicates that a programmed number of messages have been received.

The field is implemented as relative to the buffer base address (scaled with the SIZE field).

#### 9.10.18 Receive Channel Mask Register[CanRxMASK]R/W

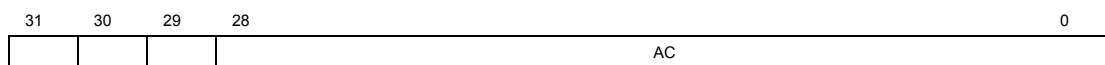


**Figure 9-19. Receive Channel Mask Register**

31-29: Reserved.                No effect when written to. Undefined when read.  
 28-0: AM                        Acceptance Mask, bits set to 1b are taken into account in the comparison between the received message ID and the CanRxCODE.AC field  
 All bits are set to 1 at reset.

Note that Base ID is bits 28 to 18 and Extended ID is bits 17 to 0.

#### 9.10.19 Receive Channel Code Register[CanRxCODE]R/W



**Figure 9-20. Receive Channel Code Register**

31-29: Reserved.                No effect when written to. Undefined when read.

28-0: AC Acceptance Code, used in comparison with the received message  
All bits are cleared to 0 at reset.

Note that Base ID is bits 28 to 18 and Extended ID is bits 17 to 0.

A message ID is matched when:

$$((\text{Received-ID XOR CanRxCODE.AC}) \text{ AND CanRxMASS.AM}) = 0$$

### 9.10.20 Interrupt registers

The interrupt registers give complete freedom to the software, by providing means to mask interrupts, clear interrupts, force interrupts and read interrupt status.

When an interrupt occurs the corresponding bit in the Pending Interrupt Register is set. The normal sequence to initialize and handle a module interrupt is:

- Set up the software interrupt-handler to accept an interrupt from the module.
- Read the Pending Interrupt Register to clear any spurious interrupts.
- Initialise the Interrupt Mask Register, unmasking each bit that should generate the module interrupt.
- When an interrupt occurs, read the Pending Interrupt Status Register in the software interrupt-handler to determine the causes of the interrupt.
- Handle the interrupt, taking into account all causes of the interrupt.
- Clear the handled interrupt using Pending Interrupt Clear Register.

**Masking interrupts:** After reset, all interrupt bits are masked, since the Interrupt Mask Register is zero. To enable generation of a module interrupt for an interrupt bit, set the corresponding bit in the Interrupt Mask Register.

**Clearing interrupts:** All bits of the Pending Interrupt Register are cleared when it is read or when the Pending Interrupt Masked Register is read. Reading the Pending Interrupt Masked Register yields the contents of the Pending Interrupt Register masked with the contents of the Interrupt Mask Register. Selected bits can be cleared by writing ones to the bits that shall be cleared to the Pending Interrupt Clear Register.

**Forcing interrupts:** When the Pending Interrupt Register is written, the resulting value is the original contents of the register logically OR-ed with the write data. This means that writing the register can force (set) an interrupt bit, but never clear it.

**Reading interrupt status:** Reading the Pending Interrupt Status Register yields the same data as a read of the Pending Interrupt Register, but without clearing the contents.

**Reading interrupt status of unmasked bits:** Reading the Pending Interrupt Masked Status Register yields the contents of the Pending Interrupt Register masked with the contents of the Interrupt Mask Register, but without clearing the contents.

The interrupt registers comprise the following:

- Pending Interrupt Masked Status Register [CanPIMSR] R
- Pending Interrupt Masked Register [CanPIMR] R
- Pending Interrupt Status Register [CanPISR] R
- Pending Interrupt Register [CanPIR] R/W
- Interrupt Mask Register [CanIMR] R/W
- Pending Interrupt Clear Register [CanPICR] W

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

Rx Miss	Tx Err Cntr	Rx Err Cntr	Tx Sync	Rx Sync	Tx	Rx	Tx Empty	Rx Full	Tx IRQ	Rx IRQ	Tx AHB Err	Rx AHB Err	OR	Off	Pass

**Figure 9-21. Interrupt registers**

31-17:	Reserved.	No effect when written to. Undefined when read.
16:	TxLoss	Message arbitration lost during transmission (could be caused by communications error, as indicated by other interrupts as well)
15:	RxMiss	Message filtered away during reception
14:	TxErrCntr	Transmission error counter incremented
13:	RxErrCntr	Reception error counter incremented
12:	TxSync	Synchronization message transmitted
11:	RxSync	Synchronization message received
10:	Tx	Successful transmission of message
9:	Rx	Successful reception of message
8:	TxEmpty	Successful transmission of all messages in buffer
7:	RxFull	Successful reception of all messages possible to store in buffer
6:	TxIRQ	Successful transmission of a predefined number of messages
5:	RxIRQ	Successful reception of a predefined number of messages
4:	TxAHBErr	AHB error during transmission
3:	RxAHBErr	AHB error during reception
2:	OR	Over-run during reception
1:	OFF	Bus-off condition
0:	PASS	Error-passive condition

All bits in all interrupt registers are reset to 0b after reset.

Note that the TxAHBErr interrupt is generated in such way that the corresponding read and write pointers are valid for failure analysis. The interrupt generation is independent of the CanCONF.ABORT field setting.

Note that the RxAHBErr interrupt is generated in such way that the corresponding read and write pointers are valid for failure analysis. The interrupt generation is independent of the CanCONF.ABORT field setting.

## 9.11 Memory mapping

The CAN message is represented in memory as shown in next table.

AHB addr															
0x0	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17 16
	IDE	RTR	-	bID										eID	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1 0
	eID														
0x4	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17 16
	DLC								TxErrCntr						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1 0
	RxErrCntr												Ahb Err	OR	Off Pas s
0x8	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17 16
	Byte 0 (first transmitted)								Byte 1						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1 0
	Byte 2								Byte 3						
0xC	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17 16
	Byte 4								Byte 5						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1 0
	Byte 6								Byte 7 (last transmitted)						

**Figure 9-22. CAN message representation in memory.**

Values: Levels according to CAN standard:

1b is recessive,

0b is dominant

Legend: Naming and number in according to CAN standard

IDE Identifier Extension:

1b for Extended Format,

0b for Standard Format

RTR Remote Transmission Request:

1b for Remote Frame,

0b for Data Frame

bID Base Identifier

eID Extended Identifier

DLC Data Length Code, according to CAN standard:

0000b 0 bytes

0001b 1 byte

0010b 2 bytes

0011b 3 bytes

0100b 4 bytes

0101b 5 bytes

0110b 6 bytes

0111b 7 bytes

	1000b    8 bytes
	OTHERS illegal
TxErrCntr	Transmission Error Counter
RxErrCntr	Reception Error Counter
AHBErr	AHB interface blocked due to AHB Error when 1b
OR	Reception Over run when 1b
OFF	Bus Off mode when 1b
PASS	Error Passive mode when 1b
Byte 00 to 07 Transmit/Receive data, Byte 00 first Byte 07 last	

## 10. SPACEWIRE LINK INTERFACE

This chapter gives an overview of the functions of the SpaceWire (SPW2) Module. It is written as a descriptive text used to increase the understanding of the functions.

### 10.1 System overview

The SpaceWire (SPW2) Module is intended to fit into systems where there is a need for communication via SpaceWire links.

The figure below shows how the SpaceWire Module would fit into a typical application. Note that the routers are optional and direct SpaceWire links may be used instead.

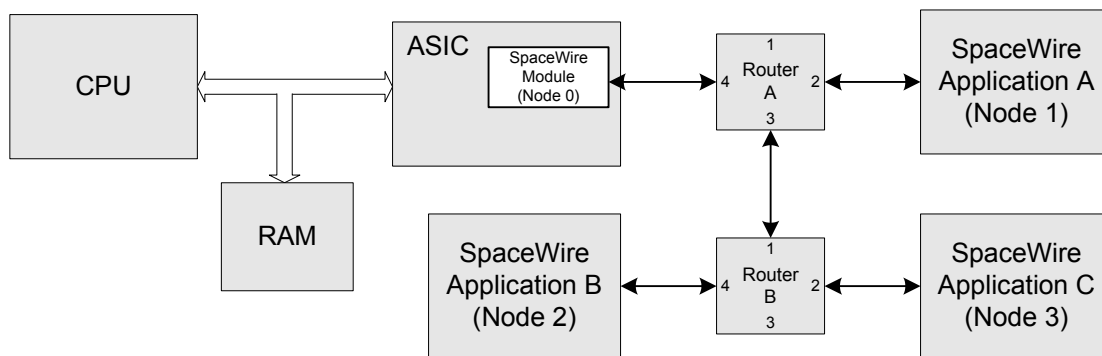


Figure 10-1. System overview

### 10.2 Functions

Each SPW2 module handles a bi-directional non-redundant SpaceWire link. In reception the module optionally supports VCTP (SPW Virtual Channel Transfer Protocol), basic RMAP (Remote Memory Access Protocol) commands and redirects non-supported commands and protocols to the software. The module supports multiple independent transmit send lists with dual pointers for header and data.

### 10.3 Interfaces

The SpaceWire (SPW2) Module interfaces one SpaceWire link. The SpaceWire link encodes data using one signal pair of data and strobe in each direction.

The SpaceWire link provides full duplex operation, which means the SpaceWire Module may transmit and receive data simultaneously.

The SpaceWire link handled by the SpaceWire Module is non-redundant. Two SpaceWire (SPW2) Modules could be used, each with its own SpaceWire link, if redundancy is required.

### 10.4 Module overview

This section provides an introduction to the functions implemented by the SpaceWire (SPW2) Module.

### 10.4.1 SpaceWire Link

SpaceWire is a bi-directional, serial, point-to-point data link. The SpaceWire link is used for transferring data from one node to another node. The nodes may be part of a network or work stand-alone. In a network routers are needed to direct transmitted packets to the correct destinations. In this case a transmitted packet contains a header that indicates to the router(s) which destination the packet is targeted for.

The SpaceWire Module implements a single SpaceWire link. This link can be used for either transmitting data in both directions on one or more virtual channels, or to implement a remote memory access protocol. Both transmission and reception are supported by direct memory access to the on-chip bus on which the SpaceWire Module is implemented. The SpaceWire Module only locally implements the buffering resources that are strictly necessary for its functionality, all other data storage is performed via the on-chip bus.

### 10.4.2 Transmit protocols

The SpaceWire (SPW2) module supports the transmission of any type of SpaceWire packets, not limited to any type of higher protocols. It implements virtual channels that allow transfer from one or many dedicated areas in memory. The virtual channels allow multiple communication channels over a single SpaceWire link.

The selection of the virtual channel from which data are to be transmitted on the SpaceWire link is performed by round-robin arbitration. Each virtual transmit channel has a separate send list stored in memory. The send list entries define what data is to be sent from memory. The send list entry structure is shown in 10.5.9.6. Note that automatically generated RMAP responses have higher priority than packets defined in the virtual channel send lists, thus bypassing the round-robin priority scheme.

Each send list entry specifies the position and size of the header and the position and size of the data to be transmitted. The header and data to be transmitted, as well as the send lists, can be located anywhere in memory. The SpaceWire Module performs direct memory accesses to read data to be transmitted from a memory area.

### 10.4.3 Receive protocols

The SpaceWire (SPW2) module supports the reception of several protocols as explained hereafter.

The SpaceWire Link operates on packets. A packet includes an optional destination path address, a destination logical address, a protocol identifier, a cargo and an end of packet marker, as defined in 10.5.9.2. The basic packet structure is used for implementing the more advanced structures discussed hereafter.

The SpaceWire Virtual Channel Transfer Protocol (VCTP), as defined in 10.5.9.5, is used to implement virtual channels on a single SpaceWire link. The Remote Memory Access Protocol (RMAP), as defined in [RMAP] and in 10.5.9.4, is used to implement remote memory access to resources in the node via the SpaceWire link. Both the VCTP and the RMAP protocols can coexist on the same SpaceWire link simultaneously.

Both protocols use the SpaceWire Transfer Protocol Packet Structure, in which the Protocol Identifier is used for making the distinction between different protocols, including VCTP and RMAP. Additionally, other protocols can also be supported, but require processing support from the node in which the SPW2 module is embedded.

#### 10.4.3.1 SpaceWire Virtual Channel Transfer Protocol (VCTP)

The SpaceWire Virtual Channel Transfer Protocol (VCTP) protocol allows transfer to one or many dedicated areas in memory. The virtual channels allow multiple communication links over a single SpaceWire link.

The selection of the virtual channel to which the received data from the SpaceWire link belong is made using the Virtual Channel Identifier carried in the packet. Each virtual receiver channel has a separately programmable memory area to which received data are

stored. This area can be located anywhere in memory. The SpaceWire Module performs direct memory accesses to write received data to a memory area.

#### **10.4.3.2 Remote Memory Access Protocol (RMAP)**

The Remote Memory Access Protocol (RMAP) protocol is used for writing to and reading from memory, registers, FIFO memory, mailboxes, etc., in a destination node on a SpaceWire network. Input/output registers, control/status registers are assumed to be memory mapped and accessed as memory.

All read and write operations defined in the RMAP protocol are posted operations i.e. the source does not wait for an acknowledgement or reply to be received. This means that many read and write accesses can be outstanding at any time. It also means that there is no timeout mechanism implemented in RMAP for missing acknowledgements or replies. If an acknowledgement or reply timeout mechanism is required it must be implemented in the source user application.

The RMAP protocol is realised in three ways in the SpaceWire Module, with hardware support, with software support or with both.

The hardware implements a subset of the RMAP commands. A detailed list of command constraints is provided in 10.6.4.1 and 10.6.4.2. Automatic RMAP responses are generated to RMAP commands that are implemented in hardware. A received packet is not stored in memory for RMAP commands that are executed in hardware.

Commands not supported by hardware can optionally be relayed to software for further processing. If no hardware support is required, all commands can be relayed to software for processing. Commands are relayed to software using a dedicated virtual receive channel, similar to what has been described for the VCTP protocol. Responses can be generated by software using a virtual transmit channel as described in 10.4.2.

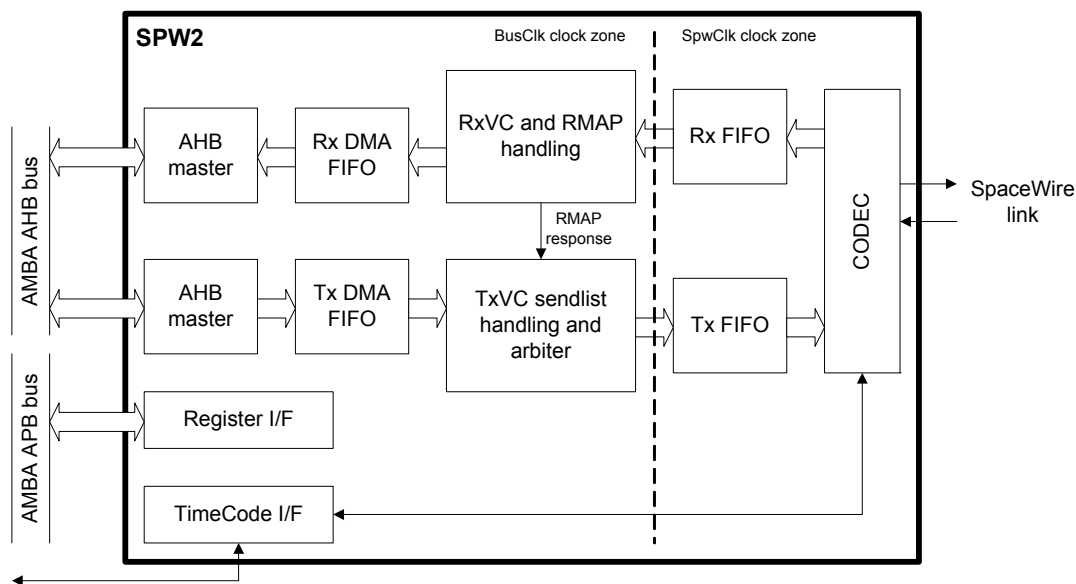
Any received RMAP responses are also relayed to software in the same way as RMAP commands, using the same virtual receive channel.

#### **10.4.3.3 Additional protocols**

Other protocols than the VCTP and RMAP protocols can be implemented with software support. The commands are relayed to the software using a dedicated virtual receive channel. This relaying to software can be controlled automatically using the Transfer ID field of the packet header, but it can also be enforced by configuring the SPW2 to route all incoming packets to the software.



## 10.4.4 Block Diagram



**Figure 10-2. SpaceWire functional block diagram**

The SpaceWire (SPW2) Module is used for transmitting and receiving data over a SpaceWire link. It provides support for transmitting any type of protocol or data structure using SpaceWire packets.

It provides hardware support for receiving two types of SpaceWire Transfer Protocols, and can relay packets of other protocols to software. The SpaceWire Virtual Channel Transfer Protocol (VCTP) implements multiple virtual channels on a single SpaceWire link. The Remote Memory Access Protocol (RMAP) implements remote memory access to resources in the node via the SpaceWire link.

The SpaceWire (SPW2) Module implements the higher-level transfer protocols, whereas the SpaceWire link itself is implemented by the encapsulated University of Dundee SpaceWire CODEC.

Data to be sent are read by the SpaceWire Module from memory via direct memory access. An AMBA AHB master dedicated to transmission performs this over the internal AMBA AHB bus. Data are then temporarily stored in a Tx FIFO when forwarded to the SpaceWire CODEC for transmission over the link. Multiple Virtual Transmit Channels (TxVC) can be used, each with its private send list stored in memory from which data are read. All TxVC share the same link. Round-robin arbitration between the TxVC is performed to ensure fair bandwidth allocation between the channels. The arbitration is performed for each packet sent.

Data received over the link by the SpaceWire CODEC are temporarily stored in an Rx FIFO. Data are then stored to memory by the SpaceWire Module via direct memory access. An AHB master dedicated to reception performs this over the internal AMBA AHB bus. Multiple Virtual Receive Channels (RxVC) can be used, each with its private memory area to which data are written. All RxVC share the same link.

The SpaceWire Module implements hardware support for the RMAP. RMAP is used for remotely accessing resources on the local AMBA bus. The RMAP implementation can receive commands and generate responses, utilizing the aforementioned Rx FIFO and the Tx FIFO, and the two AMBA AHB masters. RMAP has priority over any TxVC.

The SpaceWire Module is configured, controlled and monitored via an AMBA APB bus slave interface, accessing all module internal registers.

The SpaceWire CODEC uses a separate clock, the SpwClk, as its clock while the rest of the SpaceWire Module uses the BusClk. The SpaceWire CODEC requires a clock frequency that is a multiple of 10 MHz to produce a data rate of 10 Mbit/s required for reliable start-up procedures. The SpaceWire CODEC clock is also used for sending data at high transfer rates.

## 10.5 Definitions

This section and the following subsections define the typographic and naming conventions used throughout this document.

### 10.5.1 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.

### 10.5.2 Names

The following conventions are used for all names (for signals and registers some extra conventions are defined below):

- A name may never start with a digit, e.g. 1553 could instead be M1553.
- A dollar sign (\$) in a name is used as a wildcard representing a number. (If the dollar sign (\$) is used in a context it must then be defined somewhere else in the document)
- An asterisk (\*) in a name is used as a wildcard representing one or more characters.

### 10.5.3 Radix

The following conventions is used for writing numbers:

- Binary numbers are indicated by the subscript "2", e.g. 1<sub>2</sub>, 1011\_1010\_1011\_1110<sub>2</sub>, 010010<sub>2</sub> etc.
- Decimal numbers are indicated by the subscript "10", e.g. 67,8723<sub>10</sub>, 47860<sub>10</sub>.
- Hexadecimal numbers are indicated by the subscript "16", e.g. E<sub>16</sub>, BABE<sub>16</sub>.
- Unless the Radix is explicitly declared as above the number should be considered to be decimal number.

### 10.5.4 Signal Names

The following conventions are used for signal names:

- Signal names are written in italics, e.g. *SignalName*.
- Active low signals have a capital N appended to their name, e.g. *SignalNameN*.
- Bus indices are indicated with brackets, e.g. *SignalName*[12:3].
- Signals maybe grouped into subsignals, e.g. *SignalName.SubSignal*.
- Signals with two functions are named with the name and then the first functionality followed by the second function, e.g. *SignalNameFunction1Function2N*. The second function is the valid when the signal is deasserted (thus the suffix N in the name).

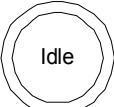



### 10.5.5 Externally Accessible Register Names

The following convention is used for externally accessible registers.

- Register names are underlined, e.g. RegisterName.
- Fields of a register are indicated by the name of the register and the field, separated by a period and underlined, e.g. RegisterName.Field.

## 10.5.6 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

## 10.5.7 Terminology

### 10.5.7.1General

ASIC Module	An ASIC internal module
Issue an Interrupt	The corresponding bit has been set in the pending interrupt register in the ASIC/FPGA module
Reset Assertion	An internal reset activation as seen by ASIC/FPGA modules
GoStop Assertion	An internal go-stop activation as seen by ASIC/FPGA modules

### 10.5.7.2Basic Data Types

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

### 10.5.7.3Registers

Register Read	A read access to a ASIC external accessible register
Register Write	A write access to a ASIC external accessible register
Set	Indicates that the bit in the register is 1
Clear	Indicates that the field or bit in the register is 0
Reset	Indicates that the field or bit in the register is set to its default Value (indicated in the Register definition chapter)

#### 10.5.7.4 Register Access

Read (R)	Register read where the register is read as is, the read has no effect on the register
Read and Clear (RC)	Register read where the register is read as is and the register content is cleared
Read Masked (RM)	Register read where the register is read and masked with the content of the corresponding mask register
Read and Clear Masked (RCM)	Register read where the register is read and masked with the content of the corresponding mask register and unmasked register bits are cleared
Read Masked and Clear (RMC)	Register read where the register is read and masked with the content of the corresponding mask register and the register content is cleared
Write (W)	Register write where the register content is updated according to the provided parameter
Write and Trigger (WT)	Register write where the register content is updated according to the provided parameter and hardware processing is triggered (activated)
Set (S)	Register write where the bits that are set in the provided parameter is also set in the register, bits that are cleared in the provided parameter is unaffected in the register
Clear (C)	Register write where the bits that are set in the provided parameter is cleared in the register, bits that are cleared in the provided parameter is unaffected in the register
Arm (A)	Register write where the bits that are set in the provided parameter is set ready to receive a new value, once executed
Execute (E)	Register write where the register bits that have previously been armed are updated according to the provided parameter
Trigger (T)	Register write where hardware processing is triggered (activated) (no parameter or fixed parameter)

#### 10.5.7.5 Signals

Assert	To put a signal into its active state. A signal is <i>asserted</i> when in its active state
Deassert	To put a signal into its inactive state. A signal is <i>deasserted</i> when in its inactive state

#### 10.5.7.6 Direct memory access

DMA Channel	A unit for accessing memory
DMA Error	Signal to DMA Channel for failing memory access
DMA Read	A transfer of data to a DMA channel from a DMA controller, e.g. read data from memory
DMA Write	A transfer of data from a DMA channel to a DMA controller, e.g. write data to memory.

#### 10.5.7.7SPW2 Specific

Active RxVC[\$]/ TxVC[\$] Inactive RxVC[\$]/ TxVC[\$]	Virtual receive or transmit channel, RxVC[\$] or TxVC[\$] is currently receiving or transmitting data.
Early EOP Late EOP Early EEP Late EEP	No data is currently being received or transmitted on virtual receive or transmit channel, RxVC[\$] or TxVC[\$].  EOP has been received after less data than expected from the RMAP command header EOP has been received after more data than expected from the RMAP command header. EEP has been received after less data than expected from the RMAP command header EEP has been received after more data than expected from the RMAP command header.

## 10.5.8 Abbreviations

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture <sup>TM</sup>
APB	Advanced Peripheral Bus
ASIC	Application Specific Integrated Circuit
BPS	Bit Per Second
CMOS	Complementary Metal Oxide Semiconductor
CRC	Cyclic Redundancy Code
DMA	Direct Memory Access
EEP	Error End of Packet
EOP	End Of Packet
EOM	End Of Message
EOB	End Of Block
ESA	European Space Agency
FFPR	First Failing Packet Register
FPGA	Field Programmable Gate Array
HW	Hardware
Id	Identifier
IO	Input/Output
IP	Intellectual Property
LS	Least Significant
LSB	Least Significant Bit
LSW	Least Significant Word
LVDS	Low Voltage Differential Signalling
MS	Most Significant
MSB	Most Significant Bit
MSW	Most Significant Word
MTBF	Mean Time Between Failures
NA	Not Applicable
PROM	Programmable Read Only Memory
RAM	Random Access Memory
SPW	SpaceWire
RMAP	Remote Memory Access Protocol
RxVC	Virtual Receive Channel
SRAM	Static Random Access Memory
SW	Software
TBC	To Be Confirmed
TBD	To Be Determined
TxVC	Virtual Transmit Channel
VC	Virtual Channel
VCID	Virtual Channel Identifier
VCTP	SPW Virtual Channel Transfer Protocol
VCTPID	SPW Virtual Channel Transfer Protocol Identifier
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

## 10.5.9 Data Structures

### 10.5.9.1 SPW2 Data Structure Definitions

**Destination Path Address** is a SpaceWire path address, which defines the route to a destination node by specifying, for each router encountered on the way to the destination, the output port that a packet is to be forwarded through. A path address comprises one byte for each router on the path to the destination. Once a path address byte has been used to specify an output port of a router it is deleted to expose the next path address byte for the next router. All path address bytes will have been deleted by the time the packet reaches the destination.

**Destination Logical Address (DLA)** byte is the logical address of the destination. This may be used to route the packet to the destination or, if path addressing is being used, to simply confirm that the final destination is the correct one i.e. that the logical address of the destination matches the logical address in the packet. If the logical address of the destination is unknown then the default logical address of 254 may be used. The destination may choose to accept or reject packets with a logical address of 254. The SPW2 requires a perfect DLA match for accepting a packet, but its DLA reset value is 254. Note also that for received RMAP responses and when configured to forced unknown protocol interpretation the SPW2 does not check the DLA.

**Protocol ID** byte identifies the particular protocol being used for communication. For the Remote Memory Access protocol [RMAP] the protocol identifier has the value 1 ( $01_{16}$ ). For Virtual Channel Transfer Protocol (VCTP) the default protocol identifier is 240.

**Command & Type** byte determines the type of the packet i.e. a command, a response or an acknowledgement. This byte also includes two bits that determine the length in words of the Source Path Address field. For example, if these bits are set to the value two then the Source Path Address will consist of eight bytes. If they are set to zero then there is no Source Path Address field at all.

**Destination Key** provides a one byte key which must be matched by the destination application in order for the RMAP command to be accepted.

**Source Path Address** bytes provide the destination path address for the response or acknowledgement of a command. The source path address is not needed if logical addressing is being used. The Source Path Address, if used, is normally set to the address from the destination node back to the source node which sent the command. Leading all-zero bytes of the return address are ignored. If a packet is to be sent to address zero then this is done by setting the entire Source Path Address to zero. This will result in a single zero path address byte being sent in front of the logical address.

**Source Logical Address** byte is the destination logical address to be used in a response or acknowledgement to a command. The Source Address is normally set to the logical address of the source node that is sending the command. The Source Address byte may be set to 254 ( $0FE_{16}$ ), which is the default logical address, if the command source node does not have a logical address.

**Transaction Identifier** bytes are used to identify command, response, and acknowledge transactions that make up a particular read or write operation. The source of the command gives the command a unique transaction identifier. This transaction identifier is returned in the response or acknowledgement to the command. This allows the command source to send many commands without having to wait for a response to each command before sending the next command. When a response or acknowledge comes in it can be quickly associated with the command that caused it by the transaction identifier.

**Extended Address** byte is used to extend the 32-bit memory address to 40-bits allowing a 1 Terabyte address space to be accessed directly in each node. Note that the SPW2 RMAP protocol handler supports only 4 bits of extended addressing, i.e. a 64 GiB address space; and thus bits 7-4 of the Extended Address in commands directed to the SPW2 should always be zero.

**Memory Address** bytes form the bottom 32-bits of the memory address to which the data in a write command is to be written or from where data is to be read for a read command. Input/output registers and control/status registers are assumed to be memory mapped.

**Data Length** bytes form the 24-bit length of the data that is to be written or read. The length is the length in bytes with the most-significant byte of the length sent first.

**Header CRC** byte is an 8-bit Cyclic Redundancy Check (CRC) used to confirm that the header is correct before executing the command. The header CRC is formed using the CRC-8 code used in ATM (Asynchronous Transfer Mechanism). CRC-8 has the following polynomial:  $x^8 + x^2 + x^1 + 1$ , with a starting value of  $00_{16}$ . Each byte in the header starting with the destination logical address and ending with the byte before the header CRC itself is used in the calculation of the CRC. The CRC is implementing a Galois version Linear Feedback Shift Register.

**Data** bytes are the data that is to be written in a write command or the data that is read in a read response.

**Data CRC** is an 8-bit Cyclic Redundancy Check (CRC) used to confirm that the data is correct before being written in a verified write command or was correctly transferred in a non-verified write command or read reply. The data CRC starts with the byte after the header CRC and covers all the data bytes. The same CRC encoding is used as for the header CRC. Note that for a zero length data field the CRC is the starting value, i.e.  $00_{16}$ .

**EOP** character is the End Of Packet marker of the SpaceWire packet.

**EEP** character is the Error End of Packet marker of an erroneous SpaceWire packet.

10.5.9.2SpaceWire Packet Structure

A SpaceWire packet consists of a destination address (path address + logical address), a cargo and an end of packet (EOP or EEP) marker.

<destination address><cargo><end of packet>

where:

- The destination address consists of a list of one or more bytes, called destination identifiers: <destination address> = <id 0><id 1> ... <id N-1>
- The cargo contains zero or more bytes
- The end of packet is either an EOP, indicating a normal termination of a packet, or an EEP, indicating a packet in which an error has occurred.

Note that SpaceWire packets without any cargo, containing only a single EOP, are possible to transmit with the SpaceWire (SPW2) Module. If the SpaceWire (SPW2) Module receives such a packet, it will be discarded without any side effects.

10.5.9.3SpaceWire Transfer Protocol Packet Structure

The structure of the SpaceWire Transfer Protocol is defined in [RMAPID], and it is used for RMAP and other transfer protocol packets. The SPW2 module supports structures using non-extended protocol identification as shown in the figures below.

Destination Logical Address	Protocol ID	Remaining Header	Data cargo	EOP
Header			Data	
1 byte	1 byte	0 – 253 bytes	0 – 16 MiB-1	

Figure 10-3. Transfer Protocol packet structure using logical addressing



Destination Path Address	Destination Logical Address	Protocol ID	Remaining Header	Data cargo	EOP
Header				Data	
1 – 12 bytes	1 byte	1 byte	0 – 253 – Path Address Length bytes	0 – 16 MiB-1	

**Figure 10-4. Transfer Protocol packet structure using path addressing**

Where:

- The path address consists of a list of zero or more bytes, called path address: <path address> = <id 0><id 1> ... <id N-1>, they are stripped of the packet during its path through a routed network and only the logical address remains when the packet arrives at its destination node.
- The Logical Address consists of one byte.
- The Protocol ID consists of one byte.
- The cargo contains zero or more bytes of header and/or data. The SPW2 receiver limits the minimum packet size (header + data) to 4 bytes, and the SPW2 transmitter limits the maximum header size to 255 bytes.
- The end of packet is either an EOP, indicating a normal termination of a packet, or an EEP, which might appear before the nominal EOP position, indicating a packet in which an error has occurred

### SpaceWire Transfer Protocol RMAP packet structure

The supported packet structures for the RMAP SpaceWire transfer protocol, using Protocol ID equal to 01<sub>16</sub>, are shown in the paragraphs below.

Destination Logical Address	Protocol ID	Remaining RMAP Header	Optional RMAP Data and Data CRC	EOP
Header			Data	
1 byte	1 byte	Command or Response specific size (6 – 26 bytes)	0 – 16 MiB-1	

**Figure 10-5. RMAP Transfer Protocol packet structure using logical addressing**

Destination Path Address	Destination Logical Address	Protocol ID	Remaining RMAP Header	Optional RMAP Data and Data CRC	EOP
Header				Data	
1-12 bytes	1 byte	1 byte	Command or Response specific size (6 – 26 bytes)	0 – 16 MiB-1	

**Figure 10-6. RMAP Transfer Protocol packet structure using path addressing**

Note that a zero byte data cargo for read or write commands is allowed.

## RMAP Write Command format

The write command:

Dest. Path Addr	Dest. Path Addr	Dest. Path Addr	Dest. Path Addr
Dest. Path Addr	Dest. Path Addr	Dest. Path Addr	Dest. Path Addr
Dest. Path Addr	Dest. Path Addr	Dest. Path Addr	Dest. Path Addr
Dest. Logical Addr	Protocol ID	Command & Type	Dest. Device Key
Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Logical Addr	Transaction ID (MS)	Transaction ID (LS)	Extended Addr
Addr (MS)	Addr	Addr	Addr (LS)
Data length (MS)	Data length	Data length (LS)	Header CRC8
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data CRC8	EOP	

**Figure 10-7. RMAP transfer protocol packet structure for Write command**

Note that the shaded fields are optional and/or can contain a variable number of bytes. E.g. the amount of Destination Path Address bytes may be any number from 0 to 12.

Note that Header CRC8 is the last byte of the header. Note that the Data CRC8 byte is not optional.

## RMAP Write Response format

The write command response:

Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Logical Addr	Protocol ID	Command & Type	Status
Dest. Logical Addr	Transaction ID (MS)	Transaction ID (LS)	Header CRC8
EOP			

**Figure 10-8. RMAP transfer protocol packet structure for Write response**

Note that the shaded fields are optional and/or can contain a variable number of bytes. E.g. the amount of Source Path Address bytes may be any number from 0 to 12.

Note that there is no data cargo and thus no data CRC in this packet.

## RMAP Read Command format

The read command:

Dest. Path Addr	Dest. Path Addr	Dest. Path Addr	Dest. Path Addr
Dest. Path Addr	Dest. Path Addr	Dest. Path Addr	Dest. Path Addr
Dest. Path Addr	Dest. Path Addr	Dest. Path Addr	Dest. Path Addr
Dest. Logical Addr	Protocol ID	Command & Type	Dest. Device Key
Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Logical Addr	Transaction ID (MS)	Transaction ID (LS)	Extended Addr
Addr (MS)	Addr	Addr	Addr (LS)
Data length (MS)	Data length	Data length (LS)	Header CRC8
EOP			

**Figure 10-9. RMAP transfer protocol packet structure for Read command**

Note that the shaded fields are optional and/or can contain a variable number of bytes. E.g. the amount of Destination Path Address bytes may be any number from 0 to 12.

Note that there is no data cargo and thus no data CRC in this packet.

## RMAP Read Response format

The read command response:

Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Path Addr	Source Path Addr	Source Path Addr	Source Path Addr
Source Logical Addr	Protocol ID	Command & Type	Status
Dest. Logical Addr	Transaction ID (MS)	Transaction ID (LS)	Reserved = 0
Data Length (MS)	Data Length	DataLength (LS)	Header CRC8
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data CRC8	EOP	

**Figure 10-10.RMAP transfer protocol packet structure for Read response**

Note that the shaded fields are optional and/or can contain a variable number of bytes. E.g. the amount of Source Path Address bytes may be any number from 0 to 12.

Note that Header CRC8 is the last byte of the header. Note that the Data CRC8 byte is not optional.

## Command and Type field

The Command & Type field used in the RMAP command and response headers:

Command & Type		Command				Source Path Addr Len	
7	6	5	4	3	2	1	0
Reserved	Command	Write	Verify Data	Ack	Increment	Source Path Addr Len	
0					1		

MSB

LSB

Field:	Value	Description
Source Path Addr Len	00 <sub>2</sub>	No Source Path Addr Words
	01 <sub>2</sub>	One Source Path Addr Word
	10 <sub>2</sub>	Two Source Path Addr Words
	11 <sub>2</sub>	Three Source Path Addr Words (Leading bytes equal to zero are removed. But if Source Path Addr Len $\neq$ 0 and all bytes of the Source Path Address are zero, a single byte is kept.)
Increment	1	Increment Address (Data are written to consecutive addresses)
	0	Non-Incrementing Address (Data are written to the same address. Not supported by protocol handler.)
Ack	1	Acknowledge: Send Response.
	0	NoAcknowledge: Don't send a Response (NoAcknowledge is only possible for write commands.)

Verify Data	1	Check data CRC before writing. Only supported for 4 bytes word-aligned by hardware protocol handler. (Commands of other sizes or with mis-aligned addresses are rejected.)
	0	Don't check data CRC before write.
Write	1	Write command
	0	Read command
Command	1	The header is a command header.
	0	The header is a response header. Packet is passed on to software.
Reserved	0	Must always be 0.

Note that commands of formats not supported by the RMAP hardware implementation in the SpaceWire (SPW2) Module (like e.g. Non-Incrementing Address) are passed on to the software if the RMAP hardware support is turned off, provided that software support for transfer protocols is enabled.

## Extended Address field

The Extended Address field as supported by the RMAP hardware implementation in the SpaceWire (SPW2) Module:

Extended Address							
7	6	5	4	3	2	1	0
-				DMA Page			
MSB				LSB			
Field:	Value	Description					
DMA page	0000 <sub>2</sub> - 1111 <sub>2</sub>	Extended Address range, i.e. address bits A35:A32 above the normal 32 bit range					

Note that the SpaceWire (SPW2) Module does not support an Extended Address value above F<sub>16</sub>. This gives a total address range of 64 GiB.

### 10.5.9.4SpaceWire Virtual Channel Transfer Protocol packet structure (VCTP)

The virtual channel transfer protocol is a legacy SpaceWire packet protocol where the last byte before the data cargo of a packet contains the virtual channel identification information.

Logical Address	Protocol ID	Dummy	Virtual Channel ID	Data cargo	EOP
1 byte	1byte	1 byte	1 byte	0 – 16 MiB	

Figure 10-11.VCTP transfer protocol packet structure using logical addressing

Path Address	Logical Address	Protocol ID	Dummy	Virtual Channel ID	Data cargo	EOP
1-12 bytes	1 byte	1 byte	1 byte	1 byte	0 – 16 MiB	

Figure 10-12.VCTP transfer protocol packet structure using path addressing

The Protocol ID used for identifying a VCTP packet is defined in the SPW VC Transfer Protocol ID Register. The default value is F0<sub>16</sub>.

The Virtual Channel ID indicates on which virtual channel the data shall be stored. The SPW2 supports a configurable number of independent virtual receive channels (maximum of 7).

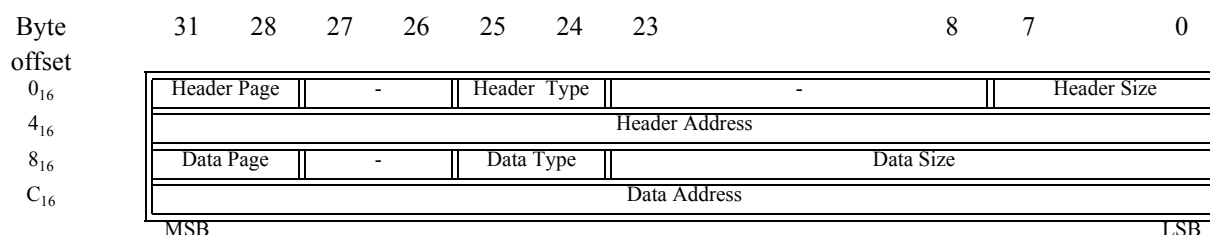
It is strongly recommended that the dummy byte is set equal to the virtual channel identifier, i.e. a copy. This allows the dummy byte to represent the VCID if the packet is trapped in the SPW First Failing Packet Register of a SpaceWire (SPW2) Module, where only the first 3 bytes of a received packet are stored.

Note that at least one byte must be present in the data cargo. If not, then an early EOP error indication is generated in the receiver.

### 10.5.9.5 Transmitter Send List entry structure

The Virtual Transmit Channels, TxVC[\$], use send lists to define what data is to be transmitted. A send list contains one or more send list entries, as defined hereafter.

#### SPW Send List entry



Field:	Value:	Description:
Header Page	0 - 15	Header Page Address, i.e. address bits A35:A32 enabling 64 Gbyte addressing space
Header Type	00 <sub>2</sub>  01 <sub>2</sub> -11 <sub>2</sub>	Send header and attach data without EOP in-between. (Note that automatic RMAP checksum is not supported since the routing byte(s) is included in header but should not be included in the checksum.) Illegal values. (Will be handled as 00 <sub>2</sub> by the SPW2.)
Header Size	0 – 255	Header size in bytes
Header Address	0 – 4 GiB -1	Header start address, (byte address).
Data Page	0 – 15	Data Page Address, i.e. address bits A35:A32 enabling 64 Gbyte addressing space
Data Type	00 <sub>2</sub> 01 <sub>2</sub>  10 <sub>2</sub> -11 <sub>2</sub>	Nominal or RMAP read command, puts EOP at the end. No data CRC. RMAP write command, CRC checksum is generated over the data and is put at the end followed by EOP. Illegal values. (Will be handled as 00 <sub>2</sub> by the SPW2.)
Data Size	0 – 16 MiB -1	Data size in bytes
Data Address	0 – 4 GiB -1	Data start address, (byte address)

**Figure 10-13.**Send list entry structure

Note that the send list elements must be word aligned.

### 10.5.10 Applicable Documents

[DERATE]	Derating Requirements applicable to electronic, electrical and electro-mechanical components for ESA space systems
[ESA_DAR]	ESA PSS-01-301, Issue 2 ESA ASIC Design and Assurance Requirements QC/172/RdM, Issue 1, June 1992
[CCSDS]	Packet Telemetry Standard, PSS-04-106, Issue 1
[SPW]	ECSS-E-STD-50-12C Space Engineering, SpaceWire – Links, Nodes, Routers and Networks
[RMAPID]	ECSS-E-ST-50-50C SpaceWire protocol identification
[RMAP]	ECSS-E-ST-50-51C SpaceWire - Remote memory access protocol
[AMBA]	AMBA Specification, Rev 2.0
[SPW2]	SpaceWire (SPW2) Module Specification, P-ASIC-SPC-00113-SE

### 10.5.11 Reference Documents

[ESA_DMR]	ESA ASIC Design and Manufacturing Requirements WDN/PS/700, Issue 2, October 1994
[CODEC_Desc]	SpaceWire CODEC VHDL Functional Description, Issue 1.2, 3-March-2004
[CODEC]	SpaceWire CODEC VHDL User Manual, Issue 1.1, 19-April-2004

## 10.6 Functional behaviour

This section describes the SpaceWire (SPW2) Module software interface. The description is divided into subsections, which treats different parts of the SpaceWire (SPW2) Module.

The description is divided into:

- **General**  
Short introductory description of the SPW2 module
- **SpaceWire link**  
Describes the low level handling of the SpaceWire link
- **SpaceWire transfer protocol support in the receiver**  
Describes protocol identification and support enabling
- **Remote Memory Access Protocol (RMAP)**  
Describes the handling of incoming RMAP commands
- **Virtual Receive Channels, RxVC**  
Describes the handling of incoming VCTP packets
- **Virtual Transmit Channels, TxVC**  
Describes the handling of send lists and transmission of VCTP packets and RMAP commands
- **Time-Codes**  
Describes the handling of time-codes
- **First Failing Packet Register**  
Describes the handling of the First Failing Packet Register
- **SpaceWire CODEC**  
Describes the handling of the SpaceWire CODEC
- **Initialisation**  
Describes the initialisation sequence for the module
- **Operation/Usage**  
Describes how the module shall be used during normal operation
- **Error Handling**  
Describes how to handle internal errors in the module, e.g. address error.
- **Usage Constraints**  
Describes actions that are not allowed and the resulting consequences.
- **Examples**  
Gives a few examples how to perform different tasks using the module.
- **Interrupt Handling**  
Describes the interrupt handling of the module

### 10.6.1 General

The SpaceWire (SPW2) Module comprises receive and transmit capabilities supporting several protocols. The core of the module is the SpaceWire link codec, described in [CODEC], which implements the low level SpaceWire protocol receive and transmit handling. On packet level the receiver implements two transfer protocols in hardware and provides support for processing of other unknown protocols in software.

The receiver comprises an identification stage which identifies the different transfer protocols and routes the data to the corresponding handler. The identification is based on the Protocol Identifier concept described in [RMAPID]. During the identification several types of error conditions can be detected and are reported in the First Failing Packet Register. This register is also used for reporting errors detected during the reception of the rest of the packet.

After the identification, the incoming packet is handled by one of the following:

- Remote Memory Access Protocol (RMAP) handler implemented in hardware, or
- Virtual Receive Channel (RxVC), or

- Other protocols, which are redirected to software.

The Remote Memory Access Protocol (RMAP) protocol is implemented in hardware with some constraints. Each received command that has passed the identification process is checked for header checksum consistency and for a matching Destination Key. Also the data checksum is checked for write commands that require verification before the command is executed. For all other write commands, the data is checked in parallel with the execution and a potential error is reported in the RMAP response if required by the incoming command. Data are read or written from and to the internal bus using directed memory accesses. An RMAP response is generated for all read commands and for write commands if so required by the incoming command. No software support is required for these operations. Any failures logged in the First Failing Packet Register.

The Virtual Channel Transfer Protocol (VCTP) protocol implements one or more virtual receive channels (RxVC). The Virtual Channel Identifier in the VCTP header is used for routing the packet to the corresponding RxVC.

Note that unknown protocols, and RMAP commands and responses, can be routed to a dedicated virtual receive channel RxVC[0] for further processing. As an additional service, any RMAP command or response received on RxVC[0] is also checked for combined header and data checksum errors, which are reported as an interrupts to software. Any failure is logged in the First Failing Packet Register.

The transmitter comprises one or more Virtual Transmit Channels (TxVC). Any type of packets can be transmitted using these channels. The transmitters provide support for data checksum generation for RMAP commands and responses.

The SpaceWire (SPW2) Module is able to perform simultaneous transfers as described hereafter:

- Transmit an RMAP command and simultaneously receive an RMAP response.
- Receive, verify and execute an RMAP command, and simultaneously start transmitting an RMAP response if so required by the command.
- Transmit an RMAP command and simultaneously receive a Virtual Channel Transfer Protocol packet, VCTP.
- Receive, verify and start executing an RMAP command, and simultaneously transmit a Virtual Channel Transfer Protocol packet, VCTP.
- Transmit a Virtual Channel Transfer Protocol packet, VCTP, and simultaneously receive an RMAP response.
- Transmit an RMAP response and simultaneously receive a Virtual Channel Transfer Protocol packet, VCTP.
- Simultaneously receive and transmit Virtual Channel Transfer Protocol packets, VCTP.
- Receive, verify and execute an RMAP command, and simultaneously transmit an RMAP command to another node.
- Transmit an RMAP response, and simultaneously receive an RMAP response to a previously transmitted RMAP command.

There are no known limitations to the simultaneous receive and transmit capabilities of the SpaceWire (SPW2) Module.

## 10.6.2 SpaceWire Link

The SpaceWire link is implemented using the University of Dundee SpaceWire CODEC, which implements low level transmission and reception over the SpaceWire link.

The SpaceWire CODEC contains a Link Interface Control State Machine, referred to as LICSM, that controls the overall operation of the link interface. The LICSM provides link initialisation, normal operation and error recovery services.

The SpaceWire CODEC contains status and configuration signals that are read and set via the internal register bus. Time-codes to be received or transmitted by the SpaceWire CODEC are read or written via registers. The detailed description of the SpaceWire CODEC can be found in [CODEC].



### 10.6.2.1 Link configuration and start-up

Link start-up works as seen in the figure below. Link start and disable is controlled by the SPW CODEC Configuration Register. Note that when reset this register enables the auto start mode. This means that the CODEC at start-up will remain in the Ready state, where the Tx remains reset, until a NULL token is received from the node at the other end of the link. Thus, it is possible for the SpaceWire link to autonomously start-up after reset and to allow RMAP commands to be executed without the need for any interaction from software in the destination node. This also means that for the start-up to proceed to the Run state, the other node must be set in the LinkStart mode, in which the transition from Ready to Started takes place without waiting for a NULL token.

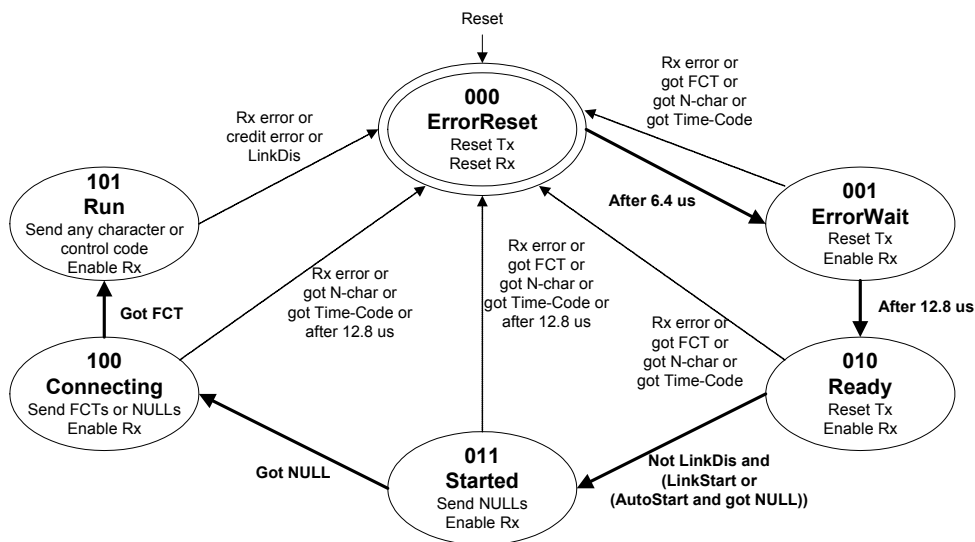


Figure 10-14.Space Wire link state diagram

The SpaceWire link is always (if implemented in the system accordingly) able to start-up at a 10 Mbit/second rate without the need for any configuration by software. This makes it possible to configure or initialise a destination node after reset using RMAP commands without the destination node being operational.

### 10.6.2.2 Link transfer and receive rates

The SpaceWire Module supports full data bandwidth on the receiver and transmitter for the following clock frequency relations between *BusClk* and *SpwClk*:

*SpwClk* has a frequency not higher than 2.5 times the *BusClk* frequency. If this relation is not fulfilled, the SPW2 will still be functionally correct, but the full nominal data bandwidth can not be maintained. (Occasional NULL-tokens will be interspersed with the data.)

Since the SPW2 implements double data rate (DDR) this corresponds to a maximum SpaceWire transmit bit rate of  $5 \cdot \text{BusClk}$  [Mbit/second].

The nominal transmit data rate is configured via the SPW Clock Division Register. The input *SpwClk* frequency can be divided by a denominator in the range of 0.5 to 32 to produce the nominal transmit data rate. Note that the DDR implementation only gives a 50/50 duty cycle when an even value is written in SPW Clock Division Register, i.e. for division rates 0.5, 1.5, 2.5, 3.5 etc, thus affecting signal timing.

The transmit data rate of 10 Mbit/second  $\pm$  10%, which is required for a reliable link start-up, is possible to derive after reset from the *SpwClk* clock and the clock prescaler, controlled by the configuration inputs *SpwClk10MBit* and *SpwClkMul*. The reset value of the SPW Clock Division Register corresponds to the link start-up bit rate.

The *SpwClkMul* configuration input allows for divisions by (1/4, 1/3, 1/2, 1/1).

The *SpwClk10MBit* configuration input allows for divisions by (1/10, 1/8, 1/6, 1/5, 1/4, 1/3, 1/2, 1/1).

The configuration inputs *SpwClk10MBit* and *SpwClkMul* support the following *SpwClk* frequency values:

10, 20, 30, 40, 50, 60, 80, 90, 100, 120, 150, 160, 180, 200, 240, 300 and 320 MHz

Note that  $1/4 * 1/10 = 1/40$ , which would give a *SpwClk* value of 400 MHz, is not a legal value, since the SPW Clock Division Register can't be programmed to a division rate higher than 32.

A variance of  $\pm$  10 % of the above values is allowed, still meeting the SpaceWire link start-up frequency requirement.

A switch to the nominal transmit data rate is automatically performed when the link start-up has been completed. After a link disconnection, the link start-up transmit data rate is automatically used again until the link start-up has been completed. Note that NULL characters are sent with the nominal transmit data rate after the completion of the link start-up.

The nominal transmit data rate is changed immediately when a write access is made to the SPW Clock Division Register. The nominal data rate can be changed without disrupting the link even after link start-up.

### 10.6.2.3 Link status

The SPW CODEC Status Register reflects the state of the CODEC. Note that all fields except LinkState, TxParked and RxParked are included for debug purposes only, and should not be used in normal operation.

The detected link interface errors are issued as interrupts after the corresponding events are reported from the SpaceWire CODEC and are reported in the SPW Link Interrupt Status Register.

### 10.6.3 SpaceWire transfer protocol support in the receiver

The receiver in the SpaceWire (SPW2) Module supports the Protocol Identification process defined in [RMAPID]. The identification is performed on all data received over the SpaceWire link, and the result is used for routing the incoming data to the corresponding protocol handler. Note that the SpaceWire (SPW2) Module does not support any type of protocol based on other kinds of protocol identification concepts. However, the Protocol Identification process can be bypassed by setting the SPW RxVC Config Register [0].ForceUnk, thus causing all packets of four bytes or more to be treated as an unknown transfer protocol.

The following SpaceWire Transfer Protocols are explicitly identified by the hardware:

- Virtual Channel Transfer Protocol, VCTP, the default Protocol ID being  $F0_{16}$ .
- RMAP Transfer Protocol, Protocol ID being  $01_{16}$ . Note that this Protocol ID is reserved for RMAP, even if RMAP support is turned off in the module, and should never be used for e.g. VCTP.

Unknown protocols, i.e. protocols not explicitly identified by hardware, are redirected to software via RxVC[0].

The Virtual Channel Transfer Protocol supports one or more virtual channels. A virtual channel identifier is used for identifying the corresponding virtual channel. A virtual channel identifier of zero is not allowed, as explained below, neither are virtual channel identifiers above 7.

A special case is the virtual channel RxVC[0] which is used for the reception of other transfer protocols than VCTP. This includes RMAP commands and responses.

Received RMAP commands are either handled by a protocol handler in hardware, or forwarded to virtual receive channel RxVC[0] for further software processing.

Received RMAP responses are also forwarded to virtual receive channel RxVC[0] for further software processing.

Note that the virtual receive channel RxVC[0] is always implemented, even if no RMAP hardware handler is implemented. Note that when the RMAP hardware handler is implemented, the dedicated virtual transmit channel TxVC[0] is also implemented. This virtual transmit channel is used for automatic RMAP response generation and cannot be accessed by the software. In the transmit bandwidth arbitration, TxVC[0] has priority over any other TxVC[\$] channel. TxVC[0] can however not starve the other TxVC[\$] channels, since there is always time to transmit on the other channels while an RMAP command is being received.

### 10.6.3.1 Enabling of protocol support

Enabling of the Virtual Channel Transfer Protocol, VCTP, support is performed individually per virtual channel. The only input used for the identification processes is whether a virtual receive channel is implemented and if it is enabled by software.

Enabling of the Remote Memory Access Protocol, RMAP, support is performed in two ways:

- Enabling of hardware supported RMAP
- Enabling of software supported RMAP

Enabling of RMAP command support in hardware is done by means of the SPW RxVC Config Register[0].RMAPEn bit. This enables both hardware execution of RMAP commands and hardware generation of the corresponding RMAP responses. The reset value for this bit is RMAP enabled for both modules. It is thus possible to disable hardware handling of RMAP commands during operation.

Enabling of RMAP command and response support in software is done by means of the SPW RxVC Config Register[0]. When enabled, all RMAP commands that are not handled by the RMAP command support in hardware are forwarded to RxVC[0] for further software processing. Note that if the RMAP command support in hardware is not enabled or implemented, then all RMAP commands can still be forwarded to RxVC[0]. When enabled, all RMAP responses are forwarded to RxVC[0] for further software processing. Software handling is by default disabled at reset.

If neither hardware nor software RMAP support is enabled, the Spw2 will auto-generate responses when required, giving Authorisation Error.

Enabling of support for unknown transfer protocols is done in the same way as for software support for RMAP commands and responses, using the same resources.

### 10.6.3.2 Identification of received protocol

The SpaceWire (SPW2) Module identifies an incoming packet using the first bytes of the packet. The identification is only performed when four bytes have been received. No identification will take place if an EOP or EEP has been detected before this, and the packet will be rejected with a header error being reported. Packets with no data, containing only a single EOP or EEP, do not trigger the identification process and are discarded without any error being reported.

The following steps are performed during the identification:

- Check exact match of the first byte (DLA) with the internally programmed DLA.
- Check the second byte (Protocol ID).
- Check the third byte, if applicable, for a valid RMAP command
- Check fourth byte, if applicable, for a valid VCID.
- Reject and discard the packet and issue an interrupt, if no enabled RxVC[\$], RxVC[0] or RMAP hardware handler could be identified.

The identification process results in one of the following:

- The received packet belongs to the VCTP protocol and is forwarded to the corresponding RxVC[\$] channel when it is implemented and enabled;
- The received packet is a hardware supported RMAP command, and is forwarded to the RMAP handler when it is enabled;
- The received packet is a RMAP command not supported in hardware, or the hardware is disabled, and is forwarded to RxVC[0] when it is enabled;
- The received packet belongs to an unknown protocol, or forced unknown protocol interpretation is configured through SPW RxVC Config Register [0].ForceUnk, and is forwarded to RxVC[0] when it is enabled;
- The received packet is rejected, and all data is discarded until the next EOP or EEP.

The SpaceWire (SPW2) Module rejects packets:

- If the packet DLA does not match the corresponding programmable field in the SPW RxVC Config Register[0]. Note that for RMAP commands a response will be generated if the packet is otherwise correct and an acknowledge is requested; and a received RMAP response will be stored if enabled for RxVC[0]. DLA is not checked if forced unknown protocol interpretation is configured.
- If the packet Protocol ID does not correspond to neither RMAP nor VCTP, and no software handling is enabled for RxVC[0].
- If the packet belongs to the VCTP protocol and has a virtual channel address that is outside the implemented or allowed range, or if the virtual channel is disabled.
- If the RMAP command byte is either illegal, or not supported in hardware, and no software handling is enabled for RxVC[0].

Errors are reported in the First Failing Packet Register and SPW Link Interrupt Status Register.

Note that the identification process is always performed, even if no hardware support for RMAP commands is implemented or enabled. The continued checking of the full RMAP header is done by the RMAP handler in either hardware or software. E.g. the Destination Key is checked by hardware in hardware supported commands, and by software in software supported commands. The identification is considered finished after the four first bytes have been examined.

Note that the RMAP command/header CRC checksum need not be correct during the identification. The CRC checksum is checked subsequently by the RMAP handler in either hardware or software

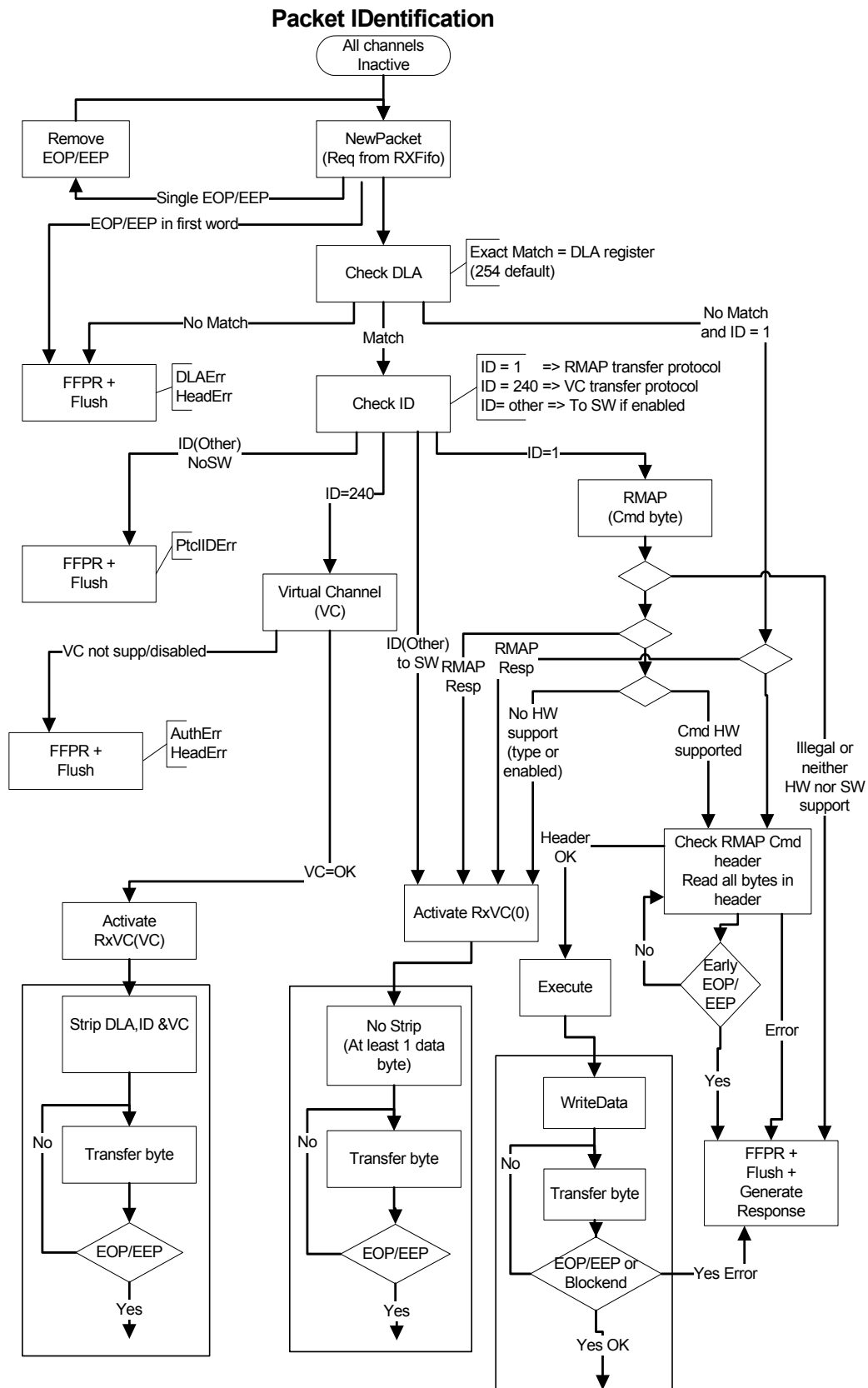


Figure 10-15.Receiver packet identification and qualification flow diagram

## 10.6.4 Remote Memory Access Protocol (RMAP)

### 10.6.4.1 Hardware supported RMAP commands

The SpaceWire (SPW2) Module supports in hardware the Remote Memory Access Protocol [RMAP] with the following constraints:

- *Increment Address* setting only, with no support for *No-Increment Address* setting.
- *Extended Address* support, using the 4 least significant bits only (thus limiting the address range to  $2^{(32+4)}=64$  GiB).
- *Verified Write* commands only for 4 byte block sizes and with word-aligned addressing.
- No *Read-Modify-Write* command support.

The following RMAP commands are handled in hardware when enabled:

- *Read* commands with *Incrementing Address*, an *Extended Address* less or equal to  $0F_{16}$ , and a mandatory acknowledge request.
- *Write* commands with *Incrementing Address*, an *Extended Address* less or equal to  $0F_{16}$ , and with or without an acknowledgement request.
- *Verified-Write* commands with *Incrementing Address*, an *Extended Address* less or equal to  $0F_{16}$ , with exactly a 4 byte block size, with word-aligned addressing, and with or without an acknowledgement request.

The SpaceWire (SPW2) Module supports in hardware only Big-Endian Byte and HalfWord to/from Word conversions. I.e. first byte at address zero is written to bit positions (31:24) in a Word and fourth byte at address 3 is written to bit positions (7:0) in the same Word. Little-Endian conversion is not supported.

### 10.6.4.2 Software supported RMAP commands and responses

Since the protocol identification described in 10.6.3.2 is performed only on the four first bytes of a packet, only the Type and Command field byte is used for determine which commands can be handled in hardware and which should be forwarded to memory for further software processing, provided that this support is enabled.

The following RMAP commands are therefore forwarded to software when hardware support is enabled:

- *Read* commands with *No-Increment Address* setting.
- *Write* and *Verified-Write* commands with *No-Increment Address* setting.
- All *Read-Modify-Write* commands.

The following RMAP commands are therefore not forwarded to software when hardware support is enabled and are thus rejected during header verification:

- Any otherwise hardware supported command with an *Extended Address* value larger than  $0F_{16}$ .
- *Verified Write* commands with any other block size than 4 bytes, or with a non-word-aligned address.

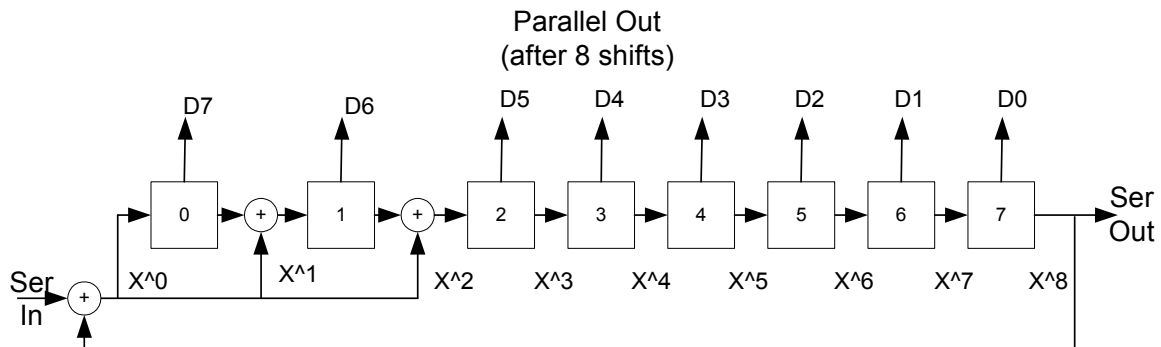
RMAP commands and responses with the Reserved bit of the Command & Type field set, and RMAP responses with the Ack bit cleared, are rejected and thus not forwarded to software.

### 10.6.4.3 Header and data verification

The SpaceWire (SPW2) Module supports command header and data field verification according to [RMAP], as well as combined response header and data verification which is used for RMAP commands and responses forwarded to software.

8-bit CRC checking is used for both header and data (implemented as a Galois version LFSR, with the least significant bit being used first). The generator polynomial is  $g(x) = x^8 + x^2 + x^1 + 1$ . The least significant bit is used first in the CRC generation algorithm as the SpaceWire protocol sends the least significant bit of a byte first on the link.

Note that the Galois version LFSR generates the result zero if the checksum itself is included in the generation. This characteristic allows the calculation of a checksum over the header field, the header checksum, and the data field, which should match the checksum calculated over the data field alone. Note also that the checksum is only calculated for the bytes received at the destination, i.e. the eventual Destination Path Address bytes (which are stripped off along the way) are not included.



**Figure 10-16. Galois LFSR implementation of CRC8 polynomial**

Here is an implementation example of a CRC computation table in C:

```
static natural8 RMAP_CRCTable[256] = {
    0x00, 0x91, 0xe3, 0x72, 0x07, 0x96, 0xe4, 0x75,
    0x0e, 0x9f, 0xed, 0x7c, 0x09, 0x98, 0xea, 0x7b,
    0x1c, 0x8d, 0xff, 0x6e, 0x1b, 0x8a, 0xf8, 0x69,
    0x12, 0x83, 0xf1, 0x60, 0x15, 0x84, 0xf6, 0x67,
    0x38, 0xa9, 0xdb, 0x4a, 0x3f, 0xae, 0xdc, 0x4d,
    0x36, 0xa7, 0xd5, 0x44, 0x31, 0xa0, 0xd2, 0x43,
    0x24, 0xb5, 0xc7, 0x56, 0x23, 0xb2, 0xc0, 0x51,
    0x2a, 0xbb, 0xc9, 0x58, 0x2d, 0xbc, 0xce, 0x5f,
    0x70, 0xe1, 0x93, 0x02, 0x77, 0xe6, 0x94, 0x05,
    0x7e, 0xef, 0x9d, 0x0c, 0x79, 0xe8, 0x9a, 0x0b,
    0x6c, 0xfd, 0x8f, 0x1e, 0x6b, 0xfa, 0x88, 0x19,
    0x62, 0xf3, 0x81, 0x10, 0x65, 0xf4, 0x86, 0x17,
    0x48, 0xd9, 0xab, 0x3a, 0x4f, 0xde, 0xac, 0x3d,
    0x46, 0xd7, 0xa5, 0x34, 0x41, 0xd0, 0xa2, 0x33,
    0x54, 0xc5, 0xb7, 0x26, 0x53, 0xc2, 0xb0, 0x21,
    0x5a, 0xcb, 0xb9, 0x28, 0x5d, 0xcc, 0xbe, 0x2f,
    0xe0, 0x71, 0x03, 0x92, 0xe7, 0x76, 0x04, 0x95,
    0xee, 0x7f, 0x0d, 0x9c, 0xe9, 0x78, 0x0a, 0x9b,
    0xfc, 0x6d, 0x1f, 0x8e, 0xfb, 0x6a, 0x18, 0x89,
    0xf2, 0x63, 0x11, 0x80, 0xf5, 0x64, 0x16, 0x87,
    0xd8, 0x49, 0x3b, 0xaa, 0xdf, 0x4e, 0x3c, 0xad,
    0xd6, 0x47, 0x35, 0xa4, 0xd1, 0x40, 0x32, 0xa3,
    0xc4, 0x55, 0x27, 0xb6, 0xc3, 0x52, 0x20, 0xb1,
    0xca, 0x5b, 0x29, 0xb8, 0xcd, 0x5c, 0x2e, 0xbf,
    0x90, 0x01, 0x73, 0xe2, 0x97, 0x06, 0x74, 0xe5,
    0x9e, 0x0f, 0x7d, 0xec, 0x99, 0x08, 0x7a, 0xeb,
    0x8c, 0x1d, 0x6f, 0xfe, 0x8b, 0x1a, 0x68, 0xf9,
    0x82, 0x13, 0x61, 0xf0, 0x85, 0x14, 0x66, 0xf7,
    0xa8, 0x39, 0x4b, 0xda, 0xaf, 0x3e, 0x4c, 0xdd,
    0xa6, 0x37, 0x45, 0xd4, 0xa1, 0x30, 0x42, 0xd3,
    0xb4, 0x25, 0x57, 0xc6, 0xb3, 0x22, 0x50, 0xc1,
```

```

        0xba, 0x2b, 0x59, 0xc8, 0xbd, 0x2c, 0x5e, 0xcf,
    };

```

Computation of the CRC of Data(1 to M) should be done as follows:

$CRC(0) = 0$

$CRC(N) = RMAP\_CRCTable[CRC(N-1) \text{ xor } Data(N)]$ , for  $N = 1$  to  $M$

## Hardware supported RMAP commands

When the hardware support is enabled, the SpaceWire (SPW2) Module automatically verifies the command header and data CRC according to [RMAP] for all hardware supported commands as follows:

- Check header CRC checksum for RMAP command.
- Check data CRC checksum for RMAP command.
- Reject the RMAP command if the header CRC checksum is erroneous.
- Reject the RMAP command if the data CRC checksum is erroneous for a *Verified-Write* command.
- Reject superfluous write data in case of late EOP or EEP.

When the hardware support is enabled, the SpaceWire (SPW2) Module automatically generates the RMAP response header and data CRCs according to [RMAP], for the hardware supported commands only.

## Software supported RMAP commands and responses

The SpaceWire (SPW2) Module automatically verifies the combined command header and data CRCs for non-hardware supported commands as follows:

- Check CRC for the complete RMAP command, as calculated over header, header checksum and data.
- Issue the *CRCDataErr* Interrupt if the total checksum is not correct and RMAP software handling is enabled. The interrupt is issued after all data have been stored to memory.
- Remove the data CRC checksum byte before the command is stored in memory for further software processing. Note that the header CRC is not removed, even in read commands.

The SpaceWire (SPW2) Module automatically verifies the combined response header and data CRCs as follows:

- Check CRC for the complete RMAP responses, as calculated over header, header checksum and data.
- Issue the *CRCDataErr* Interrupt if the total checksum is not correct and RMAP software handling is enabled. The interrupt is issued after all data have been stored to memory.
- Remove the data CRC checksum byte before the response is stored in memory for further software handling.

Note that the RMAP response headers cannot be checked explicitly, as the position of the checksum is unknown to the hardware. Note that verification of the data consistency for other transfer protocol cannot be performed since the protection algorithms are unknown.

### 10.6.4.4 Destination Key verification

When the hardware support is enabled, the SpaceWire (SPW2) Module supports the RMAP command Destination Key for hardware supported command as follows:

- An exact match between the RMAP command Destination Key field and the SPW RMAP Destination Key Register (default  $00_{16}$ ) must occur for an RMAP command to be executed.
- If an incoming RMAP command with a non-matching Destination Key arrives, the RMAP command is rejected during the header check and the Invalid Destination Key error response is generated when a reply is requested by the incoming RMAP command.



#### 10.6.4.5 RMAP responses

The SpaceWire Module generates automatically RMAP responses according to [RMAP] for commands executed by the hardware. Responses are only generated if an acknowledgement is requested in the received RMAP command.

The SpaceWire Module generates the following errors in accordance with [RMAP]:

- *RMAP Command not supported:*  
RMAP Command field combination unused according to [RMAP].
- *Invalid destination key:*  
If there is not an exact match of the destination key field.
- *General error code:*  
If error during data transfer to internal bus, e.g. due to illegal address or data protection error detected on internal bus.
- *Invalid data CRC:*  
If RMAP data CRC error.
- *Early EOP:* If early EOP in data, i.e. EOP has been received with less data than expected from the RMAP command header.
- *Cargo Too Large:* If late EOP or EEP in data, i.e. EOP or EEP has been received with more data than expected from the RMAP command header..
- *Early EEP:* If early EEP in data, i.e. EEP has been received with less data than expected from the RMAP command header.
- *Authorisation error:*  
If RxVC Config Register[0].RMAPEn and RxVC Config Register[0].RXEn are disabled, or if RxVC Config Register[0].RMAPEn and RxVC Config Register[0].TrPtcISW are disabled. I.e. either hardware or software support, or both, are disabled for the specific RMAP command. Or if a command was supposed to be handled by hardware, based on the Command & Type field, but was later found to be not supported; e.g. verified write to a mis-aligned address.
- *Invalid destination logical address:*  
If header CRC decoded correctly but non-matching DLA.

The SpaceWire (SPW2) Module generates RMAP responses as follows:

- The common Tx DMA channel (which is shared with the virtual transmit channels) is used for RMAP responses..
- In case a DMA error occurs when reading data via the Tx DMA; an EEP is inserted and the DMA is released.

No RxVC interrupts are issued when an RMAP command is rejected, since it is up to the RMAP source to handle this situation using the RMAP response information. The PktRej interrupt is issued, though, every time the SPW First Failing Packet Register is triggered.

#### 10.6.5 Virtual Receive Channels, RxVC

The Virtual Channel Transfer Protocol, VCTP, implements Virtual Receive Channels, RxVC[\$]. Each virtual channel allows transfers from the SpaceWire link to its own programmable area in memory. This area constitutes a memory buffer and can be located anywhere in memory. Access to the memory area is performed by means of direct memory access. Each virtual receive channel features a set of DMA handling registers, which are used for configuring, controlling and monitoring the virtual channel.

A channel is individually enabled and triggered by software via the DMA handling registers. The selection of the virtual channel to which the received data from the SpaceWire link is to be routed is made using the Virtual Channel Identifier, VCID. An incoming packet is matched to fit into any of the triggered and enabled channels and if a match occurs a data transfer to memory starts. The matching is done during the identification of the packet, as previously described.

For each incoming packet, the four first bytes are stripped and the remainder of the packet is stored in the memory buffer consecutively until a halt event occurs (i.e. a predefined number of packets are received, an upper buffer boundary is reached or an error occurs). The channel then has to be re-triggered to continue its operation. The concept of triggering is essential as incoming packets otherwise could stall the SpaceWire network from the source to the destination, including any pending RMAP commands.

A virtual channel identifier of zero is not allowed for the virtual channel transfer protocol, VCTP, as explained below, neither are virtual channel identifiers above 7 (or less if so configured at module instantiation).

An additional virtual receive channel, RxVC[0], is provided for the handling of the RMAP transfer protocol and for any future transfer protocols. The RxVC[0] virtual receive channel is treated in the same way as the virtual channels described above, except that the first four bytes of a packet are not stripped.

The dedicated RxVC[0] is used for the following purposes:

- For all RMAP responses
- For any RMAP command not supported in hardware (or for all RMAP commands when hardware support is not enabled).
- For all unknown transfer protocols.

Note that RMAP commands that are supported by hardware do not use RxVC[0].

When SPW RxVC Config Register [0].WordAlign is cleared, the first byte of a packet is written at the address directly after the last byte of the previous one. Bytes outside a packet with mis-aligned start or end will not be modified. When the bit is set, the end of every packet is padded (with arbitrary content) up to the closest 32-bit word boundary; i.e. the first byte of a packet will always be word-aligned.

The last byte of a packet will always be written before, and never simultaneously with, the first byte of the next. I.e. if a packet ends in mid-word and the next packet arrives directly after, the word will still be accessed at least twice.

Four consecutive bytes of the same packet will, if word-aligned, be written in a single word access.

When possible the SPW2 will perform burst writes in order to preserve internal bus bandwidth.

#### 10.6.5.1 Configuration

The Virtual Channel Transfer Protocol, VCTP, uses a Protocol Identifier equal to the SPW VC Transfer Protocol ID Register, the default value being F0<sub>16</sub>.

Each virtual receive channel, RxVC[\$], provides the following configurability through a set of registers:

- An *RxVC Enable/Disable* bit.
- An *RxVC Word Alignment Enable/Disable* bit.
- An *RxVC DMA Base Address* and an *RxVC DMA Page Address*, supporting up to 64 Gbyte addressing space.
- An *RxVC DMA Block Size* and a counter, *RxVC DMA Offset*, supporting blocks of up to 16 Mbytes-1.
- An *RxVC Packet Count Trigger* configuring the number of packets to be received before finalising a block transfer.

#### 10.6.5.2 Status

Each virtual receive channel, RxVC[\$], provides the following status monitoring:

- An *RxVC Current Packet Status Register* that holds the present packet start address, which can be used for recovery purposes if packet reception fails.
- An *RxVC Status Register* indicating the present packet count position, whether the RxVC channel is triggered (started), and whether the channel is active (transfer ongoing).

#### 10.6.5.3 Interrupts

When a valid RxVC[\$] has been identified and the selected channel has started the handling of the received packet, the RxVC, when triggered, can issue interrupts as defined in SPW Rx VC Interrupt Status Register [\$]. Further reception is halted for the virtual channel until re-triggered by software.

## 10.6.6 Virtual Transmit Channels, TxVC

The SpaceWire (SPW2) Module implements Virtual Transmit Channels, TxVC[\$]. Each virtual channel allows transfers from memory to the SpaceWire link. Each transmitter virtual channel has a uniquely defined send list stored in memory. The send list entries define what data is to be sent from memory. Each send list entry specifies the position and size of the header and the position and size of the data to be transmitted. The SpaceWire (SPW2) Module performs direct memory accesses to read the send list, header and data to be transmitted. Each virtual transmit channel features a set of DMA handling registers, which are used for configuring, controlling and monitoring the channel.

All TxVC are handled by a common DMA handler. The selection of the virtual channel from which data are to be transmitted on the SpaceWire link is performed by round-robin arbitration for each send list entry.

Note that for the SPW2 module receiver, a virtual channel identifier of zero is not allowed for the virtual channel transfer protocol, VCTP, as explained hereafter, neither are virtual channel identifiers above 7 (or less if so configured at module instantiation).

Note that RMAP responses automatically generated when RMAP hardware support is enabled utilize a virtual transfer channel that is not under the control of the software. The RMAP response is included in the arbitration of the SpaceWire link, and has the highest priority.

Note that TxVC and RxVC are independent and there is no connection between e.g. TxVC[3] and RxVC[3].

All (read) memory accesses of the TxVC are on word format. When possible the SPW2 will perform burst reads in order to preserve internal bus bandwidth.

### 10.6.6.1 Configuration

Each virtual transmit channel, TxVC[\$], provides the following configurability:

- *TxVC Send List Pointer Register*, a pointer to the first word in the send list entry (called current position during transmission).
- *TxVC Send List Size Register*, the size of send list, counted in number of send list entries (indicating the remaining size during transmission).
- *TxVC Enable/Disable* bit

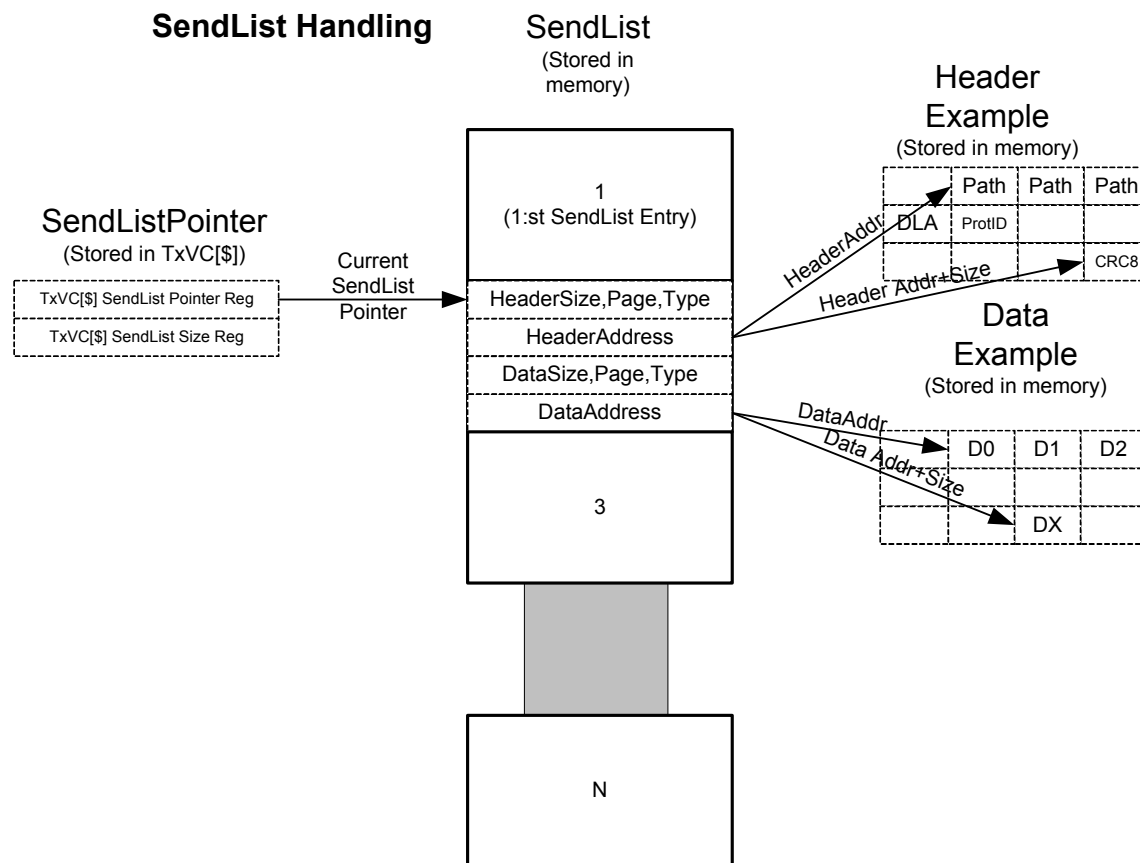
The number of send lists is equal to the number of TxVC channels. The SpaceWire Module supports send lists containing from 0 element up to 255 elements and send list locations within a 32 bit addressing range. Note that extended addressing cannot be used, when defining a send list location, i.e. page address equal to zero is assumed.

A send list entry consists of four words, the first word being the Header Size, Type and Page word. The fields and usage of the fields is defined in the Transmitter Send List entry structure.

Note that send list sizes of zero and send list entries with header and/or data sizes being zero are supported. If both header and data sizes are set to zero, only a single EOP will be transmitted. If both header and data sizes are set to zero and CRC checksum generation is enabled, an all-zero byte followed by an EOP will be transmitted.

Direct memory access using word based data width is always used to fetch header or data.

An example of a send list structure is shown below:



**Figure 10-17. TxVC send list handling**

### 10.6.6.2 Arbitration

The arbitration between virtual transmit channels is performed as follows:

- A new TxVC is arbitrated after each completed transmission of a send list entry.
- The arbitration only takes into account the TxVC[\$] that have been triggered.
- The arbitration uses a round robin activation algorithm for the TxVC[\$] channels, except for the RMAP response sent on TxVC[0], which always has the highest priority.

### 10.6.6.3 Transmission

The transmission on a virtual transmit channel adheres to the following procedure:

At start of send list:

- After a write access to the SPW TxVC Send List Size Register[\$], the SPW TxVC Status Register[\$].ChTrig is set, indicating that the transmission is triggered. This is only needed when a new send list is started.

For each send list entry:

- TxVC arbiter defines when an atomic send list entry handling can start by setting SPW TxVC Status Register[\$].ChAct.
- The TxVC[\$] uses the SPW TxVC Send List Pointer Register[\$] as address and fetches the first two words of the send list entry (i.e. header address, size, type, and page) and loads and starts the common DMA handler with this content.

- The header bytes fetched are directly passed on to the TxFIFO for transmission. Note that both the DMA start address and size can be misaligned and only the portion starting from DMA start address and the exact number of bytes are transmitted. A null size is also possible, in which case no bytes are fetched.
- Next, the second part of the send list entry (i.e. data address, size, type, and page) are fetched, followed by a reload and start of the common DMA handler channel with this content.
- The data fetched is directly passed on to the TxFIFO for transmission. Note that both the DMA start address and size can be misaligned and only the portion starting from DMA start address and the exact number of bytes is transmitted. A null size is also possible, in which case no bytes are fetched.
- During data fetch, the CRC8 checksum is calculated over the full data transmission. The result is added after the last data byte transmitted, when the Data Type field is set to an RMAP write command.
- Finally, an EOP is transmitted.
- The SPW TxVC Send List Size Register[\$] is decremented by one.
- The SPW TxVC Send List Pointer Register[\$] is incremented by 16.
- The SPW TxVCStatus Register[\$].ChAct is then cleared, i.e. allowing the TxVC arbiter to reassign another channel.

At send list completion:

- When the last send list entry is completed, the TxEOB interrupt is issued and the SPW TxVC Status Register[\$].ChTrig is disabled. Note that this applies even for zero-length send lists.

Note that there is no automatic generation of the CRC checksum for the header.

#### 10.6.6.4Status

Each virtual transmit channel, TxVC[\$], provides the following status monitoring:

- A *TxVC Send List Size Register* which indicates the number of remaining packets in the send list.
- A *TxVC Send List Pointer Register* which points to the current send list element.
- A *TxVC Status Register* indicating if the channel is triggered (started), and whether the channel is active (transfer ongoing).

#### 10.6.6.5Interrupts

Each TxVC, while transmitting, can issue interrupts as defined in SPW TxVC Interrupt Status Register [\$]. Further transmission is halted for the virtual channel.

### 10.6.7 Time-Codes

The SpaceWire Time-Code is used to support the distribution of system time across a SpaceWire network. The time-code interface is implemented by the SpaceWire CODEC and is specified in [CODEC]. The SpaceWire module adds a software interface for transmitting and receiving time-codes as well as qualifying the generation of an internal time reference pulse.

The time-code information is carried in a single byte. Six bits of time information are held in the least significant six bits of the time-code (T0-T5) and the two most significant bits (T6, T7) contain control flags that are distributed isochronously with the time-code.

#### 10.6.7.1Time-Code transmission

A time-code is transmitted over the SpaceWire link when the input signal *SpwTxTick* of the SpaceWire Module is activated (caused by an interrupt on the SpaceWire-RTC common interrupt bus). The time information transmitted is defined by the SPW Tx Time-Code Register.TxTimeCnt field and the SPW Tx Time-Code Register.TxCtrl field.

The SPW Tx Time-Code Register.TxTimeCnt is increased by one, modulo 64, before the new value is transmitted on the SpaceWire Link. It is possible to configure the start value of the time information (T0-T5) to be transmitted by writing to the SPW Tx Time-Code Register.TxTimeCnt field.

The SPW Tx Time-Code Register.TxCtrl field only reflects the status of the input signal *SpwTxTimeCtrl* of the SpaceWire Module, and contains the control flags (T6, T7).

The *TxTime-Code* interrupt is issued when the time-code is transmitted.

### 10.6.7.2Time-code reception

The received time-code can be read from the SPW Rx Time-Code Status Register, comprising the time information (T0-T5), and the control flags (T6, T7). The register is updated for every time-code received, independently of the value.

When time information that has been received with a value exactly one higher, modulo 64, than the previously received time information, which is locally stored in the SPW Rx Time-Code Register.RxCnt field, a pulse is generated on the SpaceWire Module output signal *SpwRxTick* which in turn is connected to the SpaceWire-RTC common interrupt bus, and the *RxTime-Code* interrupt is issued.

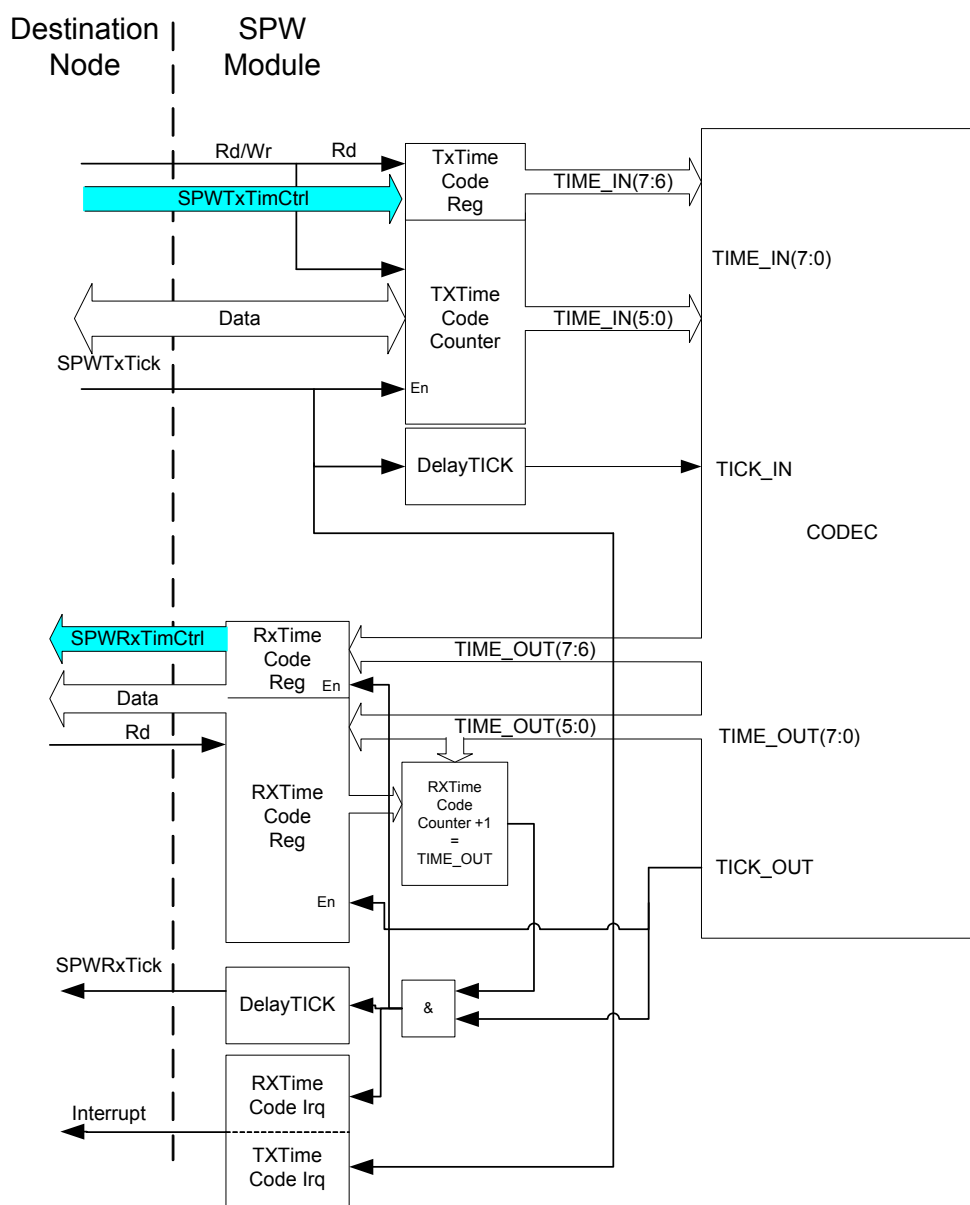


Figure 10-18.Time-Code handling block diagram

### 10.6.7.3 Alternative Time-code reception

The SPW2 core RxTimeTick signal is connected to the interrupt bus. The tick can thus be used to latch the counter values in the Timer core, using the above function. The time-code information can be readout through the SPW Receive Time-Code Status Register [SPW\_RTCSR] in the SPW2 core.

### 10.6.7.4 Alternative Time-code transmission

Time-code transmission can be performed in two ways: through software control, using the SPW SW Transmit Time-Code Register [SPW\_SWTTCR], through hardware control, via the interrupt bus, using the SPW Transmit Time-Code Mask Register [SPW\_TTCMR] and the SPW Transmit Time-Code Register [SPW\_TTCR].

The time-code information corresponding to the first of the above events occurring will be transmitted. Any subsequent event occurring before the time-code has been transmitted will be ignored. This also applies if the subsequent event is of the same type as the preceding one.

#### Alternative Time-code transmission through software control

The time-code is transmitted as soon as a write access is performed to the SPW SW Transmit Time-Code Register [SPW\_SWTTCR]. The SPW\_SWTTCR.TxTimeCtrl field defines the value of the corresponding time-code control flags (bits T6 and T7) to be transmitted.

#### Alternative Time-code transmission through hardware control

The time-code is transmitted when an interrupt occurs on the interrupt bus and the corresponding interrupt is not masked by the SPW Transmit Time-Code Mask Register [SPW\_TTCMR]. The TxTimeCtrl field of the SPW Transmit Time-Code Register [SPW\_TTCR] defines the value of the time-code control flags (bits T6 and T7) to be transmitted.

The mask register, SPW Transmit Time-Code Mask Register [SPW\_TTCMR], can be written and read via the APB bus. A time-code is sent only when an edge is detected on the filtered interrupt. Any interrupt in the system can be used for initiating time-code transmission. Note: The SPW2 core TxTimeTick input signal is generated from the interrupt bus.

The time-code information to be transmitted can be written and read via the APB bus through the SPW Transmit Time-Code Register [SPW\_TTCR].

Note: The SPW\_TTCR register has been augmented to allow the SPW\_TTCR.TxTimeCtrl field (i.e. bits 6 and 7) to be written as well. These bits can thus be controlled via the augmented register instead of being input via discrete inputs to the SPW2 core.

### 10.6.8 First Failing Packet Register

The SPW First Failing Packet Register is used for recording failure events during reception of SpaceWire packets. It is used for all types of transfer protocols and provides failure information that is either common to all transfer protocols or specific to a particular transfer protocol.

Most of the failure causes are detected during the Protocol Identification, the RMAP command header CRC verification, and other failure causes are detected during the data reception. As soon as a failure cause is detected, the remains of the packet are discarded. Note that only one unique error code is set per packet; if more than one error is found in a packet only the first detected one is recorded.

The SPW First Failing Packet Register is triggered, if cleared and ready, when a packet has been rejected and then records the three first bytes of the packet and the cause of the failure. Note that the three first bytes of every packet received, correct or not, are continuously saved, and further saving is stopped when a failure occurs.

If less than three bytes have been received, due to an EOP or EEP, the type of exit character will be stored after the data bytes (EOP=0, EEP=1), and the possible remaining byte in the SPW First Failing Packet Register will be cleared to zero and the *HeadErr* failure cause will be recorded.

The SPW First Failing Packet Register keeps its value after a detected failure until a read access has been performed to the register. After a read access the function is ready to once again capture on a failure event.

The SPW First Failing Packet Register contains the following information:

- Destination Logical Address (1:st received byte)
- Protocol Identifier (2:nd received byte)
- Protocol dependent information (3:rd received byte)
- Failure Type, which caused the trigger event (see below).

The contents of the 3:rd received byte is protocol dependent:

- For RMAP commands and RMAP responses, it comprises the Packet Type, Command and Source Path Address Length fields.
- For VCTP, it comprises the Dummy field.
- For unknown protocols, it comprises the contents of the 3:rd received byte.

Once the protocol identification has succeeded (i.e. the first bytes of a packet were correct), then erroneous VCTP data transfers or protocol handled by software (RMAP or unknown) do not trigger the SPW First Failing Register as hardware cannot distinguish where the header checksum is located or if a data CRC exists. Neither do EOP or EEP in these cases cause a triggering to occur.

Note that a disconnection of the SpaceWire Link does not trigger the SPW First Failing Packet Register.

### 10.6.9 SpaceWire CODEC

The SpaceWire (SPW2) Module is based on the University of Dundee SpaceWire CODEC. The CODEC implements the SpaceWire link protocol.

The SpaceWire CODEC is responsible for making a connection with the SpaceWire interface at the other end of a link and managing the flow of data across the link. The interface transmits and receives SpaceWire characters, which can be link characters (L-Chars) or normal characters (N-Chars). L-Chars are characters that are used to manage the flow of data across a link (NULL and FCT). N-Chars are the characters that are used to pass information across the link (data characters, EOP, EEP and time-codes).

The following subsections define the functional blocks that constitute the SpaceWire interface.

The SpaceWire CODEC is completely encapsulated by the SpaceWire Module, and all communication with the SpaceWire CODEC goes through registers and data structures defined for the SpaceWire Module.

Note that the TxFIFO and the RxFIFO are located outside the SpaceWire CODEC.

#### 10.6.9.1 Initialisation State Machine

The SpaceWire CODEC state machine is responsible for initiating a connection on the link and performing related synchronisation. The state machine determines its next state by monitoring receiver signals that indicate the type of characters received and any receiver errors. The state machine enables the transmitter to send NULL characters, FCT characters and N-char characters.



### 10.6.9.2Receiver

The receiver is responsible for decoding the received data-strobe encoded bit-stream into SpaceWire characters. The receiver reports the type of characters received and any errors encountered to the initialisation state machine. The receiver is implemented in two parts, a decoder that decodes characters in the receiver clock domain and a synchroniser that synchronises receiver signals to the *BusClk* clock domain. This method ensures all receiver outputs are synchronous to the *BusClk* clock.

### 10.6.9.3Receiver Credit Count

The receiver credit counter keeps a record of the buffer space available in the receiver RxFIFO and the number of characters that have been requested from the link. This allows the SpaceWire CODEC to implement flow control.

### 10.6.9.4Receiver Error Recovery

The receiver error recovery block recovers the receiver credit counters and the receiver RxFIFO after an error occurs. After an error an EEP marker is added to the receiver RxFIFO. The receiver credit counter must be updated because all previously transmitted FCT characters are discarded at the other end of the link due to error recovery.

### 10.6.9.5Transmitter

The transmitter is responsible for transmitting L-Chars and N-chars over a link using data-strobe encoding. The interface state machine determines which type of character the transmitter can send over the link. The transmitter accepts N-char characters and end of packet markers from the transmitter TxFIFO. N-char characters are transmitted when there is data in the TxFIFO and the transmitter has credit to send at least one more N-char. The transmitter sends an FCT character for each block of space for eight N-chars in the receive RxFIFO. As the host system reads out data from the receive RxFIFO the transmitter sends more FCT characters to indicate that the receiver has room to receive eight more N-chars. The transmitter sends NULL characters when there is no other information to send.

### 10.6.9.6Transmitter Clock Generator

The transmitter clock enable generator is responsible for generating the variable transmitter bit rate and default data signalling rate dependent on the configuration.

### 10.6.9.7Transmitter Credit Counter

The transmitter credit counter holds a count register that indicates the number of SpaceWire N-char characters that can be sent along the link.

### 10.6.9.8Transmitter Error Recovery

The transmitter error recovery module recovers the transmitter TxFIFO after an error occurs. In a network situation the first byte of a packet is interpreted as the packet address, therefore the error recovery block reads from the transmitter TxFIFO until the next end of packet marker is read from the TxFIFO. The transmitter is allowed to start up when error recovery is taking place but the transmitter is prevented from reading from the transmit TxFIFO until error recovery is complete.

### 10.6.9.9CODEC interconnections

The SpaceWire (SPW2) Module includes the University of Dundee SpaceWire CODEC. The CODEC is connected internally to the rest of the SpaceWire (SPW2) Module using buffer memory.

On the receiver side, four levels of buffering are used.

- The first FIFO, the RxFIFO, is directly controlled by the CODEC and has byte width. The SpaceWire Credit Count is defined using this RxFIFO.
- The second FIFO is a byte-to-word re-formatter used to transfer words between the *SpwClk* to *BusClk* clock domains, with room for 8 bytes. (Byte transfers would limit the throughput.) Also note that a transfer is initiated at packet end, even if a word has not been filled yet.

- The third buffer is used for storing RMAP command headers and four bytes of data cargo, allowing them to be verified, and is also used for other types of transfer, storing multiple words before transfer.
- The fourth buffer is used for allowing multiple words to be written to memory while still receiving more data from the previous FIFOs.

Incoming packets are inspected as their header reaches the end of the second level receiver FIFO. I.e., if the first packet to arrive is directed to an enabled but not triggered channel, it will remain in the first two FIFOs, blocking the following packets. And if the first packet to arrive is directed to a disabled or not implemented channel, it will be flushed out immediately, letting the next packets through.

On the transmitter side, three levels of buffering are used.

- The first buffer is used to allow multiple words to be read from memory before being transferred to the second FIFO.
- The second FIFO is a word-to-byte re-formatter used to transfer words between the
- *BusClk* and *SpwClk* clock domains.
- The third FIFO, the TxFIFO, is directly controlled by the CODEC and has byte width. The SpaceWire Credit Count is defined using this.

Note that there is additional protocol dependent data buffering on both the receiver and transmitter side before reaching the internal bus. The buffer size figures above should thus be regarded as estimates, and should never be used for computing exactly the amount of data located in various buffers at a given time.

## 10.6.10 Initialisation

### SpaceWire link configuration

The SpaceWire link is configured at reset to automatically start-up. It is possible to configure the link not to automatically start-up after a link disconnection. This is done with the [SPW CODEC Configuration Register](#).

#### 10.6.10.1 SpaceWire transmit Clock configuration

The SpaceWire link nominal data transfer rate is configured with the [SPW Clock Division Register](#). The default nominal transfer rate after reset matches the link start-up transfer data rate, which is configured with configuration inputs. After successful start-up, the link will use the nominal transfer data rate. The nominal data transfer rate can be changed during nominal operation. In the case of a link disconnection, the following link start-up will take place using the link start-up transfer data rate.

#### 10.6.10.2 SpaceWire logical address configuration

A SpaceWire node in a SpaceWire network has dedicated logical address when logical addressing is used. After a typical network start-up, all nodes have a default logical address of  $FE_{16}$ , and explicit path addressing is used to reach each node. The logical address can then be programmed for each node. When completed, path addressing is not longer required in the network and only logical addressing is used.

The SpaceWire (SPW2) Module logical address is stored in [SPW RxVC Config Register \[0\]](#). Note that although this register is dedicated to RxVC[0], the [Destination Logic Address](#) field is used for all communication to the SpaceWire (SPW2) Module.

#### 10.6.10.3 SpaceWire Virtual Transfer Protocol configuration

The SpaceWire (SPW2) Module implements the SpaceWire Virtual Channel Transfer Protocol (VCTP). The Protocol Identifier of this protocol is configured using the [SPW VC Transfer Protocol ID Register](#). The default reset value is such that it falls within the range of Protocol Identifiers available for general use as defined in [RMAPID].

Each virtual receive channel RxVC[\$] can be configured individually whether received packets should be stored contiguously in memory, or whether the end of each packet is to be padded up to the closest 32-bit word boundary. Padding is done with unknown arbitrary content. The configuration is done in the [SPW RxVC Config Register \[\\$\]](#).

#### 10.6.10.4 RMAP hardware support configuration

The SpaceWire (SPW2) Module implements hardware support for the reception and execution of Remote Memory Access Protocol (RMAP) commands. The hardware support is enabled and disabled using the [SPW RxVC Config Register \[0\]](#).

The RMAP protocol provides a Destination Key in the RMAP command header that must match that of the receiving node before the command is executed. The destination key for the command executed automatically in hardware is configured using the [SPW RMAP Destination Key Register](#). Note that the destination key is not checked by hardware for RMAP commands or responses that are forwarded to software for further processing.

#### 10.6.10.5 RMAP software support configuration

The SpaceWire (SPW2) Module implements software support for the reception of Remote Memory Access Protocol (RMAP) commands and header. The software support is enabled and disabled using the [SPW RxVC Config Register \[0\]](#).

The virtual receive channel RxVC[0] can be configured individually whether received packets should be stored contiguously in memory, or whether the end of each packet is to be padded up to the closest 32-bit word boundary. Padding is done with unknown arbitrary content. The configuration is done in the [SPW RxVC Config Register \[0\]](#).

#### 10.6.10.6Unknown protocol support configuration

The SpaceWire (SPW2) Module implements software support for the reception of unknown transfer protocols. The software support is enabled and disabled using the SPW RxVC Config Register [0]. The user can configure the SPW2 to interpret all incoming messages of four bytes or more as being of an unknown protocol by setting SPW RxVC Config Register [0].ForceUnk.

## 10.6.11 Operation/Usage

### 10.6.11.1 Link Abort handling

If SPW CODEC Status Register.LinkState makes a transition from the `Run` state, e.g. due to an Rx error, a credit count error, etc., this is called a link abort event, and the `LinkAbort` link interrupt is issued. The CODEC will at this point insert an EEP after the last received byte, and discard the remainder of the packet currently being transmitted. The TxVCs and RxVCs will retain their triggering status; i.e. once the link is up again the RxVCs will continue to receive packets (except for the eventual channel being untriggered due to the EEP in the truncated packet), and the TxVCs will only experience the link abort as a temporary link congestion. Note that if there was data left in the Rx FIFO when the link was disconnected, this will be handled by the RxVCs and written to memory while the link is still down.

Note that this 'pretend that nothing happened' handling of link abort is the one recommended by the standard. The host system should employ an acknowledge protocol or read back written values to ensure that a packet reached the recipient, and not assume that every transmitted packet reaches the recipient as long as the link stays up.

Note also that the FCT (Flow Control Token) needed for the other node of the link to go to the `Run` state will only be transmitted if there is room for at least 8 more bytes in the Rx FIFO. This means that if the Rx FIFO was flooded when the link abort occurred, no FCT will be transmitted from the flooded node. Thus the un-flooded node will go to `Connecting`, time-out, return to `ErrorReset` and try to connect again. In other words, a congested link will remain congested after a link abort and will fail to connect. This is the behaviour recommended by the standard. Note also that the flooded node might reach the `Run` state, since it can receive an FCT from the other node, but will go to `ErrorReset` when the other node disconnects.

### RxFifoFlush and TxFifoFlush usage

Some systems use legacy protocols where the reception order of packets is essential. In these systems there is a need to flush all Tx and also maybe Rx data when a link abort occurs, and (partially) restart the send-lists after reconnection. This can be accomplished by using the `TxFifoFlush` and `RxFifoFlush` SPW CODEC Configuration Register bits. Note that using this flush functionality is a deviation from the recommendation in the standard.

When `TxFifoFlush` is set, currently active Tx packets are flushed and all currently triggered Tx channels become un-triggered. Setting `RxFifoFlush` has the same effect on the Rx packets and RxVC channels. If any RxVC or TxVC (except `TxVC[0]`) channel was either active or triggered when the corresponding flush configuration bit was set, the corresponding `FlushAbort` channel interrupt is issued. As long as any `FlushAbort` interrupt is still pending, either in an SPW RxVC Interrupt Status Register [\$] or an SPW TxVC Interrupt Status Register [\$], no virtual channel RxVC, or TxVC (except `TxVC[0]`) can be (re-)triggered.

Note that this trigger blocking mechanism does not apply to hardware supported RMAP command handling; once the flush configuration bits are cleared, hardware supported commands will be handled and automatically generated responses will be transmitted even if there are still `FlushAbort` interrupts pending.

Since the `TxFifoFlush` and `RxFifoFlush` causes an indiscriminate discarding of data, it shall only be used when the link is disconnected, and should always continue until all data has been flushed from the system. This shall be accomplished in the following manner:

- Ensure that the link will not connect during the flushing by one of these methods:
  - Disconnect the link by setting SPW CODEC Configuration Register.LinkDis.or
  - Directly after initial connection clear the `LinkStart` and `AutoStart` bits in SPW CODEC Configuration Register, thus blocking automatic reconnection.
- Await the `LinkAbort` interrupt.
- Set the `TxFifoFlush` and/or `RxFifoFlush` bit in SPW CODEC Configuration Register. As a side effect, this will trigger the `FlushAbort` interrupts of the corresponding active or triggered channels.
- Wait until the flushed data pipe has been completely emptied. This is observable through the `TxParked` and/or `RxParked` bits in SPW CODEC Status Register.
- Clear the `TxFifoFlush`, `RxFifoFlush` and `LinkDis` bits, and set the `AutoStart` or `LinkStart` bit in SPW CODEC Configuration Register.

- Wait until the link is connected. This is observable through SPW CODEC Status Register.LinkState.
- Re-trig the relevant channels. Note that all FlushAbort interrupt status bits for both Rx and Tx must be cleared before any Rx or Tx channel can be triggered.

### 10.6.11.2 Virtual Receive Channels (RxVC)

The SpaceWire (SPW2) Module implements zero or more (not counting RxVC[0] which is always present) Virtual Receive Channels RxVC[\$] adhering to the SpaceWire Virtual Channel Transfer Protocol (VCTP).

To setup reception on RxVC[\$], the following steps need to be performed:

- Setup the word padding option and enable the channel using the SPW RxVC Config Register [\$].
- Setup up the location in memory for the buffer to which packets are to be received. The location is defined using the SPW RxVC DMA Page Register [\$] and the SPW RxVC DMA Base Address Register [\$].
- Setup optionally the maximum number of packets that should be received using the SPW RxVC Packet Counter Register [\$].
- Setup the size of the buffer, limiting the amount of data that should be received, using the SPW RxVC DMA Block Size Register [\$]. This also triggers the channel, which is now ready to receive packets.

The progress of the reception can be observed using the following registers:

- The address of the next data to be stored, relative to the SPW RxVC DMA Base Address Register [\$], can be observed using the SPW RxVC DMA Offset Register [\$].
- The start address of the currently received packet can be observed using the SPW RxVC Current Packet Status Register [\$].
- The number of packets received since the channel was last trigger can be observed using the SPW RxVC Status Register [\$].
- SPW RxVC Status Register [\$] can also be used for observing if the channel is triggered and if the channel is active receiving a message.

Note that if ChAct is set but not ChTrig, then there is a packet in the Rx FIFOs waiting to be handled. The decision to handle this packet has already been made, so if the RxVC is disabled at this point, the packet will remain in the Rx FIFOs until the channel is triggered, thus blocking the reception of the next packets.

The overall progress of reception can be observed using the SPW Status Register, which indicates which channel is active receiving a message.

During nominal reception, interrupts are issued to indicate that the pre-set number of packets has been received (CntTrig interrupt), or that the allocated memory buffer has been filled (RxEOB interrupt). If an error is detected which un-triggers the channel, then this is reported by issuing an error interrupt (CRCDataErr, FlushAbort, DmaWrErr and RxEEP interrupts). This information can be observed using the SPW RxVC Interrupt Status Register [\$].

When reception has been halted due to an error (e.g. DmaWrErr), SPW RxVC Status Register [\$].PktCnt will indicate the number of received error free packets and SPW RxVC Current Packet Status Register [\$] will point to the start of the faulty packet. Note that due to system behaviour, DmaWrErr may be detected after the SPW RxVC DMA Offset Register [\$] has been incremented. The current offset decremented by one will point to the packet with the faulty access, but not necessarily to the faulty address.

The progress of the reception can also be observed using the SPW First Failing Packet Register, indicating if an error has occurred in the first four bytes of a VCTP.

When a packet is stored to memory, the first four bytes of the received packet are stripped, removing the protocol header with the virtual channel routing information.

To disable or prematurely un-trigger a virtual receive channel do the following:

- Clear SPW RxVC Config Register [\$].RxEn.
- Check SPW RxVC Status Register [\$].ChAct to see if the channel had started to receive a packet before being disabled. If that is the case wait (and if necessary re-trigger the channel) until the entire packet has been received.

To disable or prematurely un-trigger a virtual receive channel in mid-packet, the complete SpaceWire link needs to be disconnected and the Rx flushed using the [SPW CODEC Configuration Register](#).

### 10.6.11.3 Virtual Transmit Channels (TxVC)

The SpaceWire (SPW2) Module implements zero or more (not counting TxVC[0] which is always present) Virtual Transmit Channels TxVC[\$].

To setup transmission on TxVC[\$], the following steps need to be performed:

- Create a send list containing send list entries, each defining the size and length of a header and a data field.
- Setup the channel's pointer to point to the location of the first send list entry. This is done using the [SPW TxVC SendList Pointer Register \[\\$\]](#).
- Setup the size of the send list, defining the number of send list entries. This is done using the [SPW TxVC SendList Size Register \[\\$\]](#). This also triggers the channel, which is now ready to transmit packets.

The progress of the transmission can be observed using the following registers:

- The address to the send list entry of the latest successfully transmitted packet can be observed using the [SPW TxVC SendList Pointer Register \[\\$\]](#).
- The number of remaining send list entries in the send list can be observed using the [W TxVC SendList Size Register \[\\$\]](#).
- [SPW TxVC Status Register \[\\$\]](#) can also be used for observing if the channel is triggered and if the channel is active transmitting a message.

The overall progress of transmission can be observed using the [SPW Status Register](#), which indicates which channel is active transmitting a message.

During nominal transmission, interrupts are issued to indicate that a send list has been completed (TxEOB interrupt). If an error is detected which un-triggers the channel, then this is reported by issuing an error interrupt (FlushAbort and DmaRdErr interrupts). This information can be observed using the [SPW TxVC Interrupt Status Register \[\\$\]](#).

To disable or prematurely un-trigger a virtual transmit channel, the complete SpaceWire link needs to be disconnected and the Tx flushed using the [SPW CODEC Configuration Register](#).

### 10.6.11.4 RMAP command reception in hardware

Reception of RMAP commands that are executed in hardware does not require any software support, more than the enabling of this support, as described in 10.6.10.4.

The progress of the reception of RMAP commands can be observed using the [SPW First Failing Packet Register](#), indicating if an error has occurred. This is also indicated by the issuing of the PktRej link interrupt.

For hardware supported RMAP commands, responses are generated and transmitted automatically, when required by the protocol. New packets of all protocol types can be received and handled in parallel with the response transmission. The only limitation is that the automatic response generation mechanism can only handle one response at a time. Thus, if a second RMAP that requires an automatic response is received while an older automatic response is being transmitted, then the reception mechanism will stall until the older response is finished. In other words, if multiple hardware supported RMAP commands that generate automatic responses are received in quick succession, this might cause congestion on the Rx.

### 10.6.11.5 RMAP command and response reception in software

When transfer protocol support is enabled for RxVC[0], using the [SPW RxVC Config Register \[0\]](#), any RMAP command not handled by hardware and any RMAP response will be stored in the memory buffer allocated to this channel.

The reception handling is similar to what has been described for the Virtual Receive Channels (RxVC). The only differences are that the first four bytes of a received packet are not deleted and that the last byte in a packet containing the RMAP data CRC checksum is deleted before the packet is saved to memory.

As an additional service, an interrupt is issued and the channel is un-triggered if the combined CRC as calculated over the RMAP header, the RMAP header CRC checksum byte and the RMAP data field does not match the received data CRC checksum. This information can be observed using the [SPW RxVC Interrupt Status Register \[0\]](#).

### 10.6.11.6 RMAP command and response transmission from software

Any virtual transmit channel TxVC[\$] can be used to transmit RMAP commands or responses from software. The transmit handling is similar to what has been described for the Virtual Transmit Channels (TxVC). The only difference is that an additional service is provided which calculates the RMAP data CRC checksum byte and adds it to the end of the data field before adding the EOP. This service is enabled in the send list entry. Note that the RMAP header CRC checksum byte must be calculated by software.

### 10.6.11.7 Reception of unknown protocols in software

When transfer protocol support is enabled for RxVC[0], using the [SPW RxVC Config Register \[0\]](#), any unknown transfer protocols will be stored in the memory buffer allocated to this channel. Note that the identification of an unknown protocol is made either by the Protocol ID byte, or forcedly for all packets of four bytes or more by setting the [SPW RxVC Config Register \[0\]](#), [ForceUnk](#) bit.

The reception handling is similar to what has been described for the Virtual Receive Channels (RxVC). The only difference is that the first four bytes of a received packet are not deleted before the packet is saved to memory.

### 10.6.11.8 Time-Codes

#### Time-Code transmission

Time-code transmission can be triggered either by the internal *SpwTxTick* signal which is asserted by an interrupt on the common SpaceWire-RTC interrupt bus not being masked by the [SPW Transmit Time-Code Mask Register](#) (hardware triggered time-code), or by writing the [SPW SW Transmit Time-Code Register](#) (software triggered time-code). When triggered by *SpwTxTick* the control part of the time-code is defined by the [SPW Transmit Time-Code Register](#); and when triggered by a register write it is defined by the register write data. The counter part of the time-code is defined using the [SPW Transmit Time-Code Register](#), which when read yields the last transmitted time-code (both control and counter part). The TxTime-Code link interrupt is issued when a time-code is transmitted.

#### Time-Code reception

Time-codes are received using the [SPW Receive Time-Code Status Register](#). The RxTime-Code link interrupt is issued when a time-code has been received; and also the internal *SpwRxTick* signal is asserted, which in turn is connected to the SpaceWire-RTC common interrupt bus.



## 10.6.12 Error Handling

### 10.6.12.1 CODEC Status

The overall status of the SpaceWire codec can be observed via the SPW CODEC Status Register. Note that all fields except LinkState are included for debug purposes only, and should not be used in normal operation. Error events are captured by the corresponding link interrupts.

### 10.6.12.2 First Failing Packet

The First Failing Packet Register indicates failures that can occur during reception of a packet. When an error has occurred, the register is not updated due to new errors until the register has been read. In addition to an error code identifying the cause for the failure, the register also contains the first three received bytes received in a packet.

## **10.6.13 Usage Constraints**

### **10.6.13.1 Functional**

None. TBC

### **10.6.13.2 Timing**

None. TBC

### **10.6.13.3 Examples**

None. TBC

## 10.6.14 Interrupt Handling

The interrupt handling in the SpaceWire (SPW2) Module is implemented in two layers.

The top layer contains the SPW Pending Interrupt Masked Status Register, the

SPW Pending Interrupt Status Register and the SPW Interrupt Mask Register. These register are used for masking or propagating the interrupts from the lower layer.

The lower layer contains three groups of interrupt registers:

- One SpaceWire Link group
- One or more multiple Virtual Receive Channel RxVC[\$] groups
- One or more multiple Virtual Transmit Channel TxVC[\$] groups

Each lower group contains an Interrupt Status Register, an Interrupt Status Set Register and an Interrupt Status Clear Register. These register are used for observing, setting and clearing the interrupts belonging to the group. When an interrupt is issued, it is propagated to the top layer set of register for further propagation outside the module.

The interrupt registers give complete freedom to the software, by providing means to mask interrupts, clear interrupts, force interrupts and read interrupt status.

All of the interrupt registers are listed in Table 10-1 Interrupt Register Summary, along with the effect of reading and writing them.

Register	Acronym	Read	Write
<u>SPW Pending Interrupt Masked Status Register</u>	SPW_PIMSR	Reads PISR and IMR	-
<u>SPW Pending Interrupt Status Register</u>	SPW_PISR	Reads	-
<u>SPW Link Interrupt Status Register</u>	SPW_LISR	corresponding	
<u>SPW RxVC Interrupt Status Register[\$]</u>	SPW_RxISR[\$]	ISR	
<u>SPW TxVC Interrupt Status Register[\$]</u>	SPW_TxISR[\$]		
<u>SPW Interrupt Mask Register</u>	SPW_IMR	Reads IMR	Writes IMR
<u>SPW Link Interrupt Status Set Register</u>	SPW_LISSR	-	Sets selected bits in
<u>SPW RxVC Interrupt Status Set Register[\$]</u>	SPW_RxISSR[\$]		corresponding ISR
<u>SPW TxVC Interrupt Status Set Register[\$]</u>	SPW_TxISSR[\$]		
<u>SPW Link Interrupt Clear Register</u>	SPW_LISCR	-	Clears selected bits
<u>SPW RxVC Interrupt Status Clear Register[\$]</u>	SPW_RxISCR[\$]		in corresponding
<u>SPW TxVC Interrupt Status Clear Register[\$]</u>	SPW_TxISCR[\$]		ISR

**Table 10-1. Interrupt Register Summary**

When an interrupt occurs the corresponding bit in the Interrupt Status Register is set. The normal sequence to initialise and handle a module interrupt is:

- Set up the software interrupt-handler to accept an interrupt from the module.
- Read the SPW Pending Interrupt Status Register and clear any spurious interrupts by writing the corresponding Interrupt Status Clear Register.
- Initialise the SPW Interrupt Mask Register, unmasking each bit that should generate the module interrupt.
- When an interrupt occurs, read the SPW Pending Interrupt Status Register in the software interrupt-handler to determine the causes of the interrupt.
- Handle the interrupt, taking into account all causes of the interrupt.
- Clear the handled interrupt using the corresponding Interrupt Status Clear Register.

*Masking interrupts:* After reset, all interrupt bits are masked, since the SPW Interrupt Mask Register is zero. To enable generation of a module interrupt for an interrupt bit, set the corresponding bit in the SPW Interrupt Mask Register.

*Clearing interrupts:* Selected bits can be cleared by writing ones to the bits that shall be cleared to the corresponding Interrupt Status Clear Register.

*Forcing interrupts:* When an Interrupt Status Set Register is written, the resulting value is the original contents of the corresponding Interrupt Status Register logically OR-ed with the write data. This means that writing the register can force (set) an interrupt bit, but never clear it.

*Reading interrupt status:* Reading the Interrupt Status Register yields the data without clearing the contents.

*Reading interrupt status of unmasked bits:* Reading the SPW Pending Interrupt Masked Status Register yields the contents of the SPW Pending Interrupt Status Register masked with the contents of the SPW Interrupt Mask Register, without clearing the contents.

### **10.6.15 Interrupt bus**

In the SpaceWire-RTC, all interrupts from the peripheral units, such as CAN and SPW, are routed through a 32-bit wide interrupt bus. This bus is an input and an output to all cores in the design. The bus is also connected to the secondary interrupt controller in the LEON2FT core.

The interrupt bus is used for example in the Timer core, which takes the full bus and combines it with a local mask register to form a latch signal. This latch signal is used to latch the timer values on the occurrence of the filtered interrupt. Thus any interrupt in the system (excluding specific LEON2FT peripheral interrupts only connected to the primary interrupt controller) can be used to latch the timers.

## 10.7 Register definition summary

This chapter contains all commands and registers available for the SpaceWire (SPW2) Module.

The registers of the SpaceWire (SPW2) Module are addressed according to the tables below. The external address of each unique register is its register address added to the external base address of the module, or to the external address of each of the DMA channels, which is implementation specific.

Unused register bits are zero on register reads and don't care on register writes

Register byte address:	Register name:	Acronym:
0000_0000 <sub>16</sub>	<u>SPW Pending Interrupt Masked Status Register</u>	[SPW_PIMSR]
0000_0008 <sub>16</sub>	<u>SPW Pending Interrupt Status Register</u>	[SPW_PISR]
0000_0010 <sub>16</sub>	<u>SPW Interrupt Mask Register</u>	[SPW_IMR]
0000_0014 <sub>16</sub>	<u>SPW Link Interrupt Status Register</u>	[SPW_LISR]
0000_0018 <sub>16</sub>	<u>SPW Link Interrupt Status Set Register</u>	[SPW_LISSR]
0000_001C <sub>16</sub>	<u>SPW Link Interrupt Status Clear Register</u>	[SPW_LISCR]
0000_0020 <sub>16</sub>	<u>SPW CODEC Configuration Register</u>	[SPW_CCR]
0000_0024 <sub>16</sub>	<u>SPW Clock Division Register</u>	[SPW_CDR]
0000_0028 <sub>16</sub>	<u>SPW RMAP Destination Key Register</u>	[SPW_RDKR]
0000_002C <sub>16</sub>	<u>SPW Transmit Time-Code Register</u>	[SPW_TTCR]
0000_0030 <sub>16</sub>	<u>SPW VC Transfer Protocol ID Register</u>	[SPW_VCTPIDR]
0000_0034 <sub>16</sub>	<u>SPW SW Transmit Time-Code Register</u>	[SPW_SWTTCR]
0000_0040 <sub>16</sub>	<u>SPW Status Register</u>	[SPW_SR]
0000_0044 <sub>16</sub>	<u>SPW CODEC Status Register</u>	[SPW_CSR]
0000_0048 <sub>16</sub>	<u>SPW Receive Time-Code Status Register</u>	[SPW_RTCSR]
0000_0080 <sub>16</sub>	<u>SPW First Failing Packet Register</u>	[SPW_FFPR]
0000_0MN0 <sub>16</sub> MN = 16 + 4*\$	<u>SPW RxVC Config Register [\$]</u> 0 ≤ \$ ≤ RxVC	[SPW_RxCnf[\$]]
0000_0MN4 <sub>16</sub> MN = 16 + 4*\$	<u>SPW RxVC Packet Counter Register [\$]</u> 0 ≤ \$ ≤ RxVC	[SPW_RxPR[\$]]
0000_0MN8 <sub>16</sub> MN = 16 + 4*\$	<u>SPW RxVC DMA Page Register [\$]</u> 0 ≤ \$ ≤ RxVC	[SPW_RxDPage[\$]]
0000_0MNC <sub>16</sub> MN = 16 + 4*\$	<u>SPW RxVC DMA Base Address Register [\$]</u> 0 ≤ \$ ≤ RxVC	[SPW_RxDBAR[\$]]
0000_0MN0 <sub>16</sub> MN = 16 + 4*\$+1	<u>SPW RxVC DMA Block Size Register [\$]</u> 0 ≤ \$ ≤ RxVC	[SPW_RxDBSR[\$]]
0000_0MN4 <sub>16</sub> MN = 16 + 4*\$+1	<u>SPW RxVC DMA Offset Register [\$]</u> 0 ≤ \$ ≤ RxVC	[SPW_RxDOR[\$]]
0000_0MN8 <sub>16</sub> MN = 16 + 4*\$+1	<u>SPW RxVC Status Register [\$]</u> 0 ≤ \$ ≤ RxVC	[SPW_RxSR[\$]]
0000_0MNC <sub>16</sub> MN = 16 + 4*\$+1	<u>SPW RxVC Current Packet Status Register [\$]</u> 0 ≤ \$ ≤ RxVC	[SPW_RxCPSR[\$]]
0000_0MN0 <sub>16</sub> MN = 16 + 4*\$+2	<u>SPW RxVC Interrupt Status Register [\$]</u> 0 ≤ \$ ≤ RxVC	[SPW_RxISR[\$]]
0000_0MN4 <sub>16</sub> MN = 16 + 4*\$+2	<u>SPW RxVC Interrupt Status Set Register [\$]</u> 0 ≤ \$ ≤ RxVC	[SPW_RxISSR[\$]]
0000_0MN8 <sub>16</sub> MN = 16 + 4*\$+2	<u>SPW RxVC Interrupt Status Clear Register [\$]</u> 0 ≤ \$ ≤ RxVC	[SPW_RxISCR[\$]]
0000_03N0 <sub>16</sub> N = 2*\$	<u>SPW TxVC SendList Pointer Register [\$]</u> 0 ≤ \$ ≤ TxVC	[SPW_TxSLPR[\$]]

0000_03N4 <sub>16</sub> N = 2*\$	<u>SPW TxVC SendList Size Register [\$]</u> 0 ≤ \$ ≤ TxVC	[SPW_TxSLSR[\$]]
0000_03N8 <sub>16</sub> N = 2*\$	<u>SPW TxVC Status Register [\$]</u> 0 ≤ \$ ≤ TxVC	[SPW_TxSR[\$]]
0000_03NC <sub>16</sub> N = 2*\$	<u>SPW TxVC Interrupt Status Register [\$]</u> 0 ≤ \$ ≤ TxVC	[SPW_TxISR[\$]]
0000_03N0 <sub>16</sub> N = 2*\$+1	<u>SPW TxVC Interrupt Status Set Register [\$]</u> 0 ≤ \$ ≤ TxVC	[SPW_TxISSR[\$]]
0000_03N4 <sub>16</sub> N = 2*\$+1	<u>SPW TxVC Interrupt Status Clear Register [\$]</u> 0 ≤ \$ ≤ TxVC	[SPW_TxISCR[\$]]

## 10.7.1 SpaceWire (SPW2) Module Registers

### 10.7.1.1 Interrupt registers

**SPW Pending Interrupt Masked Status Register [SPW\_PIMSR]**

**RM**

**SPW Pending Interrupt Status Register [SPW\_PISR]**

**R**

**SPW Interrupt Mask Register [SPW\_IMR]**

**R/W**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Rx7	Rx6	Rx5	Rx4	Rx3	Rx2	Rx1	Rx0	Tx7	Tx6	Tx5	Tx4	Tx3	Tx2	Tx1	Tx0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MSB															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-	Link Abort	Pkt Rej	Tx Time-Code	Rx Time-Code	Cr Err	ESC Err	Par Err	Diss Err
								0	0	0	0	0	0	0	0
LSB															

**Function:** Note that the individual bits in the register group are set and cleared using the corresponding SPW\_LISSR and SPW\_LISCR, SPW\_RxISSR[\$] and SPW\_RxISCR[\$], or SPW\_TxISSR[\$] and SPW\_TxISCR[\$] registers.

**Timing:**

**Constraints:**

<i>Field:</i>	<i>Description:</i>
DisErr	Link interface disconnection error detected
ParErr	Link interface parity error detected
ESCErr	Link interface ESC error detected
CrErr	Link interface credit error detected
RxTime-Code	Time-code received
TxTime-Code	Time-code transmitted
PktRej	A packet has been rejected and discarded, First Failing Packet Register has been triggered
LinkAbort	SPW CODEC Status Register.LinkState has made a transition from Run to Error Reset
Tx\$	TxVC \$ interrupt, see SPW TxVC interrupt registers for details
Rx\$	RxVC \$ interrupt, see SPW RxVC interrupt registers for details

**SPW Link Interrupt Status Register [SPW\_LISR]** **R**  
**SPW Link Interrupt Status Set Register [SPW\_LISSR]** **S**  
**SPW Link Interrupt Status Clear Register [SPW\_LISCR]** **C**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
MSB															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-	Link Abort	Pkt Rej	Tx Time- Code	Rx Time- Code	Cr Err	ESC Err	Par Err	Diss Err
								0	0	0	0	0	0	0	0
LSB															

**Function:** Contains the interrupts related to the link itself.  
**Timing:** All the link specific interrupts are issued as soon as detected by the SpaceWire codec. Note that LinkAbort is issued together with any link specific interrupt if the error occurred while SPW CODEC Status Register.LinkState was in Run state. Note also that when the user disconnects the link by writing the SPW CODEC Configuration Register.LinkDis, the resulting LinkAbort interrupt will be issued up to 2  $\mu$ s later.

All the time-code specific interrupts are issued on either the successful reception or transmission of the time-code.

The PktRej interrupt is issued either directly when an error is detected during protocol identification (i.e. discarded without writing to memory), or when the last byte has been written to memory.

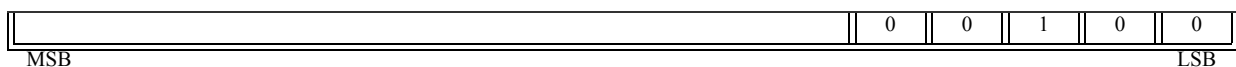
**Constraints:**

<i>Field:</i>	<i>Description:</i>
DisErr	Link interface disconnection error detected
ParErr	Link interface parity error detected
ESCErr	Link interface ESC error detected
CrErr	Link interface credit error detected
RxTime-Code	Time-code received
TxTime-Code	Time-code transmitted
PktRej	A packet has been rejected and been discarded, First Failing Packet Register has been triggered
LinkAbort	<u>SPW CODEC Status Register.LinkState</u> has made a transition from Run to Error Reset

### 10.7.1.2 Configuration registers

**SPW CODEC Configuration Register [SPW\_CCR]** **R/W**

31						5	4	3	2	1	0
							Tx Fifo Flush	Rx Fifo Flush	Auto Start	Link Dis	Link Start



**Function:** This register configures the start-up behaviour of the SpaceWire CODEC, controlling the Link Interface Control State Machine.

The link can be disconnected during normal operation by setting the LinkDis bit.

**Timing:**

**Constraints:** Note that one node on the link must be configured to LinkStart for NULL tokens to start to flow, and to thus get a link connection. The other node should be configured to AutoStart, lest the nodes send NULL tokens while the opposite node is in ErrorReset or ErrorWait.

The RxFifoFlush and TxFifoFlush bits shall only be used when the link is not in the Run state. To ensure this the user shall use one of the methods described in

<i>Field:</i>	<i>Value:</i>	<i>Description:</i>
LinkStart	0	SpaceWire link cannot proceed to Started state unless AutoStart is set.
	1	SpaceWire link can proceed to Started state, if the link has the LinkDis bit cleared.
LinkDis	0	SpaceWire link is enabled
	1	SpaceWire link is disabled, i.e. the LICSM proceeds directly to the ErrorReset state when reaching the Run state.
AutoStart	0	No Auto start of the CODEC.
	1	Auto start the CODEC, provided that the LinkDis bit is cleared. The CODEC will wait in state Ready until the first NULL character is received.
RxFifoFlush	0	Rx flush mechanism off.
	1	All data in Rx pipe (FIFOs, etc.) is flushed and all active and/or triggered RxVC are aborted.
TxFifoFlush	0	Tx flush mechanism off.
	1	All data in Tx pipe (FIFOs, etc.) is flushed and all active and/or triggered TxVC are aborted.



**SPW Clock Division Register [SPW\_CDR]****R/W**

31	6	5	0
-			TxNomDiv
			XX <sub>16</sub>
MSB			LSB

**Function:** This register configures the transmitter baud rate.**Timing:** The data rate during a link connection is the link start-up rate.**Constraints:**

Field:	Description:
TxNomDiv	<p>The nominal transmitter bit rate is <math>SpwClk / (0.5 * (TxNomDiv + 1))</math>. (Double Data Rate)</p> <p>The reset value XX of the field matches the link start-up data rate as configured with the configuration inputs <i>SpwClk10MBit</i> and <i>SpwClkMul</i>. The reset value will be <math>TxNomDiv = (2 / (\text{factor}_{SpwClk10MBit} * \text{factor}_{SpwClkMul})) - 1</math>.</p>

**SPW RMAP Destination Key Register [SPW\_RDKR]****R/W**

31	8	7	0
-			DestKey
			0
MSB			LSB

**Function:** This register configures the RMAP Destination Key.**Timing:****Constraints:**

Field:	Description:
DestKey	RMAP Destination key. Any incoming hardware supported RMAP command must have matching Destination Key in order to be accepted. If not, the complete packet is rejected during header check.

**SPW Transmit Time-Code Register [SPW\_TTCR]****R/W**

31	8	7	6	5	0
-			TxTimeCtrl	TxTimeCnt	
			-	0	
MSB				LSB	

**Function:** This registers is used for initiating the Time-Code transmission counter.**Timing:** Writes are ignored while a time-code transmission is taking place. Written values should thus be validated by a read.**Constraints:** Values written to the TxTimeCtrl field are only visible if followed by an assertion of the *SpwTxTick*.

Field:	Description:
--------	--------------

TxTimeCnt	New TxTimeCnt start value, (TxTimeCnt + 1) will be transmitted at the next <i>SpwTxTick</i> .
TxTimeCtrl	When read: TxTimeCtrl value transmitted due to a previous <i>SPWTxTick</i> or SPW_SWTTCR write. When written: TxTimeCtrl value to be transmitted

### **SPW VC Transfer Protocol ID Register [SPW\_VCTPIDR]**

**R/W**

31	8	7	0
-			VCTPID
			F0 <sub>16</sub>
MSB			LSB

**Function:** This register configures the SpaceWire Virtual Channel Transfer Protocol Identifier.

**Timing:**

**Constraints:** See [RMAPID] for constraints on Protocol ID usage. Note that RMAP in the SpaceWire (SPW2) Module has a Protocol ID value of 01<sub>16</sub> and this is thus not a valid value for VCTPID.

<b>Field:</b>	<b>Description:</b>
VCTPID	Configuration of SpaceWire Transfer protocol ID for the VCTP protocol.

### **SPW SW Transmit Time-Code Register [SPW\_SWTTCR]**

**W**

31	8	7	6	5	0
-			TxTimeCtrl	-	
			-		
MSB				LSB	

**Function:** This registers is used for triggering a Time-Code transmission.

**Timing:** Writes are ignored while a time-code transmission is taking place. A minimum separation of 10  $\mu$ s between writes is recommended.

**Constraints:**

<i>Field:</i>	<i>Description:</i>
TxTimeCtrl	New TxTimeCtrl value to be transmitted when written.

### **SPW Transmit Time-Code Mask Register [SPW\_TTCMR]**

**R/W**

31	0
Select	
0	
MSB	LSB

**Function:** This registers is used for selecting interrupts for Time-Code transmission.

**Timing:**

**Constraints:**

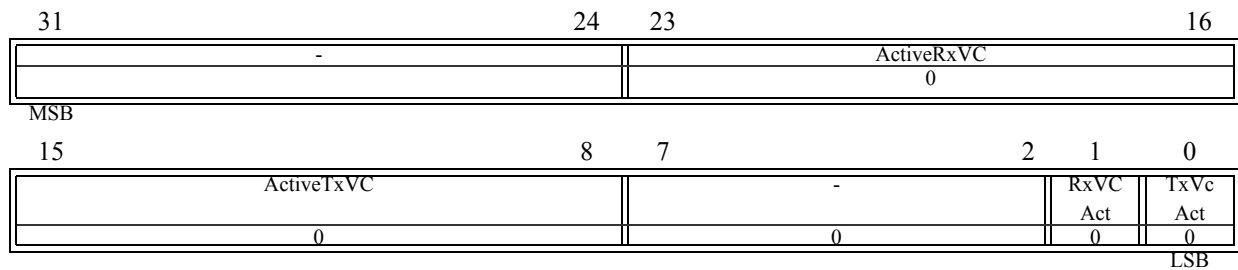
<i>Field:</i>	<i>Description:</i>
---------------	---------------------

Select	Specifies what bits of the AMBA APB interrupt bus shall cause a tick on the SpwTxTick input to the SpaceWire module.
--------	--

### 10.7.1.3 Status register

#### **SPW Status Register [SPW\_SR]**

**R**



**Function:** This register provides status information on active virtual receive and transmit channels.

**Timing:**

**Constraints:** The ActiveTxVC and ActiveRxVC values are limited by the number of implemented virtual channels. An ActiveTxVC value of 0 indicates that the hardware supported RMAP is transmitting an RMAP response. An ActiveRxVC value of 0 indicates that a command or reply is received for further software processing.

<i>Field:</i>	<i>Value:</i>	<i>Description:</i>
TxVCAct	0	All TxVCs are inactive
	1	A TxVC is active. This bit is set when any TxVC[\$] becomes active.
RxVCAct	0	All RxVCs are inactive
	1	An RxVC is active. This bit is set when any RxVC[\$] becomes active.
ActiveTxVC	0 - 7	The last selected TxVC. This field is set to \$, indicating which virtual channel TxVC[\$] is the active one. The value is only valid while the <u>TxVCAct</u> bit is set.
ActiveRxVC	0 - 7	The last selected RxVC. This field is set to \$, indicating which virtual channel RxVC[\$] is the active one. The value is only valid while the <u>RxVCAct</u> bit is set.

### 10.7.1.4CODEC Status register

#### SPW CODEC Status Register [SPW\_CSR]

R															
31															16
												Tx Parked	Rx Parked	Tx Empty	Rx Empty
												1	1	1	1
MSB															
15	14	13	12	11	10	9	8	7	5	4	3	2	1	0	
Seq Err Time	Seq ErrN Char	TxCr ed 1	Got Time	Got N Char	Got FCT	Got Null	Inf Run	LinkState		Tx Cred Err	Rx Cred Err	Esc Err	Par Err	Dis Err	
0	0	0	0	0	0	0	0	0		0	0	0	0	0	
LSB															

**Function:** This register provides status information on the SpaceWire CODEC.

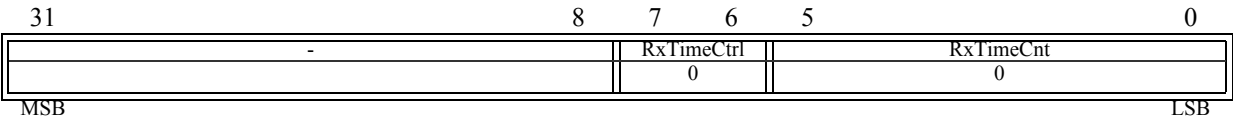
**Timing:**

**Constraints:** All fields of this register, except LinkState, RxParked and TxParked, are included for test and debug purposes only.

Field:	Value:	Description:
DisErr	1	Disconnect error status
ParErr	1	Parity error status
EscErr	1	Escape error status
RxCredErr	1	Receiver credit error status: The receiver has received more data characters then requested
TxCredErr	1	Transmitter credit error status: The transmitter has credit to send more than the allowed 56 data characters.
LinkState	000 <sub>2</sub>	CODEC link state machine in ErrorReset state
	001 <sub>2</sub>	CODEC link state machine in ErrorWait state
	010 <sub>2</sub>	CODEC link state machine in Ready state
	011 <sub>2</sub>	CODEC link state machine in Started state
	100 <sub>2</sub>	CODEC link state machine in Connecting state
	101 <sub>2</sub>	CODEC link state machine in Run state
InfRun	1	Interface state machine is in the Run state.
GotNull	1	Receiver got NULL. Remains asserted after first NULL.
GotFCT	1	Receiver got FCT. Remains asserted after first FCT
GotNChar	1	Receiver got N-chars. Remains asserted after first N-Char
GotTime	1	Receiver got Time-codes. Remains asserted after first Time-code
TxCred1	1	Transmitter has credit to send at least one more data character
SeqErrNChar	1	N-char sequence error (N-char received before link state is Run)
SeqErrTime	1	Time-code sequence error (Time-code received before link state is Run)
RxEmpty	0	The receive buffer is not empty
	1	The receive buffer is empty
TxEmpty	0	The transmit buffer is not empty
	1	The transmit buffer is empty
RxParked	0	Rx not parked
	1	No data in Rx pipe, all reception mechanisms idle and link is disconnected
TxParked	0	Tx not parked
	1	No data in Tx pipe, all transmission mechanisms idle and link is disconnected

**SPW Receive Time-Code Status Register [SPW\_RTCSR]**

**R**



**Function:**

This register is used for Time-Code reception.

**Timing:**

Due to module implementation limitations a time-code received less than 1.4  $\mu$ s after a previous one may be lost.

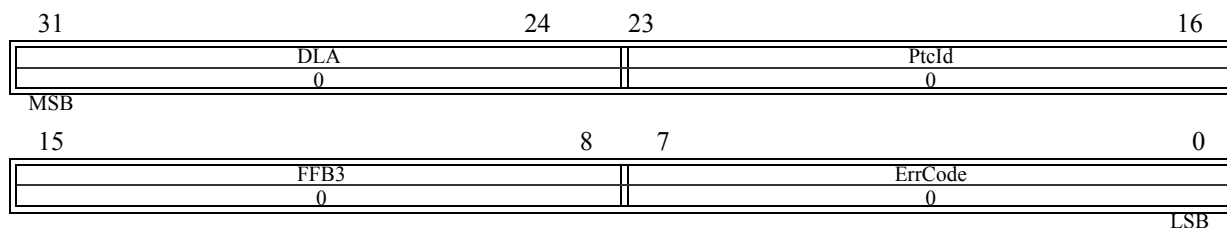
**Constraints:**

Field:	Description:
RxTimeCnt	The last received time-code, i.e. time information.
RxTimeCtrl	The last received time-ctrl, i.e. control flags.

### 10.7.1.5 Other registers

#### **SPW First Failing Packet Register [SPW\_FFPR]**

**RC**



- Function:** This register holds diagnostic information related to the reception and identification of a packet.
- Timing:** DLA, PtcId and FFB3 are, as long as no errors have been detected, updated for every subsequent packet. The register is frozen when an error is detected in the four first bytes (i.e. rejected during the identification process, e.g. VCTP to a disabled channel) of a packet or in a hardware supported RMAP command. The register is not updated for any new errors, until after being read.

#### **Constraints:**

Field:	Value:	Description:	
ErrCode	0	No Error	No error detected since the last register read access
	1	DestErr	Error while DMA accessing the internal bus, e.g. illegal address.
	2	CmdErr	Unused RMAP command according to [RMAP]
	3	DKeyErr:	Destination Key error, for RMAP commands supported by hardware
	4	DCRCErr	Data CRC error, for RMAP commands supported by hardware.
			Combined header and data CRC error, for RMAP commands supported by software.
	5	EEOP	Early EOP in data for RMAP commands supported by hardware, i.e. EOP has been received with less data than expected from the RMAP command header.
	6	CTL	Cargo too large. Late EOP or EEP in data for RMAP commands supported by hardware, i.e. EOP/EEP has been received with more data than expected from the RMAP command header.
	7	EEEP	Early EEP in data for RMAP commands. For RMAP commands supported in hardware, EEP has been received with less data or exactly as much as expected from the RMAP command header.
	9	VBOvrR	Verify Buffer Over-Run. A verified write of more than 4 bytes was attempted. Note that a verified write with a size not divisible by 4 gives an AuthErr.

ErrCode	10	AuthErr	<p>Authorisation error:</p> <p>Rejected RMAP commands when hardware support enabled:</p> <ul style="list-style-type: none"> <li>- Read or Write with extended address exceeding 0F<sub>16</sub></li> <li>- Verified-Write with non-word aligned address</li> <li>- Verified-Write with size 0 or size not divisible by 4</li> </ul> <p>Rejected RMAP commands when software support is disabled and hardware support is enabled:</p> <ul style="list-style-type: none"> <li>- Read-Modify-Write</li> <li>- Non-Incrementing Read or Write</li> </ul> <p>Any RMAP command when neither software nor hardware support is enabled.</p> <p>Rejected VCTP packet when the corresponding RxVC[\$] channel was not enabled.</p>
	12	DLAErr	Non-matching Destination Logical Address.
	16	HeadErr	<p>One of the following errors was detected while receiving the packet header:</p> <p>EOP or EEP detected before the first four bytes could be received.</p> <p>For RMAP commands supported by hardware, there was a header CRC error or an unexpected number of header bytes in the received packet.</p> <p>For VCTP packets, the received Virtual Channel Identifier was not within the implemented RxVC range</p>
	17	PtclIDErr	Protocol Identifier not supported
	Other	Other values	Other values cannot occur.
FFB3	0 - 255	<p>First Failing Packet Byte 3:</p> <p>If RMAP this field represents the Command byte.</p> <p>If VCTP this field is a dummy byte which is recommended to contain a copy of VCID.</p>	
PtclId	0 - 255	<p>First Failing Packet Byte 2:</p> <p>This field represents the Protocol ID byte.</p>	
DLA	0 - 255	<p>First Failing Packet Byte 1:</p> <p>Destination Logical Address (DLA)</p>	

### 10.7.1.6SpaceWire RxVC registers

The virtual receive channels have each a register set as defined below. RxVC[0] is dedicated to RMAP commands and responses and unknown protocols to be processed in software. The RMAP hardware supported commands that are executed directly on reception do not use the RxVC[0] register set except for the configuration fields DLA and RMAPEn. Thus, the RMAP hardware command execution is not visible, as seen from the application software.

#### **SPW RxVC Config Register [0] [SPW\_RxCnf[0]]**

**R/W**

31	15	8	4	3	2	1	0
-	DLA	-	Force Unk	TrPctl SW	RMAP En	Word Align	Rx En
	FE h		0	0	1	0	0
MSB			LSB				

**Function:** This register configures some common resources used by all reception channels and the reception of packets with other Protocol IDs than configured for the SpaceWire Virtual Channel Transfer Protocol in the SPW VC Transfer Protocol ID Register.

**Timing:**

**Constraints:** The word alignment function assumes that the RxVC DMA Base Address [\$] is programmed with a word aligned start address.

<i>Field:</i>	<i>Value:</i>	<i>Description:</i>
RxEn	0	The virtual receive channel is disabled. No packets are received on this virtual receive channel and no data are stored to memory. Note that this bit is only used during the identification process described in 5.5.3.2. I.e., when disabled any ongoing packet will be completed. Except for RMAP commands supported in hardware, all packets are rejected. If the RMAPEn bit is not set, then RMAP commands supported in hardware are also rejected. RMAP commands intended for further software processing are thus discarded. All RMAP responses are discarded All unknown transfer protocols are discarded. This ensures that no congestion occurs on the SpaceWire link.
	1	The virtual receive channel is enabled. If the RMAPEn bit is not set, then RMAP commands supported in hardware are received on RxVC[0] and stored in memory, provided that the TrPctlSW bit is set, else they are rejected. All RMAP commands unsupported in hardware are received on RxVC[0] and stored in memory, provided that the TrPctlSW bit is set, else they are rejected. All RMAP responses are received on RxVC[0] and stored in memory. All unknown transfer protocols are received on RxVC[0] and stored in memory, provided that the TrPctlSW bit is set, else they are rejected.
WordAlign	0	No padding takes place and a packet will start at the byte address following the previous packet.
	1	The end of each packet is padded up to the closest 32-bit word boundary. Padding is done with arbitrary content. The first byte of a packet will thus always be aligned to a 32-bit word address.



RMAPEn	0	RMAP hardware support is disabled. RMAP commands are received on RxVC[0] and stored in memory, provided that the RxEn and TrPctlSW bits are both set, else they are rejected.
	1	RMAP hardware support is enabled, as well as RMAP command identification. RMAP commands supported in hardware are not received on RxVC[0] and thus not stored in memory, but are handled in hardware. RMAP commands unsupported in hardware are received on RxVC[0] and stored in memory, provided that the RxEn and TrPctlSW bits are both set.
TrPctlSW	0	Software support for transfer protocols is disabled. RMAP commands are neither received on RxVC[0] nor stored in memory. If the RMAPEn bit is not set, all RMAP commands are rejected, otherwise only RMAP commands not supported in hardware are rejected. RMAP response storage is unaffected by the value of this bit. Unknown transfer protocols are neither received on RxVC[0] nor stored.
	1	Software support for transfer protocols is enabled. If the RMAPEn bit is not set, then RMAP commands supported in hardware are received on RxVC[0] and stored in memory, provided that the RxEN bit is set, else they are rejected. All RMAP commands unsupported in hardware are received on RxVC[0] and stored in memory, provided that the RxEN bit is set, else they are rejected. RMAP response storage is unaffected by the value of this bit. All unknown transfer protocols are received on RxVC[0] and stored in memory, provided that the RxEN bit is set, else they are rejected.
ForceUnk	0	Incoming packets are interpreted, tested and handled as defined by RxEn, TrPctlSW and RMAPEn.
	1	All incoming packets of 4 bytes or more are handled like packets of unknown protocol; i.e. they are received on RxVC[0] and stored in memory, provided that TrPctlSW and RxEN are set, otherwise they are rejected.
DLA	0-255	SpaceWire Destination Logical Address (DLA). The DLA of any incoming packet must match the DLA field, otherwise it will be rejected.

The handling of transfer protocols through RxVC[0] is summarised as follows:

Detailed <b>SPW RxVC Config Register[0]</b> bit usage:				
	RMAPEn=1 (Enabled)			
	RxEn=1 (Enabled)		RxEn=0 (Disabled)	
	TrPctlSW=1 (Enabled)	TrPctlSW=0 (Disabled)	TrPctlSW=1 (Enabled)	TrPctlSW=0 (Disabled)
	<b>OK</b>	<b>OK</b>	<b>OK</b>	<b>OK</b>
RMAP HW supported	<b>OK</b>	<b>OK</b>	<b>OK</b>	<b>OK</b>
RMAP not HW supported	<b>WrMem</b>	<b>Reject</b>	<b>Reject</b>	<b>Reject</b>
RMAP Response	<b>WrMem</b>	<b>WrMem</b>	<b>Reject</b>	<b>Reject</b>
Unknown protocol	<b>WrMem</b>	<b>Reject</b>	<b>Reject</b>	<b>Reject</b>
	RMAPEn=0 (Disabled)			
	RxEn=1 (Enabled)		RxEn=0 (Disabled)	
	TrPctlSW=1 (Enabled)	TrPctlSW=0 (Disabled)	TrPctlSW=1 (Enabled)	TrPctlSW=0 (Disabled)
	<b>WrMem</b>	<b>Reject</b>	<b>Reject</b>	<b>Reject</b>
RMAP HW supported	<b>WrMem</b>	<b>Reject</b>	<b>Reject</b>	<b>Reject</b>
RMAP not HW supported	<b>WrMem</b>	<b>Reject</b>	<b>Reject</b>	<b>Reject</b>
RMAP Response	<b>WrMem</b>	<b>WrMem</b>	<b>Reject</b>	<b>Reject</b>
Unknown protocol	<b>WrMem</b>	<b>Reject</b>	<b>Reject</b>	<b>Reject</b>

Legend:

*OK* means RMAP command executed by hardware, *WrMem* means that the packet is written to memory using RxVC[0] channel and *Reject* means that the full packet is rejected.

*RMAP not HW supported* means that the RMAP command byte content is not supported by the handler implemented in hardware. If a command is considered to be *RMAP HW supported*, but the address alignment or block size is not supported, the command will be rejected instead of transferred to software for further processing.

### **SPW RxVC Config Register [\$] [SPW\_RxCnf[\$]]**

**R/W**

31	2	1	0
-			Word
			Align
			0
			Rx
			En
			0
MSB			LSB

**Function:** This register configures the reception of SpaceWire Virtual Channel Transfer Protocol packets.

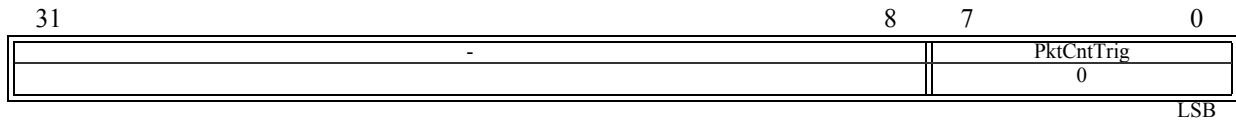
**Timing:**

**Constraints:** Valid index range is (1 <= [\$] <= RxVC\_G).

The word alignment function assumes that the RxVC DMA Base Address [\$] is programmed with a word aligned start address.

Note that all RxVC use the DLA configured in SPW RxVC Config Register [0].

<i>Field:</i>	<i>Value:</i>	<i>Description:</i>
RxEn	0	The virtual receive channel is disabled. All packets are discarded. No packets are received on this virtual receive channel and no data are stored to memory. Note that this bit is only used during the identification process described in 5.5.3.2. I.e., when disabled any ongoing packet will be completed.
	1	The virtual receive channel is enabled.
WordAlign	0	No padding takes place and a packet will start at the byte address following the previous packet.
	1	The end of each packet is padded up to the closest 32-bit word boundary. Padding is done with arbitrary content. The first byte of a packet will thus always be aligned to a 32-bit word address.

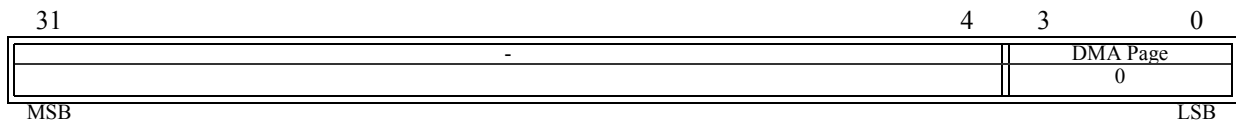
**SPW RxVC Packet Counter Register [S] [SPW\_RxPR[S]]****R/W**

**Function:** This register is used for configuring the number of packets to be received for a block transfer.

**Timing:**

**Constraints:**

Field:	Value:	Description:
PktCntTrig	0	The packet counter is not enabled. Any number of packets can be received.
	1-255	Number of SpaceWire packets to be received on the RxVC until the packet trigger (CntTrig) interrupt is issued. (After this, the virtual channel does not receive any further packets until triggered by software again.)

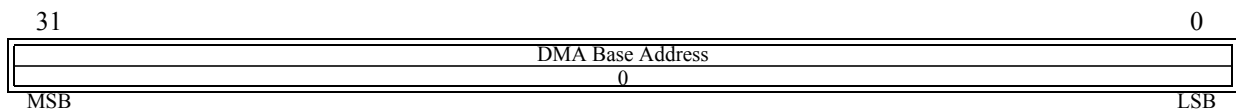
**SPW RxVC DMA Page Register [S] [SPW\_RxDPage[S]]****R/W**

**Function:** This register defines the page address for the virtual receive channel.

**Timing:**

**Constraints:**

Field:	Description:
DMA Page	Extended Address for a DMA access, corresponding to A35:A32 which allow a total of 64 Gbyte addressing space.

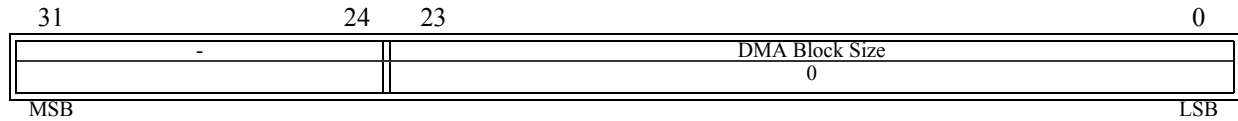
**SPW RxVC DMA Base Address Register [S] [SPW\_RxDBAR[S]]****R/W**

**Function:** This register defines the base address for the virtual receive channel.

**Timing:**

**Constraints:**

Field:	Description:
DMA Base Address	Byte base address for a DMA access.

**SPW RxVC DMA Block Size Register [\$\_] [SPW\_RxDBSR[\$\_]]****R/WT**

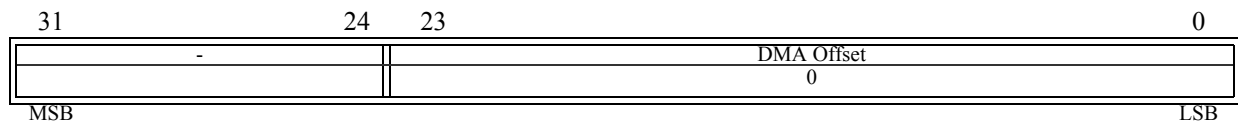
**Function:** This register is used for configuring the maximum number of bytes to be received for a block transfer.

Writing to this register is used as an automatic start of a DMA block transfer. The RxVC[\$] is triggered by software when a write access is made to the this register unless the SPW RxVC Config Register [0].RxEn bit is cleared or any RxVC or TxVC FlushAbort interrupt is pending. All relevant counters are automatically cleared for the virtual receive channel and any previous halt condition is released. The SPW RxVC Status Register \$.ChTrig bit is automatically set.

**Timing:**

**Constraints:** This register should not be written if the channel is already triggered. The block size should never be written with the value 0.

Field:	Description:
DMA Block Size	Total number of bytes to be received during a DMA block transfer.

**SPW RxVC DMA Offset Register [\$\_] [SPW\_RxDOR[\$\_]]****R/W**

**Function:** This register indicates the next byte address to be written, relative to the DMA Base Address.

**Timing:** This register is reset automatically when the channel is triggered and is incremented automatically as the reception progresses.

**Constraints:** This register must not be written to during reception. The writability of this register is implemented for test and debug purposes only.

Field:	Description:
DMA Offset	Byte address pointer relative to DMA Base Address. The DMA access address is DMA Base Address + DMA Offset.

**SPW RxVC Status Register [\$] [SPW\_RxSR[\$]]****R**

31	9	8	7	0
-	Ch Trig	Ch Act	PktCnt	
	0	0	0	

MSBLSB

**Function:** This register is used for monitoring the activity of the virtual receive channel.

**Timing:** The PktCnt field is reset to zero when the SPW RxVC DMA BlockSize Register[\$] register is written

**Constraints:**

Field:	Value:	Description:
PktCnt	Any	Number of packets received on RxVC \$ The <u>PktCnt</u> field is incremented, if the value is below the value in the <u>SPW RxVC Packet Count Register[\$]</u> , when an EOP control character is received, the packet reception was correctly finalised and the last data has been written to memory. If an error is encountered the packet counter will remain unchanged and indicate the number of packets correctly received, excluding the failing packet.
ChAct	0	RxVC \$ is inactive
	1	RxVC \$ is active. This indicates that a SpaceWire packet is being received on the channel.
ChTrig	0	RxVC \$ is not triggered
	1	RxVC \$ is triggered (SPW DMA Block Size Register [\$] written)  Note that the channel will not be triggered in the case all FlushAbort interrupt bits have not been cleared.

**SPW RxVC Current Packet Status Register [\$] [SPW\_RxCPSR[\$]]****R**

31	24	23	0
-	CurPktStart		
	0		

MSBLSB

**Function:** This register is used to indicate the start byte address of the packet currently being received, relative to the DMA base address.

The register contents can be used for setting up a new DMA block transfer after e.g. a link disconnect error. I.e., if the truncated packet is going to be resent after reconnection, then the SPW RxVC DMA Base Address Register [\$] can be incremented with the value in SPW RxVC Current Packet Status Register [\$] in order to overwrite the faulty packet.

**Timing:** The CurPktStart field is reset to zero when the SPW RxVC DMA Block Size Register[\$] is written.

**Constraints:**

Field:	Value:	Description:
--------	--------	--------------

CurPktStart	Any	The current packet's start offset The CurPktStart field is set to <u>SPW RxVC DMA Offset Register[\$]</u> when an EOP control character is received, the packet reception is correctly finalised and the last data has been written to memory. I.e. if an error is encountered, the current packet start address remains unchanged and indicates the failing packet.
-------------	-----	---

**SPW RxVC Interrupt Status Register [\$] [SPW\_RxISR[\$]]** **R**  
**SPW RxVC Interrupt Status Set Register [\$] [SPW\_RxISSR[\$]]** **S**  
**SPW RxVC Interrupt Status Clear Register [\$] [SPW\_RxISCR[\$]]** **C**

31		6	5	4	3	2	1	0
	-	CRC Data Err	Flush Abort	Dma Wr Err	Rx EEP	Cnt Trig	Rx EOB	
		0	0	0	0	0	0	

LSB

**Function:** RxVC[\$] interrupt status, set and clear registers.  
**Timing:** An RxVC channel may issue interrupts when it is active or triggered.

The RxEOB, CntTrig, RxEEP, and CRCDataErr interrupts are issued after the last access to memory.

The DmaWrErr interrupt is issued directly after an error has been detected on the internal bus.

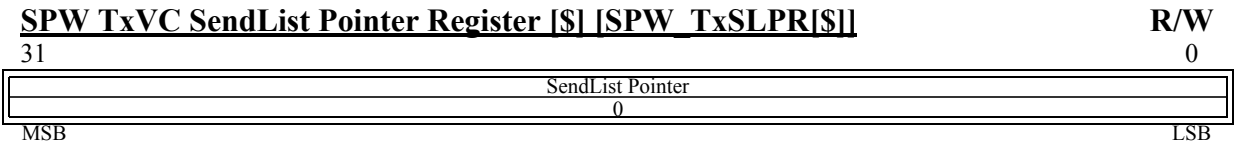
The FlushAbort interrupt is issued directly after the flush has been commanded.

**Constraints:** CRCDataErr is only available for RxVC[0].

<i>Field:</i>	<i>Description:</i>
RxEOB	RxVC DMA Offset has reached the RxVC DMA Block Size
CntTrig	The configured number of packets, <u>SPW RxVC Packet Counter Register.PktCntTrig</u> , has been received and stored in memory. Can only occur if the configured number of packets has been set higher than zero.
RxEEP	An EEP has been received.
DmaWrErr	Error when writing to memory, i.e. access error on internal bus, e.g. illegal address. The remainder of the packet is discarded until EOP or EEP. The interrupt is issued immediately.
FlushAbort	An <u>SPW CODEC Configuration Register.RxFifoFlush</u> was commanded while the channel was active or triggered. The remainder of the packet is discarded and the channel untriggered.
CRCDataErr	A CRC8 checksum error has been detected on RxVC[0], either in the header or data field of an RMAP command or response, both intended to be processed by software.

10.7.1.7SpaceWire TxVC registers

The Tx channels each has a register set as defined below. TxVC[0] is dedicated to RMAP read and responses and uses the same structure but need no register accesses to execute a transmission. The TxVC[0] registers, with the exception of SPW TxVC Status Register [0], are included for debug and test purposes only and should not be used in normal operation.



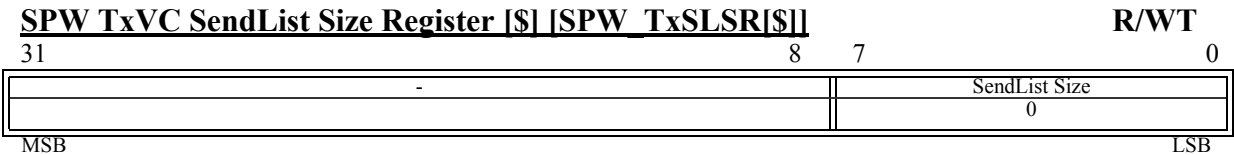
**Function:** This register is used to define the position of the send list for the virtual transmit channel, and for monitoring the transmit progress by indicating the current sent list entry.

The SendList Pointer field points to the first word in the send list entry (called current position during transmission). The pointer is incremented by 16 after the send list entry has been successfully transmitted.

**Timing:** .

**Constraints:** This register must not be written to during transmission.  
The pointer must be word-aligned.  
For TxVC[0], writes to this register have no effect.

Field:	Description:
Send List Pointer	Address to the present element in the send list.



**Function:** The SendList Size field indicates the size of the send list, counted in number of send list entries. It can be used for monitoring the progress by indicating the remaining number of entries in the send list.

A write access to the SPW TxVC Sendlist Size [\$] register triggers the TxVC[\$] send list fetch operation. The TxVC[\$] channel will participate in the arbitration and eventually start the transmission on the SpaceWire link.

The size is decremented by one after the send list entry has been successfully transmitted.

**Timing:** .

**Constraints:** This register must not be written to during transmission.  
For TxVC[0], writes to this register have no effect.

Field:	Description:
Send List Size	Number of remaining send list entries.

### **SPW TxVC Status Register [\$] [SPW\_TxSR[\$]]**

**R**

31	2	1	0
		Ch Trig	Ch Act
		0	0
MSB			LSB

**Function:** This register is used for monitoring the activity of the virtual transmit channel.

**Timing:**

**Constraints:**

Field:	Value:	Description:
ChAct	0	TxVC \$ is inactive
	1	TxVC \$ is active, i.e. is currently transmitting.
ChTrig	0	TxVC \$ is not started
	1	TxVC \$ is started (SPW TxVC Send List Size [\$] written)
Note that for all TxVC except TxVC[0] the channel will not be triggered in the case all FlushAbort interrupt bits have not been cleared.		

### **SPW TxVC Interrupt Status Register \$ [SPW\_TxISR\$]**

**R**

### **SPW TxVC Interrupt Status Set Register \$ [SPW\_TxISSR\$]**

**S**

### **SPW TxVC Interrupt Status Clear Register \$ [SPW\_TxISCR\$]**

**C**

31	4	3	2	1	0
		Flush Abort	Dma Rd Err	Tx EOB	
		0	0	0	
					LSB

**Function:** TxVC[\$] interrupt status, set and clear registers. No interrupts are ever issued in TxVC[0].

**Timing:** A TxVC channel may issue interrupts when it is active or triggered.

The TxEOB interrupt is issued after the last access to memory.

The DmaRdErr interrupt is issued directly after an error has been detected on the internal bus.

The FlushAbort interrupt is issued directly after the flush has been commanded.

**Constraints:** For TxVC[0], writes to these registers have no effect.



<i>Field:</i>	<i>Description:</i>
TxEob	All messages transmitted, the send list has been completed.
DmaRdErr	Error when reading from memory, i.e. access error on internal bus, e.g. illegal address. An EEP is inserted and the ongoing send list is stopped. The atomic send list entry handling is released, allowing the TxVC arbiter to reassign another channel. The interrupt is issued immediately. The send list pointer and the send list size remain unchanged in the case an error occurs while reading a send list entry.
FlushAbort	An <u>SPW CODEC Configuration Register.TxFifoFlush</u> was commanded while the channel was active or triggered. The channel is untriggered and the ongoing transmission is stopped.

## 10.8 Vendor and device id

The module has vendor id 0x04 and device id 0x12.

## 11. AMBA AHB CONTROLLER

### 11.1 Overview

The AHB controller is a combined AHB arbiter, bus multiplexer and slave decoder. The controller supports up to 16 AHB masters, and 16 AHB slaves.

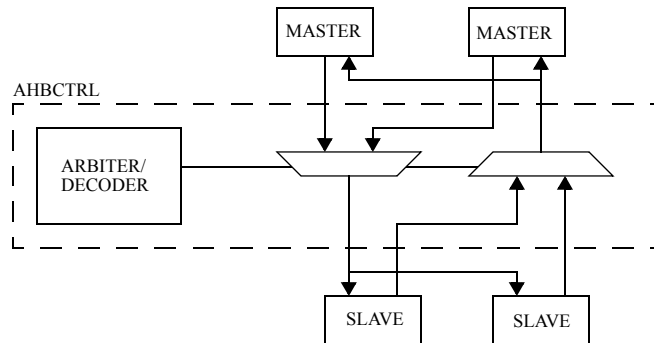


Figure 11-1. AHB Controller block diagram

### 11.2 Operation

#### 11.2.1 Arbitration

The AHB controller supports a round-robin arbitration algorithm. In round-robin mode, priority is rotated one step after each AHB transfer. If no master requests the bus, the last owner will be granted (bus parking). During incremental bursts, the AHB master should keep the bus request asserted until the last access or it might lose bus ownership.

#### 11.2.2 Decoding

Access to unused addresses will cause an AHB error response.

#### 11.2.3 Plug&play information

GRLIB devices contain a number of plug&play information words which are included in the AHB records they drive on the bus (see the GRLIB user's manual for more information). These records are combined into an array which is connected to the AHB controller unit. The plug&play information is mapped on a read-only address area mapped on address 0xFFFFF000 - 0xFFFFFFFF.

The master information is placed on the first 2Kbyte of the block, while the slave information id placed on the second 2Kbyte block. Each unit occupies 32 bytes, which means that the area has place for 64 masters and 64 slaves.

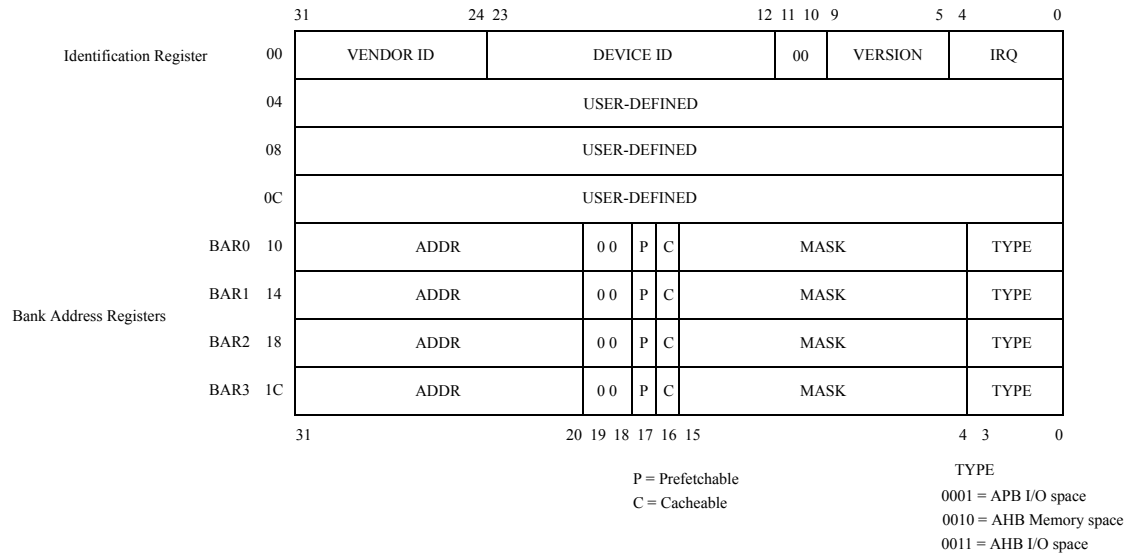


Figure 11-2. AHB plug&play information record

# 12. AMBA AHB/APB BRIDGE

## 12.1 Overview

The APB bridge is a APB bus master. The controller supports up to 16 slaves.

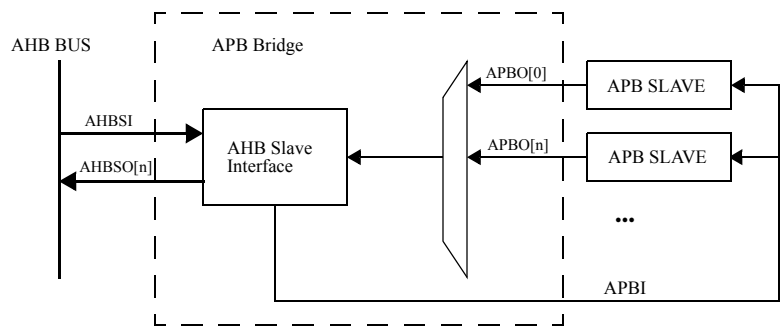


Figure 12-1. APB Bridge block diagram

## 12.2 Operation

### 12.2.1 Decoding

A slave can occupy any binary aligned address space with a size of 256 bytes - 1 Mbyte.

### 12.2.2 Plug&play information

GRLIB APB slaves contain two plug&play information words which are included in the APB records they drive on the bus (see the GRLIB User's manual for more information). These records are combined into an array which is connected to the APB bridge. The plug&play information is mapped on a read-only address area at the top 4 kbytes of the bridge address space. Each plug&play block occupies 8 bytes. The plug&play information is mapped on a read-only address area mapped on address 0x800FF000 - 0x800FF1FF.

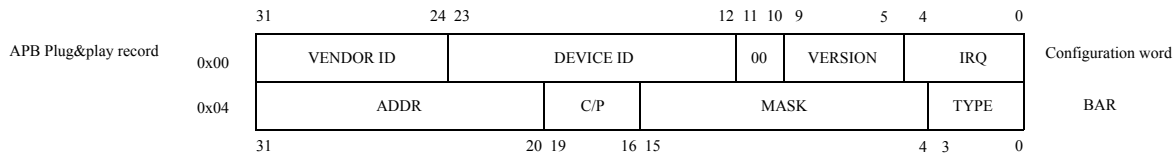


Figure 12-2. APB plug&play information

## 12.3 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x06.

## 13. MEMORY AND REGISTER MAP, INTERRUPT ASSIGNMENT

### 13.1 Addressing information

The SpaceWire-RTC global memory map is shown in the table below. The global address map is defined by the addresses on the internal AMBA AHB bus.

Address range	Size	Mapping	Module
0x00000000 - 0x1FFFFFFF	512 M	Prom	Memory controller
0x20000000 - 0x3FFFFFFF	512 M	Memory bus I/O	
0x40000000 - 0x7FFFFFFF	1 G	SRAM	
0x80000000 - 0x800FEFF		On-chip registers	APB controller / bridge
0x800FF000 - 0x800FF1FF	512	Plug and play information	
0x90000000 - 0x9FFFFFFF	256 M	Debug support unit	DSU
0xA0000000 - 0xAFFFFFFF	64 K	On-chip RAM	On-chip RAM
0xFFFFF000 - 0xFFFFFFFF	4 K	Plug and play information	AHB controller

#### Global address map (AMBA AHBbus)

The SpaceWire-RTC effective memory map is shown in the table below. The address map is defined by the addresses on the internal AMBA AHB bus. The address ranges shown correspond to a system equipped with the supported maximum number of memory devices each with the supported maximum size. The constraining of the addressable memory is made by the width of the external memory address bus.

Address range	Size	Mapping	Cacheable	Module
0x00000000 - 0x00FFFFFF	16 M	Prom	Yes	Memory controller
0x20000000 - 0x207FFFFF	8 M	Memory bus I/O	No	
0x40000000 - 0x41FFFFFF	32 M	SRAM	Yes	
0xA0000000 - 0xA000FFFF	64 k	On-chip RAM	Yes	On-chip RAM

Table 13-1. Effective memory address map with cacheability

The SpaceWire-RTC global register map is shown in the table below. The global register address map is defined by the addresses on the internal AMBA APB bus.

Address range	Size	Mapping	Module
0x80000000 - 0x800000FF	256	LEON2-FT registers	LEON2-FT
0x80010000 - 0x800100FF	256	On-Chip Memory	FTAHBRAM
0x80020000 - 0x800200FF	256	24-bit GPIO	GPPULSE
0x80030000 - 0x800300FF	256	32-bit Timers	GRTIMER
0x80040000 - 0x800400FF	256	ADC/DAC interface	GRADCDAC
0x80050000 - 0x800500FF	256	FIFO interface	GRFIFO
0x80060000 - 0x80060FFF	4096	SpaceWire Interface - 0	SPW2
0x80070000 - 0x80070FFF	4096	SpaceWire Interface - 1	SPW2
0x80080000 - 0x800803FF	1024	CAN interface	GRHCAN

Table 13-2. Global register address map (AMBA APB bus)

## 13.2 Plug & Play information

The SpaceWire-RTC plug&play information for the masters on the internal AMBA AHB bus is shown in the table below.

Address range	Size	Mapping	Module
0xFFFFF000 - 0xFFFFF01F	32	LEON2-FT Caches	LEON2-FT
0xFFFFF020 - 0xFFFFF03F	32	LEON2-FT DSU UART	LEON2-FT
0xFFFFF040 - 0xFFFFF05F	32	FIFO interface	GRFIFO
0xFFFFF060 - 0xFFFFF07F	32	SpaceWire Interface - 0 Rx	SPW2
0xFFFFF080 - 0xFFFFF09F	32	SpaceWire Interface - 0 Tx	SPW2
0xFFFFF0A0 - 0xFFFFF0BF	32	SpaceWire Interface - 1 Rx	SPW2
0xFFFFF0C0 - 0xFFFFF0DF	32	SpaceWire Interface - 1 Tx	SPW2
0xFFFFF0E0 - 0xFFFFF0FF	32	CAN interface	GRHCAN

**Table 13-3. Plug&play information for AHB masters**

The SpaceWire-RTC plug&play information for the slaves on the internal AMBA AHB bus is shown in the table below.

Address range	Size	Mapping	Module
0xFFFFF800 - 0xFFFFF81F	32	APB Controller	APB Controller
0xFFFFF820 - 0xFFFFF83F	32	Memory controller	LEON2-FT
0xFFFFF840 - 0xFFFFF85F	32	DSU	LEON2-FT
0xFFFFF860 - 0xFFFFF87F	32	On-Chip Memory	FTAHBRAM

**Table 13-4. Plug&play information for AHB slaves**

The SpaceWire-RTC plug&play information for the slaves on the internal AMBA APB bus is shown in the table below.

Address range	Size	Mapping	Module
0x800FF000 - 0x800FF007	8	LEON2-FT registers	LEON2-FT
0x800FF008 - 0x800FF00F	8	On-Chip Memory registers	FTAHB
0x800FF010 - 0x800FF017	8	24-bit GPIO	GPPULSE
0x800FF018 - 0x800FF01F	8	32-bit Timers	GRTIMER
0x800FF020 - 0x800FF027	8	ADC/DAC interface	GRADCDAC
0x800FF028 - 0x800FF02F	8	FIFO interface	GRFIFO
0x800FF030 - 0x800FF037	8	SpaceWire Interface - 0	SPW2
0x800FF038 - 0x800FF03F	8	SpaceWire Interface - 1	SPW2
0x800FF040 - 0x800FF047	8	CAN interface	GRHCAN

**Table 13-5. Plug&play information for APB slaves**

## 13.3 Registers

### 13.3.1 Processor and peripherals

Address	Register	Address	
0x80000000	Memory configuration register 1	0x800000B0	Secondary interrupt mask register
0x80000004	Memory configuration register 2	0x800000B4	Secondary interrupt pending register
0x80000008	Memory configuration register 3	0x800000B8	Secondary interrupt status register
0x8000000C	AHB Failing address register	0x800000B8	Secondary interrupt clear register
0x80000010	AHB status register		
0x80000014	Cache control register	0x800000C4	DSU UART status register
0x80000018	Power-down register	0x800000C8	DSU UART control register
0x8000001C	Write protection register 1	0x800000CC	DSU UART scaler register
0x80000020	Write protection register 2		
0x80000024	LEON configuration register	0x800000D0	Write protect start address 1
0x80000040	Timer 1 counter register	0x800000D4	Write protect end address 1
0x80000044	Timer 1 reload register	0x800000D8	Write protect start address 2
0x80000048	Timer 1 control register	0x800000DC	Write protect end address 2
0x8000004C	Watchdog register		
0x80000050	Timer 2 counter register		
0x80000054	Timer 2 reload register		
0x80000058	Timer 2 control register		
0x80000060	Prescaler counter register		
0x80000064	Precler reload register		
0x80000070	Uart 1 data register		
0x80000074	Uart 1 status register		
0x80000078	Uart 1 control register		
0x8000007C	Uart 1 scaler register		
0x80000080	Uart 2 data register		
0x80000084	Uart 2 status register		
0x80000088	Uart 2 control register		
0x8000008C	Uart 2 scaler register		
0x80000090	Interrupt mask and priority register		
0x80000094	Interrupt pending register		
0x80000098	Interrupt force register		
0x8000009C	Interrupt clear register		
0x800000A0	I/O port input/output register		
0x800000A4	I/O port direction register		
0x800000A8	I/O port interrupt config. register 1		
0x800000AC	I/O port interrupt config. register 2		

Table 13-6. LEON2-FT registers

### 13.3.2 On-Chip Memory

Register	Address
Configuration Register	0x80010000

Table 13-7. On-Chip Memory registers

### 13.3.3 FIFO Interface

Register	Address
Configuration Register	0x80050000
Control Register	0x80050008
Transmit Channel Control Register	0x80050020
Transmit Channel Status Register	0x80050024
Transmit Channel Address Register	0x80050028
Transmit Channel Size Register	0x8005002C
Transmit Channel Write Register	0x80050030
Transmit Channel Read Register	0x80050034
Transmit Channel Interrupt Register	0x80050038
Receive Channel Control Register	0x80050040
Receive Channel Status Register	0x80050044
Receive Channel Address Register	0x80050048
Receive Channel Size Register	0x8005004C
Receive Channel Write Register	0x80050050
Receive Channel Read Register	0x80050054
Receive Channel Interrupt Register	0x80050058
Data Input Register	0x80050060
Data Output Register	0x80050064
Data Direction Register	0x80050068

Table 13-8. FIFO Interface registers



### 13.3.4 ADC/DAC Interface

Register	Address
Configuration Register	0x80040000
Status Register	0x80040004
ADC Data Input Register	0x80040010
DAC Data Output Register	0x80040014
Address Input Register	0x80040020
Address Output Register	0x80040024
Address Direction Register	0x80040028
Data Input Register	0x80040030
Data Output Register	0x80040034
Data Direction Register	0x80040038

**Table 13-9. ADC/DAC Interface registers**

### 13.3.5 32-bit Timers

Register	Address
Scaler value	0x80030000
Scaler reload value	0x80030004
Configuration register	0x80030008
Timer latch configuration register	0x8003000C
Timer 1 counter value register	0x80030010
Timer 1 reload value register	0x80030014
Timer 1 control register	0x80030018
Timer 1 latch register	0x8003001C
Timer 2 counter value register	0x80030020
Timer 2 reload value register	0x80030024
Timer 2 control register	0x80030028
Timer 2 latch register	0x8003002C

**Table 13-10. 32-bit Timers registers**

### 13.3.6 24-bit General Purpose Input Output

Register	Address
Input Register	0x80020000
Output Register	0x80020004
Direction Register	0x80020008
Interrupt Mask Register	0x8002000C
Interrupt Polarity Register	0x80020010
Interrupt Edge Register	0x80020014
Pulse Register	0x80020018
Pulse Counter Register	0x8002001C

**Table 13-11. 24-bit General Purpose Input Output registers**

### 13.3.7 CAN Interface

Register	Address
Configuration Register	0x80080000
Status Register	0x80080004
Control Register	0x80080008
SYNC Mask Filter Register	0x80080018
SYNC Code Filter Register	0x8008001C
Pending Interrupt Masked Status Register	0x80080100
Pending Interrupt Masked Register	0x80080104
Pending Interrupt Status Register	0x80080108
Pending Interrupt Register	0x8008010C
Interrupt Mask Register	0x80080110
Pending Interrupt Clear Register	0x80080114
Transmit Channel Control Register	0x80080200
Transmit Channel Address Register	0x80080204
Transmit Channel Size Register	0x80080208
Transmit Channel Write Register	0x8008020C
Transmit Channel Read Register	0x80080210
Transmit Channel Interrupt Register	0x80080214
Receive Channel Control Register	0x80080300
Receive Channel Address Register	0x80080304
Receive Channel Size Register	0x80080308
Receive Channel Write Register	0x8008030C
Receive Channel Read Register	0x80080310
Receive Channel Interrupt Register	0x80080314
Receive Channel Mask Register	0x80080318
Receive Channel Code Register	0x8008031C

**Table 13-12. CAN Controller registers**

### 13.3.8 SpaceWire Link Interface - 0

Address	Register
0x80060000	SPW Pending Interrupt Masked Status Register
0x80060008	SPW Pending Interrupt Status Register
0x80060010	SPW Interrupt Mask Register
0x80060014	SPW Link Interrupt Status Register
0x80060018	SPW Link Interrupt Status Set Register
0x8006001C	SPW Link Interrupt Status Clear Register
0x80060020	SPW CODEC Configuration Register
0x80060024	SPW Clock Division Register

Address	Register
0x80060028	SPW RMAP Destination Key Register
0x8006002C	SPW Transmit Time-Code Register
0x80060030	SPW VC Transfer Protocol ID Register
0x80060034	SPW SW Transmit Time-Code Register
0x80060040	SPW Status Register
0x80060044	SPW CODEC Status Register
0x80060048	SPW Receive Time-Code Status Register
0x80060050	SPW Transmit Time-Code Mask Register
0x80060080	SPW First Failing Packet Register
0x80060100	SPW RxVC Config Register [0]
0x80060104	SPW RxVC Packet Counter Register [0]
0x80060108	SPW RxVC DMA Page Register [0]
0x8006010C	SPW RxVC DMA Base Address Register [0]
0x80060110	SPW RxVC DMA Block SizeRegister [0]
0x80060114	SPW RxVC DMA Offset Register [0]
0x80060118	SPW RxVC Status Register [0]
0x8006011C	SPW RxVC Current Packet Status Register [0]
0x80060120	SPW RxVC Interrupt Status Register [0]
0x80060124	SPW RxVC Interrupt Status Set Register [0]
0x80060128	SPW RxVC Interrupt Status Clear Register [0]
0x80060140	SPW RxVC Config Register [1]
0x80060144	SPW RxVC Packet Counter Register [1]
0x80060148	SPW RxVC DMA Page Register [1]
0x8006014C	SPW RxVC DMA Base Address Register [1]
0x80060150	SPW RxVC DMA Block SizeRegister [1]
0x80060154	SPW RxVC DMA Offset Register [1]
0x80060158	SPW RxVC Status Register [1]
0x8006015C	SPW RxVC Current Packet Status Register [1]
0x80060160	SPW RxVC Interrupt Status Register [1]
0x80060164	SPW RxVC Interrupt Status Set Register [1]
0x80060168	SPW RxVC Interrupt Status Clear Register [1]
0x80060320	SPW TxVC SendList Pointer Register [1]
0x80060324	SPW TxVC SendList Size Register [1]
0x80060328	SPW TxVC Status Register [1]
0x8006032C	SPW TxVC Interrupt Status Register [1]
0x80060330	SPW TxVC Interrupt Status Set Register [1]
0x80060334	SPW TxVC Interrupt Status Clear Register [1]

**Table 13-13. SpaceWire Link - 0 registers0**

### 13.3.9 SpaceWire Link Interface - 1

Address	Register
0x80070000	SPW Pending Interrupt Masked Status Register
0x80070008	SPW Pending Interrupt Status Register
0x80070010	SPW Interrupt Mask Register
0x80070014	SPW Link Interrupt Status Register
0x80070018	SPW Link Interrupt Status Set Register

Address	Register
0x8007001C	SPW Link Interrupt Status Clear Register
0x80070020	SPW CODEC Configuration Register
0x80070024	SPW Clock Division Register
0x80070028	SPW RMAP Destination Key Register
0x8007002C	SPW Transmit Time-Code Register
0x80070030	SPW VC Transfer Protocol ID Register
0x80070034	SPW SW Transmit Time-Code Register
0x80070040	SPW Status Register
0x80070044	SPW CODEC Status Register
0x80070048	SPW Receive Time-Code Status Register
0x80070050	SPW Transmit Time-Code Mask Register
0x80070080	SPW First Failing Packet Register
0x80070100	SPW RxVC Config Register [0]
0x80070104	SPW RxVC Packet Counter Register [0]
0x80070108	SPW RxVC DMA Page Register [0]
0x8007010C	SPW RxVC DMA Base Address Register [0]
0x80070110	SPW RxVC DMA Block SizeRegister [0]
0x80070114	SPW RxVC DMA Offset Register [0]
0x80070118	SPW RxVC Status Register [0]
0x8007011C	SPW RxVC Current Packet Status Register [0]
0x80070120	SPW RxVC Interrupt Status Register [0]
0x80070124	SPW RxVC Interrupt Status Set Register [0]
0x80070128	SPW RxVC Interrupt Status Clear Register [0]
0x80070140	SPW RxVC Config Register [1]
0x80070144	SPW RxVC Packet Counter Register [1]
0x80070148	SPW RxVC DMA Page Register [1]
0x8007014C	SPW RxVC DMA Base Address Register [1]
0x80070150	SPW RxVC DMA Block SizeRegister [1]
0x80070154	SPW RxVC DMA Offset Register [1]
0x80070158	SPW RxVC Status Register [1]
0x8007015C	SPW RxVC Current Packet Status Register [1]
0x80070160	SPW RxVC Interrupt Status Register [1]
0x80070164	SPW RxVC Interrupt Status Set Register [1]
0x80070168	SPW RxVC Interrupt Status Clear Register [1]
0x80070320	SPW TxVC SendList Pointer Register [1]
0x80070324	SPW TxVC SendList Size Register [1]
0x80070328	SPW TxVC Status Register [1]
0x8007032C	SPW TxVC Interrupt Status Register [1]
0x80070330	SPW TxVC Interrupt Status Set Register [1]
0x80070334	SPW TxVC Interrupt Status Clear Register [1]

**Table 13-14. SpaceWire Link - 1registers1**

## 13.4 Interrupts

### 13.4.1 Interrupt assignment - primary interrupt controller

The following table shows the assignment of interrupts for the primary interrupt controller

Interrupt	Source
15	Parallel I/O[7]
14	SpaceWire 1
13	SpaceWire 0 Parallel I/O[6]
12	CAN interface Parallel I/O[5]
11	DSU trace buffer
10	Second interrupt controller Parallel I/O[4]
9	Timer 2
8	Timer 1
7	Parallel I/O[3]
6	Parallel I/O[2]
5	Parallel I/O[1]
4	Parallel I/O[0]
3	UART 1
2	UART 2
1	AHB error

**Table 13-15. Interrupt assignments - primary interrupt controller**

### 13.4.2 Interrupt assignment - secondary interrupt controller

Next table shows the assignment of interrupts for the secondary interrupt controller.

Interrupt	Source	Comment
31	GPIO / Gpio[23]	24-bit GPIO input interrupt
30	GPIO / Gpio[22]	24-bit GPIO input interrupt
29	GPIO / Gpio[21]	24-bit GPIO input interrupt
28	GPIO / Gpio[20]	24-bit GPIO input interrupt
27	GPIO / Gpio[19]	24-bit GPIO input interrupt
26	GPIO / Gpio[18]	24-bit GPIO input interrupt
25	GPIO / Gpio[17]	24-bit GPIO input interrupt
24	GPIO / Gpio[16]	24-bit GPIO input interrupt
23-20	-	Unused
19	CAN/RxSync	Synchronization message received
18	CAN/TxSync	Synchronization message transmitted
17	CAN/IRQ	Common output from interrupt handler
16	SpaceWire 1/ Tick	Synchronization received
15	SpaceWire 1 / Interrupt	Common output from interrupt handler
14	SpaceWire 0 / Tick	Synchronization received
13	SpaceWire 0 / Interrupt	Common output from interrupt handler
12	FIFO/RxParity	Parity error during reception
11	FIFO/RxError	AHB access error during reception
10	FIFO/RxFull	Circular reception buffer full
9	FIFO/RxIrq	Successful reception of data block
8	FIFO/TxError	AHB access error during transmission
7	FIFO/TxEmpty	Circular transmission buffer empty
6	FIFO/TxIrq	Successful transmission of data block
5	ADC/DAC	DAC conversion ready
4	ADC/DAC	ADC conversion ready
3	32-Bit Timer/Timer 2	Timer expired
2	32-Bit Timer/Timer 1	Timer expired
1	GPIO/PULSE	Pulse command completed
0	-	Unused

**Table 13-16. Secondary interrupt controller assignments**

Note: Interrupt 17, 15 and 13 are available in primary interrupt controller and should therefore be used restrictively in the secondary interrupt controller. The secondary interrupt controller uses edge detection, whereas the aforementioned interrupt sources use level. The interrupt handling software must thus ensure that the sources for the aforementioned interrupts do not have an additional pending interrupt when clearing the corresponding bit in the pending interrupt register in the secondary interrupt controller. This limitation does not exist for the primary interrupt controller.

Note: Interrupts 31 down to 24 are connected to the inputs of the 24-bit General Purpose Input Output interface. The secondary interrupt controller uses edge detection. The 24-bit General Purpose Input Output interface must therefore only be programmed for edge detection, not for level, to ensure that multiple interrupts can be detected.

## 14. INTERFACES AND SIGNALS

The following table summarizes all the signals of the SpaceWire-RTC

Signal	Type	Description	Comment
LeonErrorN	IO, open-drain output	LEON Error	This active low output is asserted when the processor has entered error state and is halted. This happens when traps are disabled and an synchronous (un-maskable) trap occurs.
LeonWDN	IO, open-drain output	LEON watchdog	This active low output is asserted when the watchdog times-out.
LeonDsuEn	I	DSU enable	The active high input enables the DSU unit. If de-asserted, the DSU trace buffer will continue to operate but the processor will not enter debug mode.
LeonDsuTx	O	DSU UART transmit	This active high output provides the data from the DSU communication link transmitter.
LeonDsuRx	I	DSU UART receive	This active high input provides the data to the DSU communication link receiver.
LeonDsuBre	I	DSU break	A low-to-high transition on this active high input will generate break condition and put the processor in debug mode.
LeonDsuAct	O	DSU active	This active high output is asserted when the processor is in debug mode and controlled by the DSU.
LeonPio[15:0]	IO	LEON Parallel Input / Output	These bi-directional signals can be used as inputs or outputs to control external devices.
Gpio[23:0]	IO	General Purpose Input / Output	
TimeClk	I	External timer clock	
TimeTrig[2:1]	O	External timer trigger	Asserted for 8 system clock periods
CanTx[1:0]	O	CAN transmit	
CanRx[1:0]	I	CAN receive	
CanEn[1:0]	O	CAN transmit enable	
ADData[15:0]	IO	ADC/DAC data	
ADAddr[7:0]	IO	ADC/DAC address	
ADWr	O	DAC write strobe	
ADCs	O	ADC chip select	
ADRC	O	ADC read/convert	
ADRDy	I	ADC ready	
ADTrig	I	ADC trigger	
MemA[22:0]	O	Memory interface address	These active high outputs carry the address during accesses on the memory bus. When no access is performed, the address of the last access is driven (also internal cycles).



Signal	Type	Description	Comment
MemD[31:0]	IO	Memory interface data	MemD[31:0] carries the data during transfers on the memory bus. The processor only drives the bus during write cycles. During accesses to 8-bit areas, only MemD[31:24] are used.
MemCB[7:0]	IO	Memory interface checkbits	MemCB[6:0] carries the EDAC checkbits, MemCB[7] takes the value of TB[7] in the error control register. The processor only drive MemCB[7:0] during write cycles to areas programmed to be EDAC protected.
MemCsN[3:0]	O	SRAM chip select	These active low signals provide an individual output enable for each SRAM bank.
MemOeN[3:0]	O	SRAM output enable	These active low outputs provide the chip-select signals for each SRAM bank.
MemWrN[3:0]	O	SRAM byte write strobe	These active low outputs provide individual write strobes for each byte lane. MemWrN[0] controls MemD[31:24], MemWrN[1] controls MemD[23:16], etc.
RomCsN[1:0]	O	PROM chip select	These active low outputs provide the chip-select signal for the PROM area. RomCsN[0] is asserted when the lower half of the PROM area is accessed (0 - 0x10000000), while RomCsN[1] is asserted for the upper half.
IoCsN	O	I/O area chip select	This active low output is the chip-select signal for the memory mapped I/O area.
IoOeN	O	I/O area output enable	This active low output is asserted during read cycles on the memory bus.
IoRead	O	I/O area read	This active high output is asserted during read cycles on the memory bus.
IoWrN	O	I/O area write	This active low output provides a write strobe during write cycles on the memory bus.
IoBrdyN	I	I/O area ready	This active low input indicates that the access to a memory mapped I/O area can be terminated on the next rising clock edge.
MemBExcN	I	Memory exception	This active low input is sampled simultaneously with the data during accesses on the memory bus. If asserted, a memory error will be generated.
FifoD[15:0]	IO	FIFO data	
FifoP[1:0]	IO	FIFO parity	
FifoRdN	O	FIFO read strobe	
FifoWrN	O	FIFO write strobe	
FifoFullN	I	FIFO full	
FifoEmpN	I	FIFO empty	
FifoHalfN	I	FIFO half-full, half-empty	

Signal	Type	Description	Comment
SpwClkSrc	I	SpaceWire transmitter clock source	
SpwClkMult[1:0]	I	SpaceWire clock configuration	
SpwClk10Mbit[2:0]	I	SpaceWire clock configuration	
SpwClkPllCnfg[2:0]	I	SpaceWire clock configuration	
SpwClkMuxSel	I	SpaceWire clock configuration	External clock when 1, internal PLL when 0.
SpwDIn_P[1:0]	I, LVDS positive	SpaceWire Data input, positive	
SpwDIn_N[1:0]	I, LVDS negative	SpaceWire Data input, negative	
SpwSIn_P[1:0]	I, LVDS positive	SpaceWire Strobe input, positive	
SpwSIn_N[1:0]	I, LVDS negative	SpaceWire Strobe input, negative	
SpwDOut_P[1:0]	O, LVDS positive	SpaceWire Data output, positive	
SpwDOut_N[1:0]	O, LVDS negative	SpaceWire Data output, negative	
SpwSOut_P[1:0]	O, LVDS positive	SpaceWire Strobe output, positive	
SpwSOut_N[1:0]	O, LVDS negative	SpaceWire Strobe output, negative	

**Table 14-1. SpaceWire RTC Signals**





## 15. Revision Control

Rev	Date	Description
A	09/11	Creation