RMAP Initiator: Product User Guide

Contents

[Overview 3](#_Toc146895941)

[Core Architecture 3](#_Toc146895942)

[RMAP Command Controller 3](#_Toc146895943)

[Example Command Byte Structure 5](#_Toc146895944)

[Header & Data Interface 5](#_Toc146895945)

[RMAP Reply Controller 7](#_Toc146895946)

[Example Reply Byte Structure 7](#_Toc146895947)

[Header & Data Interface 8](#_Toc146895948)

[RMAP SpaceWire CoDec 9](#_Toc146895949)

[Design Notes 9](#_Toc146895950)

[Core Configuration 9](#_Toc146895951)

[Core Interface 10](#_Toc146895952)

[Parallel & FiFo Command/Reply Interface 10](#_Toc146895953)

[Tx Parallel Interface 11](#_Toc146895954)

[Submitting Path Addresses with Parallel Interfaces 11](#_Toc146895955)

[Rx Parallel Interface 11](#_Toc146895956)

[Clock & Reset Signals 12](#_Toc146895957)

[(Parallel) Command Header Ports 12](#_Toc146895958)

[Command Header Ports 12](#_Toc146895959)

[Command Data ports 13](#_Toc146895960)

[Command Error ports 13](#_Toc146895961)

[(Parallel) Reply Header Ports 13](#_Toc146895962)

[Reply Header ports 13](#_Toc146895963)

[Reply Data ports 13](#_Toc146895964)

[Reply Error ports 14](#_Toc146895965)

[SpaceWire Status ports 14](#_Toc146895966)

[SpaceWire Time Code ports 14](#_Toc146895967)

[SpaceWire IO Ports 14](#_Toc146895968)

[VHDL Packages 15](#_Toc146895969)

[rmap\_initiator\_lib 15](#_Toc146895970)

[SpaceWire\_Sim\_lib 15](#_Toc146895971)

[RMAP Simulation Framework (VHDL) 15](#_Toc146895972)

[RMAP Protected Types & Records 15](#_Toc146895973)

[Core Example Design (Xilinx Vivado) 16](#_Toc146895974)

[Getting Started **Error! Bookmark not defined.**](#_Toc146895975)

[Behavioural Simulation 19](#_Toc146895976)

[Figure 1: Simplified Core Architecture 3](#_Toc146895977)

[Figure 2: Command Controller FSM 4](#_Toc146895978)

[Figure 3: Standard RMAP Write Command Format 5](#_Toc146895979)

[Figure 4: IP -Write Command Submission Format 5](#_Toc146895980)

[Figure 5: Byte Submission Handshake 6](#_Toc146895981)

[Figure 6: Tx Data Interface waveform 6](#_Toc146895982)

[Figure 7: Reply Controller FSM 7](#_Toc146895983)

[Figure 8: RMAP Read-Reply Byte Format 7](#_Toc146895984)

[Figure 9: IP Read-Reply Output Format 8](#_Toc146895985)

[Figure 10: Rx Data Ready/Valid Handshake 8](#_Toc146895986)

[Figure 11:SpaceWire IO Configuration 10](#_Toc146895987)

[Figure 12: Initiator Generics Example Settings 11](#_Toc146895988)

[Figure 13: IP DDR Reg / IO Connections 15](#_Toc146895989)

[Figure 14: SpaceWire Debug Signal Declaration 16](#_Toc146895990)

[Figure 15: SpaceWire Debug in Testbench 16](#_Toc146895991)

[Figure 16: SpaceWire Debug Sim Waveform 16](#_Toc146895992)

[Figure 17: Testbench Architecture 17](#_Toc146895993)

[Figure 18: IP Folder Structure 18](#_Toc146895994)

[Figure 19: Example Design & Simulation Sources (Vivado) **Error! Bookmark not defined.**](#_Toc146895995)

[Figure 20: Simulation Finished OKAY **Error! Bookmark not defined.**](#_Toc146895996)

# Overview

The 4Links RMAP Initiator IP Core implements a fully open-source FPGA IP core for posting RMAP commands to a SpaceWire network. The core is pre-configured to include the 4Links SpaceWire IP Codec IP Core as standard. All supporting VHDL libraries and IP are open-sourced and accessible for modification.

The core complies with the behaviour for an RMAP initiator as outlined in ECSS-E-ST-50-52C. The IP core is intended to be used with the 4Links RMAP Target and 4Links RMAP Router IP Core(s). All of which are open-source, free to use, SpaceWire IP provided by 4Links.

# Core Architecture

The RMAP initiator is comprised of two distinct parts, a Command Controller (Tx) and a Reply Controller (Rx). The Rx and Tx sides of the core function independently of one another. Allowing a command to be posted as a reply is received and vice-versa. Each controller has its own CRC logic for implementing the RMAP compliant CRC algorithm. CRC bytes are automatically calculated and checked by the initiator IP core. CRC errors are reported to the user application as they occur.

Figure 1 shows a simplified diagram of the IP core architecture. Note the 4Links SpaceWire CoDec IP core is already instantiated within the RMAP Initiator.

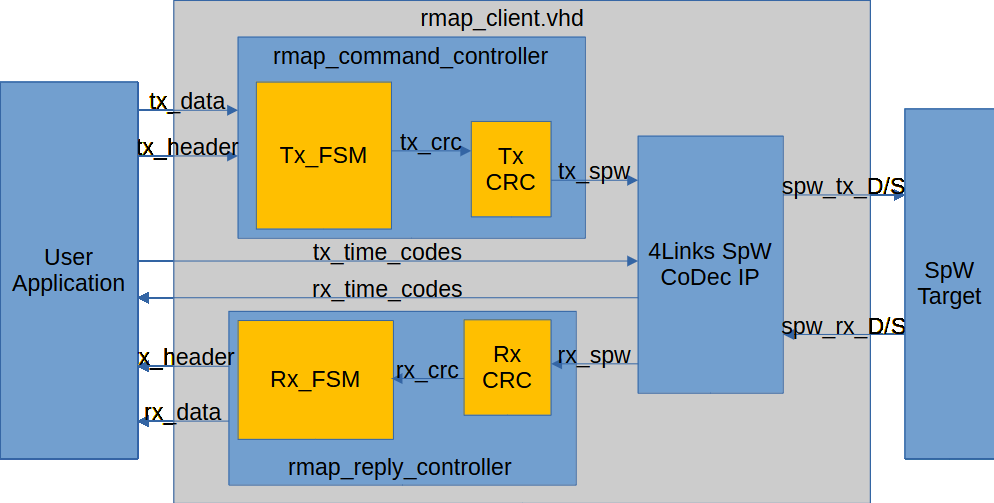


Figure : Simplified Core Architecture

## RMAP Command Controller

RMAP header and data bytes are checked before being sent across the SpaceWire link. Minimal buffering is used to prevent SpaceWire link hang-ups when using Wormhole routing. The CRC calculator performs CRC calculations automatically as data is streamed across the SpaceWire link. The CRC implementation meets the requirements as laid out by ECSS-E-ST-50-52C. Checking includes early EEP/EOP detection.

CRC Bytes and EOPs are automatically appended to the RMAP command by the Command Controller. The FSM for the command controller contains an error handler for unexpected behaviours. When in an error state, the error must be acknowledged by the host application before the Command Controller will recover to its initialized state. A list of error codes is presented in this document.

A diagram of a computer error

Description automatically generated

Figure : Command Controller FSM

In the case of an early EEP/EOP error state, if path addresses have already been submitted to the SpaceWire link, the EEP/EOP will be pushed and transmitted across the SpaceWire link. This ensures that any routers in the SpaceWire network will not lock up.

### Example Command Byte Structure

Figure 3 contains the standard format for an RMAP write command. This is lifted from the ECSS-E-ST-50-52C RMAP specification.

A close-up of a document

Description automatically generated

Figure 3: Standard RMAP Write Command Format

Below Figure 4 shows the submission order for an RMAP write command to the 4Links IP. Note that no Header/Data CRCs or EOP are required.

If using reply addressing and the number of reply addresses is not divisible by 4. Then the user application must pad the remaining reply address bytes with 0x00. Up to 12 reply addresses can be submitted.

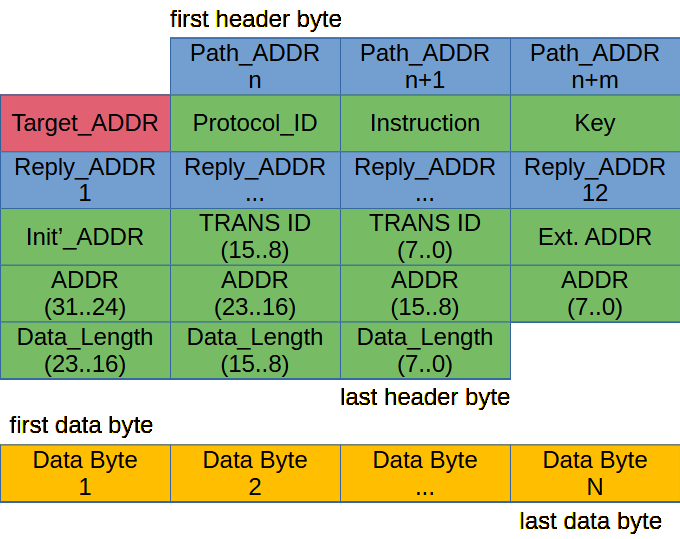


Figure 4: IP -Write Command Submission Format

### Header & Data Interface

Submitting bytes to the RMAP Command interface is achieved using a ready/valid handshake. The ready signal is asserted by the IP core when the IP core is ready to receive data. The valid signal is asserted by the user application when valid data is presented on the interface. Data submission occurs only when both ready and valid signal are asserted. These actions are aligned with the rising edge of the positive system clock. Figure 5 shows an example of this handshake where the byte “0x6C” is submitted to the header interface.

A screenshot of a computer

Description automatically generated

Figure : Byte Submission Handshake

When sending a command with a data payload. Once all header bytes have been sent, the data interface is used to transmit data bytes across the SpaceWire link (see Figure 6). As with the header bytes, a ready/valid handshake is used to load new data bytes onto the interface. Data bytes are checked for EEP/EOPs before transmission across the SpaceWire link.

The number of data bytes to be transmitted is set by the “Data\_Length” field in the RMAP header. Injecting an EEP/EOP onto the data byte interface will force the RMAP initiator to abort the current RMAP frame and generate an error. The error must be acknowledged by the user application before the RMAP initiator can recover.

A screenshot of a video game

Description automatically generated

Figure : Tx Data Interface waveform

## RMAP Reply Controller

The (Rx) Reply Controller functions independently to the RMAP initiator (Tx) Command Controller. This allows for fully asynchronous Tx/Rx operation. A simplified state-machine diagram for the Reply Controller can be found in Figure 7 below.

A diagram of a network

Description automatically generated

Figure 7: Reply Controller FSM

### Example Reply Byte Structure

Figure 8 contains the RMAP Read Reply format. The RMAP Reply Controller complies with the RMAP ECSS-E-ST-50-52C specification.

A close-up of a computer screen

Description automatically generated

Figure 8: RMAP Read-Reply Byte Format

The RMAP Reply Controller Strips off CRC, EOP and Reserved bytes (if present) when pushing Header/Data bytes to the user application. See Figure 9 for an example output of a Read-Reply frame.

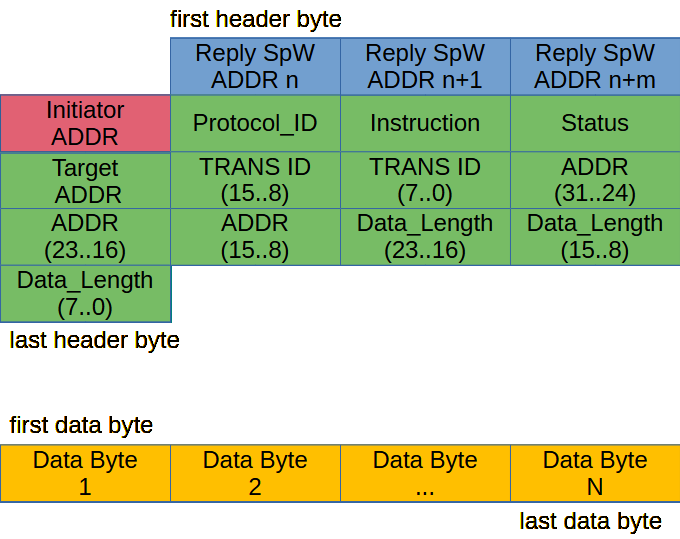


Figure 9: IP Read-Reply Output Format

Like the Command Controller, the Reply Controller uses separate header byte and data byte interfaces. Header and Data bytes are immediately pushed to the relevant interface. The RMAP Initiator IP does not perform any buffering of Reply bytes. If the user application interface is too slow, then Reply bytes could be dropped.

It is recommended that any application built around the Rx Header/Data interface uses its own dedicated FiFo Buffer logic to prevent data loss.

### Header & Data Interface

Like the Command Controller, Header and Data bytes have their own interfaces. Once all header bytes have been received, if the reply contains a data payload, then the data bytes will be pushed to the Data interface.

Both Header and Data interfaces use a Ready/Valid handshake for exchanging header/data bytes with a user application. The *valid* signal is asserted by the RMAP initiator IP when a valid byte is present on the interface. The user application should assert the *ready* signal when it is ready to receive data. Figure 10 shows an example where the data byte 0x03 is transmitted, on the rx\_data interface, to a user application.

A screenshot of a video game

Description automatically generated

Figure 10: Rx Data Ready/Valid Handshake

## RMAP SpaceWire CoDec

The RMAP Initiator is configured to use a generic wrapper version of the 4Links SpaceWire CoDec IP. It is recommended that “custom” mode is used. DDR registers/IO buffers can then be connected in the top-level design. This removes the need to modify the RMAP wrapper files, although that can be done if required.

The “single” and “diff” modes contain processes that model DDR inputs and LVDS IO buffers. Allowing for behavioural simulation without the need to modify the wrapper file.

If requiring a SpaceWire connection of >250Mb, please contact 4Links and we can provide a high-speed SpaceWire CoDec optimized for >350Mb operation. Achieving bitrates beyond 200Mb will greatly depend on your design size, architecture and FPGA technology.

## Design Notes

This section contains operational and design notes which may prove useful when debugging designs with the RMAP Initiator IP.

* When using Ready/Valid handshakes, note that the *Valid* signal direction will always align with the *Data* signal direction. The *Ready* signal will always oppose the Data signal direction.
* The RMAP Initiator Command Controller is designed to saturate the SpaceWire Tx link during operation. This minimizes the number of NULL packets on the SpaceWire link and improves the true bitrate of the transaction. During large transactions this can lead to temporary stalls of the SpaceWire CoDec whilst it receives new FCTs. These stalls do not reduce the speed of data transmission, they simply show that the link is fully saturated.
* The RMAP Initiator Reply Controller does not buffer any Reply bytes. Therefore, data can be lost if the user application does not read the existing header/data byte before a new one is received. A buffer is not provided by default to facilitate higher bitrates.
* Bitrate is tied to the Input clock speed of the IP, such that, for a 100Mb connection, a clock frequency of 100MHz is required. Maximum achievable bitrate will depend on your target FPGA technology and size of your application. By default, the RMAP Initiator IP comes with 4Links Standard Open-Source SpaceWire CoDec IP Core. Higher-speed options are available on request.

# Core Configuration

The IP core can be configured using VHDL Generics (parameters if using Verilog instantiation). RMAP initiator Generics are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **NAME** | **TYPE** | **RANGE** | **DESC** |
| g\_clock\_freq | REAL | >2\_000\_000.0 | Clock Frequency of SpW |
| g\_tx\_fifo\_size | INTEGER | >8 | Tx FiFo Depth (SpW) |
| g\_rx\_fifo\_size | INTEGER | >8 | Rx FiFo Depth (SpW) |
| g\_tx\_fifo\_interface | BOOLEAN | true/false | Generate Tx FiFo Core interface |
| g\_rx\_fifo\_interface | BOOLEAN | true/false | Generate Rx FiFo Core interface |
| g\_mode | STRING | “diff”, “single”, “custom” | IO mode of SpW IP |

More information about these configuration constants can be found in the 4Links SpaceWire CoDec IP User Guide. See below how to configure the SpaceWire IO in “custom” mode.

A diagram of a computer program

Description automatically generated

Figure :SpaceWire IO Configuration

# Core Interface

For compatibility, use the *rmap\_initiator\_top\_level* entity. Some ports on the *rmap\_initiator* entity use record types. This can cause issues if instantiating directly into a System Verilog/Verilog project. The Top-level wrapper entity breaks the record connections out to use std\_logic/vector IO. Using the *rmap\_initiator\_top\_level* is recommended to avoid toolchain issues when switching between multiple HDLs within a single project.

The Boolean-type interface ports on the SpaceWire CoDec IP are converted to use std\_logic type ports for better design integration.

## Parallel & FiFo Command/Reply Interface

The generics ***g\_tx\_fifo\_interface*** and ***g\_rx\_fifo\_interface*** allows selection between a parallel or FiFo-style interface for Tx/Rx header information. The FiFo-Style interface may be preferred when streaming RMAP command data to/from a processor and/or device memory. The FiFo interface also uses fewer device resources than the parallel interface.

The Parallel interface may be more useful for integration into RTL-Based RMAP controller designs and/or instances where RMAP header bytes remain constant between transactions.

When using parallel interface(s), the Command/Reply RMAP Data bytes still use a fifo-style write/read interface. The only difference is the header information for commands and replies.

A screen shot of a computer

Description automatically generated

Figure 12: Initiator Generics Example Settings

### Tx Parallel Interface

The Tx (command) parallel interface uses the ***tx\_header\_ready*** and ***tx\_header\_valid*** signals to submit header information to the command controller.

When the handshake is asserted, all configured header information is transmitted in the correct order across the SpaceWire link. Once all header information is sent, if required, the FiFo data interface will be used to submit required data bytes to the command controller.

The RMAP data port uses the same fifo-based submission regardless of core channel FIFO configuration.

### Submitting Path Addresses with Parallel Interfaces

When using the Parallel Tx interface, path addresses should be submitted on the ***tx\_logical\_address*** interface one at a time. Once an address with value 0xFE is submitted on the ***tx\_logical\_address***port, the Command controller will start transmitting all the configured header bytes as normal.

All header bytes should be configured before the address byte containing the default logical address (0xFE) is submitted to the command controller.

### Rx Parallel Interface

SpaceWire path addresses are pushed to the ***rx\_init\_log\_addr*** port until a logical address byte (0xFE) is received. Once the logical address byte is received, the rest of the header information will be obtained from the incoming RMAP reply.

The ***rx\_header ready*** and ***rx\_header\_valid*** handshake is used to read the full RMAP header. If receiving path address bytes, the path address bytes must be acknowledged before the next header byte can be read in, until the logical address 0xFE is received.

Once this occurs, the Reply Controller will read the full header and push to the parallel interface once it has been obtained. If required, after the header has been received and acknowledged, incoming data bytes will be pushed to the ***rx\_data***interface. This interface uses the same ready/valid handshake in both Fifo and Parallel IO operating modes.

*\*\* note \*\* acknowledgement refers to the assertion of the read/valid handshake between interface ports.*

## Clock & Reset Signals

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **Direction** | **TYPE** | **WIDTH** | **Desc** |
| clock | in | STD\_LOGIC | -- | Positive edge clock input |
| clock\_b | in | STD\_LOGIC | -- | Negative edge clock input |
| rst\_in | In | STD\_LOGIC | -- | Synchronous reset (active high) |
| enable | in | STD\_LOGIC | -- | Enable (active high) |

The *clock* and *clock\_b* signals should be produced using a MMCM/PLL/DLL. *clock\_b* should be 180° out of phase from *clock*. All interface transactions to/from the RMAP Ip Core occur with respect to the rising edge of *clock*.

## (Parallel) Command Header Ports

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **Direction** | **TYPE** | **WIDTH** | **Desc** |
| tx\_logical\_address | in | STD\_LOGIC\_VECTOR | 8 | Target logical address |
| tx\_protocol\_id | in | STD\_LOGIC\_VECTOR | 8 | RMAP Protocol ID |
| tx\_instruction | in | STD\_LOGIC\_VECTOR | 8 | Instruction Byte |
| tx\_Key | in | STD\_LOGIC\_VECTOR | 8 | Target Key |
| tx\_reply\_addresses | in | BYTE\_ARRAY | 12 | Reply addresses (up to 12) |
| tx\_init\_log\_addr | in | STD\_LOGIC\_VECTOR | 8 | Initiator Logical Address |
| tx\_Tranaction\_ID | in | STD\_LOGIC\_VECTOR | 16 | RMAP Transaction ID |
| tx\_Address | in | STD\_LOGIC\_VECTOR | 40 | Memory Address |
| tx\_Data\_Length | in | STD\_LOGIC\_VECTOR | 24 | Length of Data payload |
| Tx\_header\_valid | in | STD\_LOGIC | -- | Assert when header byte is valid |
| Tx\_header\_ready | out | STD\_LOGIC | -- | Asserted when logic is ready |

## Command Header Ports

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **Direction** | **TYPE** | **WIDTH** | **Desc** |
| Tx\_assert\_path | in | STD\_LOGIC | -- |  |
| Tx\_assert\_char | in | STD\_LOGIC | -- | Assert to send SpW Char |
| Tx\_header | in | STD\_LOGIC\_VECTOR | 8 | Header byte to transmit |
| Tx\_header\_valid | in | STD\_LOGIC | -- | Assert when header byte is valid |
| Tx\_header\_ready | out | STD\_LOGIC | -- | Asserted when logic is ready |

Tx\_assert\_path can be used to bypass a logical address and send it as a SpaceWire path address. Under normal operation, if a path address, the first SpaceWire address must be <32. If for whatever reason this is not the case, asserting “tx\_assert\_path” during the transaction will force the RMAP IP to send the address as a path address. Tx\_assert\_path is optional and is not required if the initial path address value is <32.

Tx\_assert\_char is used to send a SpaceWire control character. Asserting this high on a transaction will submit a SpaceWire character. The lower 2 bits of the Tx\_header/Tx\_data interfaces are used to assign the char value. “0x02 = EOP” etc.

## Command Data ports

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **Direction** | **TYPE** | **WIDTH** | **Desc** |
| Tx\_assert\_char | in | STD\_LOGIC | -- | Assert to send SpW Char |
| Tx\_data | in | STD\_LOGIC\_VECTOR | 8 | Data byte to transmit |
| Tx\_data\_valid | in | STD\_LOGIC | -- | Assert when data byte is valid |
| Tx\_data\_ready | out | STD\_LOGIC | -- | Asserted when logic is ready |

## Command Error ports

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **Direction** | **TYPE** | **WIDTH** | **Desc** |
| Tx\_error | out | STD\_LOGIC\_VECTOR | 8 | Output Error Code |
| Tx\_error\_valid | out | STD\_LOGIC | -- | Error Code is valid |
| Tx\_error\_ready | in | STD\_LOGIC | -- | Assert to read Error Code |

## (Parallel) Reply Header Ports

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **Direction** | **TYPE** | **WIDTH** | **Desc** |
| rx\_init\_log\_addr | out | STD\_LOGIC\_VECTOR | 8 | Initiator Logical Address |
| rx\_protocol\_id | out | STD\_LOGIC\_VECTOR | 8 | RMAP Protocol ID |
| rx\_instruction | out | STD\_LOGIC\_VECTOR | 8 | RMAP Instruction Byte |
| rx\_Status | out | STD\_LOGIC\_VECTOR | 8 | RMAP Status Byte |
| rx\_target\_log\_addr | out | STD\_LOGIC\_VECTOR | 8 | Target Logical Address |
| rx\_Tranaction\_ID | out | STD\_LOGIC\_VECTOR | 16 | RMAP Transaction ID |
| rx\_Data\_Length | out | STD\_LOGIC\_VECTOR | 24 | Length of Data Payload |
| crc\_good | out | STD\_LOGIC | -- | Asserted when CRC is OKAY |
| Rx\_header\_valid | out | STD\_LOGIC | -- | High when header byte to read |
| Rx\_header\_ready | in | STD\_LOGIC | -- | Assert when ready for header byte |

## Reply Header ports

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **Direction** | **TYPE** | **WIDTH** | **Desc** |
| Rx\_assert\_char | out | STD\_LOGIC | -- | Asserted when byte is SpW Char |
| Rx\_header | out | STD\_LOGIC\_VECTOR | 8 | Received header byte |
| Rx\_header\_valid | out | STD\_LOGIC | -- | High when header byte to read |
| Rx\_header\_ready | in | STD\_LOGIC | -- | Assert when ready for header byte |

Like Tx\_assert\_char, the *Rx\_assert\_char* port applies to both Header and Data bytes. This signal will be asserted when the current header/data byte to be read is a SpaceWire character.

EEPs/EOPs will not be pushed to the Header/Data interfaces, instead premature EOPs or any EEPs will generate a reply Error and must be acknowledged by the user application.

## Reply Data ports

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **Direction** | **TYPE** | **WIDTH** | **Desc** |
| Rx\_assert\_char | out | STD\_LOGIC | -- | Asserted when byte is SpW Char |
| Rx\_data | out | STD\_LOGIC\_VECTOR | 8 | Received data byte |
| Rx\_data\_valid | out | STD\_LOGIC | -- | High when data byte to read |
| Rx\_datar\_ready | in | STD\_LOGIC | -- | Assert when ready for data byte |

## Reply Error ports

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **Direction** | **TYPE** | **WIDTH** | **Desc** |
| Rx\_error | out | STD\_LOGIC\_VECTOR | 8 | Output Error Code |
| Rx\_error\_valid | out | STD\_LOGIC | -- | Error Code is valid |
| Rx\_error\_ready | in | STD\_LOGIC | -- | Assert to read Error Code |

## SpaceWire Status ports

See 4Links SpaceWire CoDec IP User Guide.

## SpaceWire Time Code ports

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **Direction** | **TYPE** | **WIDTH** | **Desc** |
| Tx\_time | in | STD\_LOGIC\_VECTOR | 8 | Timecode to send |
| Tx\_time\_valid | in | STD\_LOGIC | -- | Assert to send timecode |
| Tx\_time\_ready | out | STD\_LOGIC | -- | Asserted when ready for timecode |
| Rx\_time | out | STD\_LOGIC\_VECTOR | 8 | Timecode received |
| Rx\_time\_valid | out | STD\_LOGIC | -- | Asserted when valid timecode |
| Rx\_time\_ready | in | STD\_LOGIC | -- | Assert to read timecode |

## SpaceWire IO Ports

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **Direction** | **TYPE** | **WIDTH** | **Desc** |
| DDR\_din\_r | out | STD\_LOGIC | -- | Data In Rising Edge Sampled |
| DDR\_din\_f | out | STD\_LOGIC | -- | Data in Falling Edge Sampled |
| DDR\_sin\_r | out | STD\_LOGIC | -- | Strobe in Rising Edge Sampled |
| DDR\_sin\_f | out | STD\_LOGIC | -- | Strobe in Falling Edge Sampled |
| SDR\_Dout | in | STD\_LOGIC | -- | Data output (Rising Edge Clock) |
| SDR\_Din | in | STD\_LOGIC | -- | Strobe output (Rising Edge Clock) |

DDR registers and IO buffers should be added outside of the RMAP initiator architecture (use g\_mode = “custom”). See SpaceWire IP User guide for more detail.

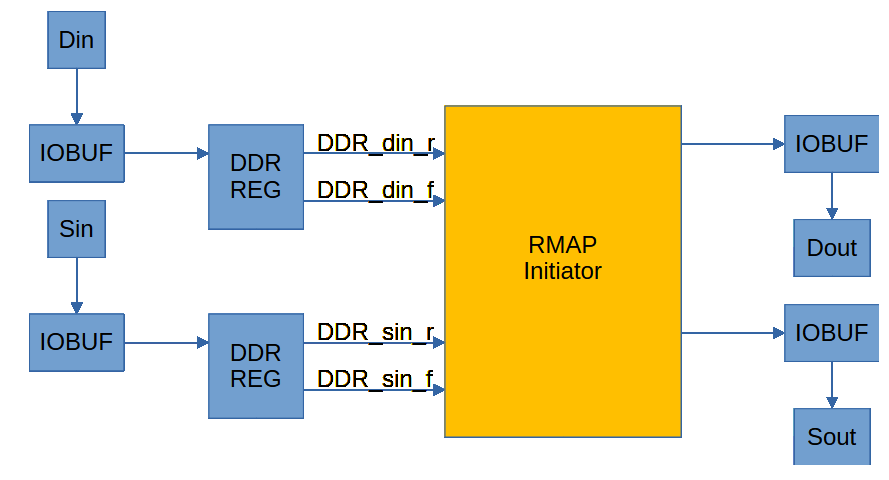


Figure 13: IP DDR Reg / IO Connections

# VHDL Packages

Packages covered here are used to build and compile the 4Links RMAP Client IP.

## rmap\_initiator\_lib

This package is used to create the RMAP Initiator IP. It contains records for simplifying interface connections between IP submodules. As well as useful types and functions. This package is required to build the RMAP initiator IP.

## SpaceWire\_Sim\_lib

This package contains simulation procedures, types and constants to create a verification environment for 4Links RMAP IP(s). This package relies on interfaces and data types created in the *rmap\_initiator\_lib* package. This package is used by the testbench *rmap\_initiator\_tb* to construct a simple verification environment.

# RMAP Simulation Framework (VHDL)

In order to ensure that designs using the 4Links RMAP Initiator IP are stable, a simulation environment with good functional coverage is required. Your approach to verification will depend on your specific application. With that in mind, the *SpaceWire\_Sim\_lib* package can be used to start developing your VHDL testbenches.

An example testbench can be found in the */Sim/* directory in the IP root folder.

## RMAP Protected Types & Records

RMAP Command frames are set using protected types. The protected type *t\_rmap\_command* on line 214 of *SpaceWire\_Sim\_lib* is used to load and modify data buffers containing RMAP command data.

Protected types are used to allow for safe access of data by multiple processes and to create a class-like approach to verification. Where multiple related signals can be grouped to speed up verification.

The *spw\_get\_poll* procedure uses operator overloading to implement three different versions of the procedure. These allow for signal, record and protected type data outputs. The *rmap\_init\_tb* testbench uses record types to display SpaceWire debug information (see Figure 15, Figure 15 and Figure 16).



Figure 14: SpaceWire Debug Signal Declaration

A screen shot of a computer program

Description automatically generated

Figure 15: SpaceWire Debug in Testbench

A screenshot of a computer screen

Description automatically generated

Figure 16: SpaceWire Debug Sim Waveform

# Core Example Design (Xilinx Vivado)

This section details bringing up and simulating the IP example design. The testbench consists of a RMAP Initiator IP and an RMAP Target IP both running at 100Mhz. The Initiator sends several read and write requests using both Path and Logical Addressing. For more detail on this, please see the testbench.

Both RMAP Target and RMAP Initiator use the 4Links SpaceWire CoDec IP.

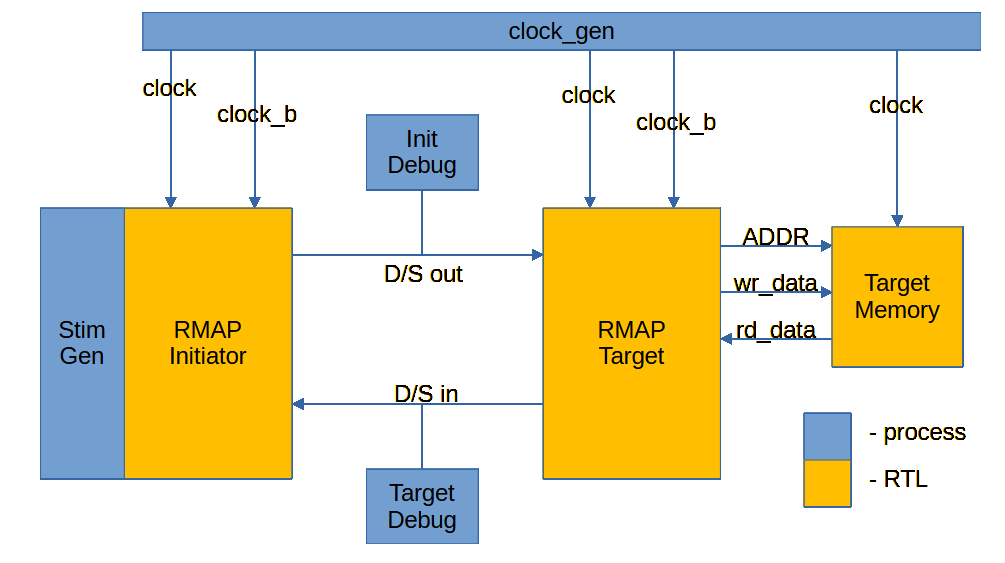


Figure 17: Testbench Architecture

Figure 17 shows the testbench architecture. Clock generation is split between two processes which generate the positive and negative system clocks. All transactions are aligned to the rising-edge of the positive system clock.

All test stimuli are driven through the RMAP Initiator IP. The Debug processes are used to breakout raw SpaceWire D/S signals and check the underlying SpaceWire transactions.

## Adding the Project Files

Before copying files into your project, make sure that your VHDL version is set for 2008 or higher.

A screenshot of a computer

Description automatically generated

Figure : Common Packages Directory

* Import sub-folders from ***common\_packages*** into your project. Packages should be added to a library which corresponds to the sub-folder name. i.e files in ***rmap*** should be added to a VHDL library called “rmap”.
* Import ***ip4l\_context.vhd*** into your project. This creates the context-clauses used for building the IP.

A screenshot of a computer

Description automatically generated

Figure : SpaceWire directory

* Import all files from the ***/spacewire/RTL/IP\_Sub\_Modules*** directory.
* Import ***spw\_wrap\_top\_level.vhd*** and ***spw\_wrap\_top\_level\_std.vhd*** from the ***/spacewire/RTL/IP\_Wrappers*** directory.

A screenshot of a computer

Description automatically generated

Figure : RMAP Directory

* From ***/rmap/RTL/rmap\_initiator*** add the ***IP\_Sub\_Modules*** & ***IP\_Wrapper\_Files*** directories to your project.
* From ***/rmap/RTL/rmap\_target*** add the three .VHD files. Do not add any ***Legacy*** directories to your project.
* Finally, add the ***/rmap/Sim/*** directory to your project. The files here are for testbench only and do not contain synthesizable code.

Setting the ***rmap\_initiator\_top\_level.vhd*** entity as your top-level design entity should allow you to build and synthesize the RMAP initiator. The Initiator interface and SpW options can be configured through use of generics as described in this document.

Setting the ***rmap\_initiator\_tb.vhd*** as your top-level simulation entity should allow you to run a behavioural simulation of the RMAP Initiator and RMAP Target IP’s.

## Behavioural Simulation

Once the design has been loaded into your project, you can run a behavioural simulation. This requires that the ***rmap\_initiator\_top\_level.vhd*** has been set as your top-level simulation entity.

The Testbench simulates a large burst write and read to/from the RMAP target memory. Data read back from the target memory is checked against