ASTRON
Netherlands Institute for Radio Astronomy

Uthernet (UTH) Module Description

	Organisatie / Organization	Datum / Date
Auteur(s) / Author(s): Eric Kooistra	ASTRON	
Controle / Checked: Andre Gunst	ASTRON	
Goedkeuring / Approval: Andre Gunst	ASTRON	
Autorisatie / Authorisation: **Handtekening / Signature** Andre Gunst	ASTRON	

Distribution list:

Group:	Others:
Andre Gunst (AG, ASTRON) Eric Kooistra (EK, ASTRON) Daniel van der Schuur (DS, ASTRON) Harm-Jan Pepping (HJP, ASTRON)	Gijs Schoonderbeek (GS, ASTRON) Jonathan Hargreaves (JH, JIVE) Salvatore Pirrucci (SP, JIVE)

Document history:

Revision	Date	Author	Modification / Change
0.1	2011-11-16	Eric Kooistra	Draft.
0.2	2012-06-29	Eric Kooistra	Define preamble word instead of IDLE word. Added packet level flow control to uth_rx via XON.

Table of contents:

1 Introduction....4

1.1 Purpose....4
1.2 Scope....4

2 Interface, design and implementation of the components....5

2.1 uth_tx – Transmit an UTH packet....5
2.2 uth_rx – Receive an UTH packet....8

3 Verification of the components....11

Terminology:

CRC	Cyclic Redundancy Check
eop	end of packet
HDL	Hardware Description Language
IDLE	Idle word between frames
PHY	Physical interface
PRE	Preamble
RL	Ready Latency
RTL	Register Transfer Level
SFD	Start of Frame Delimiter
SISO	Source in Sink Out
sop	start of packet
SOSI	Source Out Sink In
TLEN	Type / Length
UTH	Uthernet

References:

1. “Uthernet Interface Specification”, ASTRON-SP-041, E. Kooistra
2. “Specification for module interfaces using VHDL records”, ASTRON-RP-380, E. Kooistra
3. “DP Streaming Module Description”, ASTRON-RP-382, E. Kooistra
4. “Data Path Interface Description”, ASTRON-RP-394, E. Kooistra

1 Introduction

1.1 Purpose

This document describes the Uthernet (UTH) module. The UTH module contains components to transmit or receive a block of data as payload in an Uthernet packet. The Uthernet packet structure is specified in [1]. The Uthernet interface is intended data transfer over for point-to-point links.

1.2 Scope

The UTH components use the SISO and SOSI streaming interface records that are defined in [2]. For more information on the streaming interface see [3].

This UTH module makes most of the DP packetizing components that were defined in [4] obsolete.

2 Interface, design and implementation of the components

2.1 uth_tx – Transmit an UTH packet

2.1.1 Interface

The uth_tx assembles a data block into an UTH packet as shown below in Figure 1 and Figure 2:

```
-----------------------
| data ...            |
-----------------------
  sop             eop
  val   val ...   val
  channel
```

Figure 1: uth_tx snk_in

```
...  -----------------------------------------------
     | PRE | SFD |TLEN | DATA ...            | CRC |  ...
     -----------------------------------------------
       sop                                   eop
       val   val  val    val ...             val
```

Figure 2: uth_tx src_out

The uth_tx interface parameters and ports are given in respectively Table 1 and Table 2.

Generic	Type	Description
g_data_w	natural	>= 1, <= *c_uth_data_max_w* = 256
g_nof_ch	natural	The channels are numbered from 0 TO *g_nof_ch*-1, each channel represents a type of UTH packet
g_typ_arr	t_natural_arr	Array of *g_nof_ch* TLEN type values
g_out_rl	natural	Only for architecture ‘rtl_delay’. Default use 6 to avoid instantiating the RL adapter else use the required source RL value.

Table 1: uth_tx parameters

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso	SISO: ready
snk_in	IN	t_dp_sosi	SOSI: data, valid, sop, eop, channel
src_in	IN	t_dp_siso	SISO: ready
src_out	OUT	t_dp_sosi	SOSI: data, valid, sop, eop

Table 2: uth_tx ports

2.1.2 Design

The input data is a block of data marked by *snk_in.sop* and *snk_in.eop*. If the PHY link supports data valid then the input block may have gaps where *snk_in.valid* is '0', else the *snk_in.valid* has to remain active during the block of data to ensure that *src_out_valid* remains active too. The minimum block size is 1 data word, whereby the *snk_in.valid*, *snk_in.sop* and *snk_in.eop* are then all active in the same clock cycle.

After the *snk_in.sop* the uth_tx first outputs the PRE (preamble), the SFD (start of frame delimiter) and the TLEN (type or length) words. The PRE word is marked by the *src_out.sop*. If the TLEN word represents a length then its value must match the number of input data words in a block. The TLEN length must be known and fixed, it is not dynamically derived from *snk_in.sop*, *snk_in.valid* and *snk_in.eop*, because that would require some block store and forward buffering. If the TLEN word represents a type, then it can have an arbitrary type value, because then receiving side will also know the fixed length that corresponds to that TLEN type. Typically the TLEN type value should be larger than the largest supported TLEN length value, to support both TLEN interpretations on the same PHY link.

Each *snk_in.channel* input channel gets a unique TLEN value. The requirement is that each input channel has a fixed number of data. The number of channels that is supported is set by *g_nof_ch*. The TLEN value is defined per channel via the parameter *g_typ_arr(snk_in.channel)*. The *g_typ_arr* value can indicate the payload data length or a packet type. For the uth_tx this is indifferent, because it just inserts the *g_typ_arr(snk_in.channel)* value at the TLEN field and determines the payload length using the *snk_in.valid* , *snk_in.sop* and *snk_in.eop*. Hence the uth_tx does not need to know whether the TLEN field is used as length or as type. The transmitted payload length must off course be the same as the payload length that the receiver side will expect for that TLEN value.

The valid *snk_in.data* block marked by the *snk_in.valid, snk_in.sop* and *snk_in.eop* is passed on as UTH payload via the UTH packet data field. The UTH payload is protected by a cyclic redundancy check (CRC). The CRC word is passed on via the CRC field. The CRC field is marked by the *src_out.eop*. The uth_tx passes the *snk_in.data* on to *src_out.data* without restriction, i.e. the *g_data_w* parameter is not used for that. The *g_data_w* parameter is used to select the appropriate CRC polynomial as defined in [1].

Originally the intention was to output preamble words between UTH packets to fill inter frame gaps if necessary. However to decrease the change of falsely detecting an SFD the uth_tx output now outputs IDLE words with all ‘1’-s (so 0xF..FFF) with *src_out.valid* is ‘0’ between UTH packets. The *src_out.valid*, *src_out.sop* and *src_out.eop* signals are available to ease subsequent UTH packet scheduling. They can be use on the PHY link if the PHY link interface supports them, but they do not have to be used on the PHY link. For more information on the PHY link see [1].

The uth_tx sink and source assume a ready latency of RL = 1. The uth_tx does support backpressure from the downstream sink via *src_in.ready*. The uth_tx does give backpressure to the upstream source via *snk_out.ready*. If the upstream source does not support the ready then this may be handled by placing an RL adapter or a FIFO in between [3]. If the downstream sink is always ready then uth_tx can continuously output UTH packets, i.e. *src_out.sop* directly after *src_out.eop*, if sufficient input data blocks are provided.

2.1.3 Implementation

The uth_tx has been implemented in two architectures:

- rtl_hold – using dp_hold_input
- rtl_delay – using dp_latency_adapter

The ‘rtl_delay’ architecture is based on a state machine that assembles the UTH packet while ignoring the *src_in.ready* followed by a RL adapter to correct for that omission. The RL adapter adapts from internal RL = 6 to source RL = *g_out_rl*. The ‘rtl_hold’ architecture is the preferred implementation. Figure 3 shows the block diagram of ‘rtl_hold’ that uses a dp_hold_input [3] component to handle the sink and the source with RL = 1. Note that *src_buf* is only used as buffer for *next_src_buf* and *pend_src_buf*, but not directly (this is typical for using dp_hold_input).

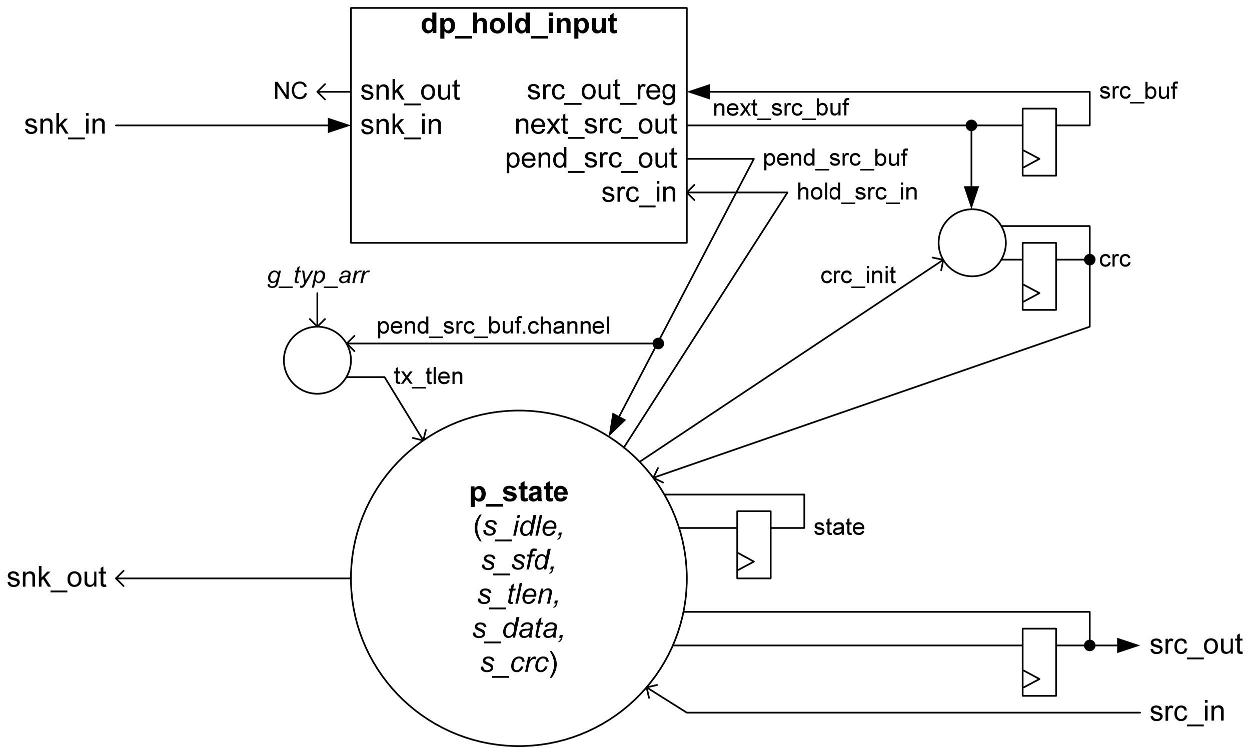

Figure 3: uth_tx(rtl_hold) implementation

2.2 uth_rx – Receive an UTH packet

2.2.1 Interface

The uth_rx disassembles a data block from an UTH packet as shown below in Figure 4 and Figure 5:

```
       -----------------------------------------------
  ...  | PRE | SFD |TLEN | DATA ...            | CRC |  ...
       -----------------------------------------------
```

Figure 4: uth_rx snk_in

```
                         -----------------------
                         | data ...            |
                         -----------------------
                           sop             eop
                           val    val ...  val
                                           err
                           channel
```

Figure 5: uth_rx src_out

The uth_rx interface parameters and ports are given in respectively Table 3 and Table 4.

Generic	Type	Description
g_data_w	natural	>= 1, <= *c_uth_data_max_w* = 256
g_len_max	natural	Defines the maximum number of data in the payload, used by the internal payload data counter.
g_nof_ch	natural	The channels are numbered from 0 TO *g_nof_ch*-1, each channel represents a type of UTH packet
g_typ_arr	t_natural_arr	Array of *g_nof_ch* TLEN type values
g_len_arr	t_natural_arr	Array of *g_nof_ch* TLEN length values
g_use_this_siso	boolean	Default use TRUE for best throughput performance. When FALSE then uth_rx does not need to control *snk_out* and it is ready when the downstream sink is ready, this may ease achieving timing closure. When TRUE then in addition uth_rx can also be ready when it is receiving inter frame gaps or the header to increase the throughput.
g_use_src_in	boolean	Only for architecture 'rtl_adapt'. Default use TRUE for backpressure support else use FALSE to avoid instantiating the RL adapter.

Table 3: uth_rx parameters

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso	SISO: ready
snk_in	IN	t_dp_sosi	SOSI: data
src_in	IN	t_dp_siso	SISO: ready
src_out	OUT	t_dp_sosi	SOSI: data, valid, sop, eop, channel, err

Table 4: uth_rx ports

2.2.2 Design

The UTH packet overhead words are stripped, only the payload data is output, indicated by the *src_out.valid*, *src_out.sop* and *src_out.eop*. The UTH packet starts with an PRE word. The *src_out.sop* is derived based on an PRE to SFD word transition in the *snk_in.data*. The number of payload data words is derived from the TLEN field. The last payload data word is held during the CRC word and output at the *src_out.eop*. The evaluation of the received CRC is passed on via the *src_out.err* field, where 0 indicates OK and 1 indicates an CRC error. The *src_out.err* field is valid at the *src_out.eop*.

The uth_rx can only receive a predefined set of different UTH packets. The different UTH packets are identified by their TLEN value. The number of TLEN values that uth_rx supports is set by *g_nof_ch*. The received TLEN word should match a value in *g_typ_arr*. The index that matches is used for *src_out.channel* and used to obtain the length of the payload from *g_len_arr(src_out.channel)*. All other UTH packets with unsupported TLEN values that are not in *g_typ_arr* will get discarded. While counting *g_len_arr(src_out.channel)* number of valid payload data words the uth_rx is insensitive to possible PRE / SFD words in the payload data. Hence the source output is always a full block of payload data. If for some reason the input UTH framing got corrupted, then this will reflect in the CRC so the *src_out.err* field will then report error. Another UTH packet may even get lost, but the UTH packet reception will recover on a next PRE / SFD detection.

Even without backpressure (so *src_in.ready* = '1') the *src_out.valid* will have a 1 cycle gap just before the *src_out.eop*, due to the processing of the CRC. Between received source output payloads there is a gap of 3 cycles between the *src_out.eop* and the next *src_out.sop* due to the 3 UTH packet header words (PRE, SFD and TLEN) that get removed from the UTH packet.

The uth_rx supports packet level flow control by flushing packets from the sink when src_in.xon is XOFF. Therefore the indication to the sink via snk_out.xon is always XON. The src_in.xon is examined when a sop arrived. During the reception of a packet src_in.xon has no effect.

2.2.3 Implementation

The uth_rx has been implemented in two architectures:

- rtl_hold – using dp_hold_input
- rtl_adapt – using dp_latency_adapter

The ‘rtl_adapt’ architecture is based on a state machine that disassembles the UTH packet while ignoring the *src_in.ready* followed by a RL adapter to correct for that omission. The RL adapter adapts from internal RL = 2 to source RL = 1. The ‘rtl_hold’ architecture is the preferred implementation, but using ‘rtl _adapt’ may also be applicable because it achieves about 0.5 % more throughput for random ready and it may achieve time closure more easily. Figure 6 shows the block diagram of ‘rtl_hold’ that uses a dp_hold_input [3] component to handle the sink and the source with RL = 1. Note that the state machine uses *next_src_buf*, whereas the state machine for uth_tx uses *pend_src_buf*, this is due to that uth_tx needs to insert data while uth_rx skips data.

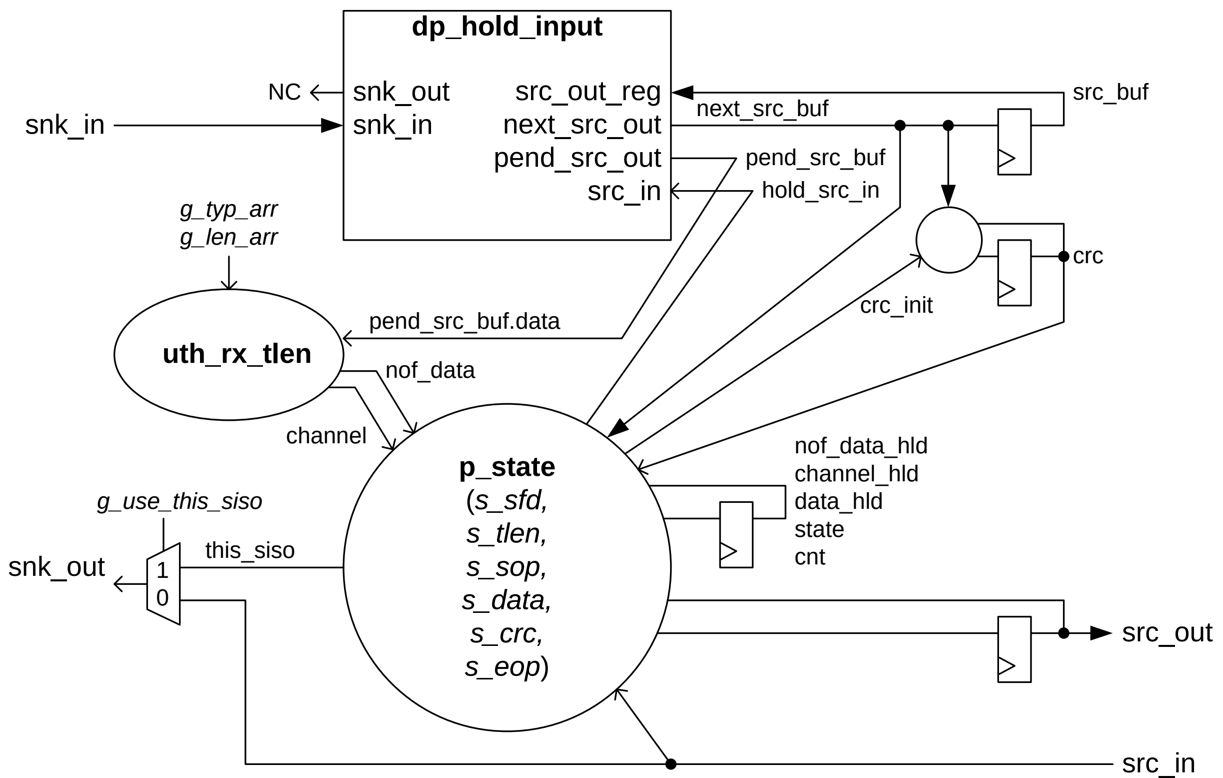

Figure 6: uth_rx(rtl_hold) implementation

3 Verification of the components

Figure 7 shows the tb_uth testbench that verifies uth_tx and uth_rx together. The two main stimuli for a streaming interface component are:

- upstream source enable
- downstream sink ready.

These stimuli can active, random or pulsed. The tb_uth uses the procedures described in [3] to generate blocks of data for the uth_tx sink and to verify that these blocks of data arrive properly at the uth_rx source. The *expected_uth_rx_data* signal in Figure 7 is used to verify that the test as run at all. When the stimuli have finished then signal *tb_end* stops the testbench clock to automatically stop the simulation.

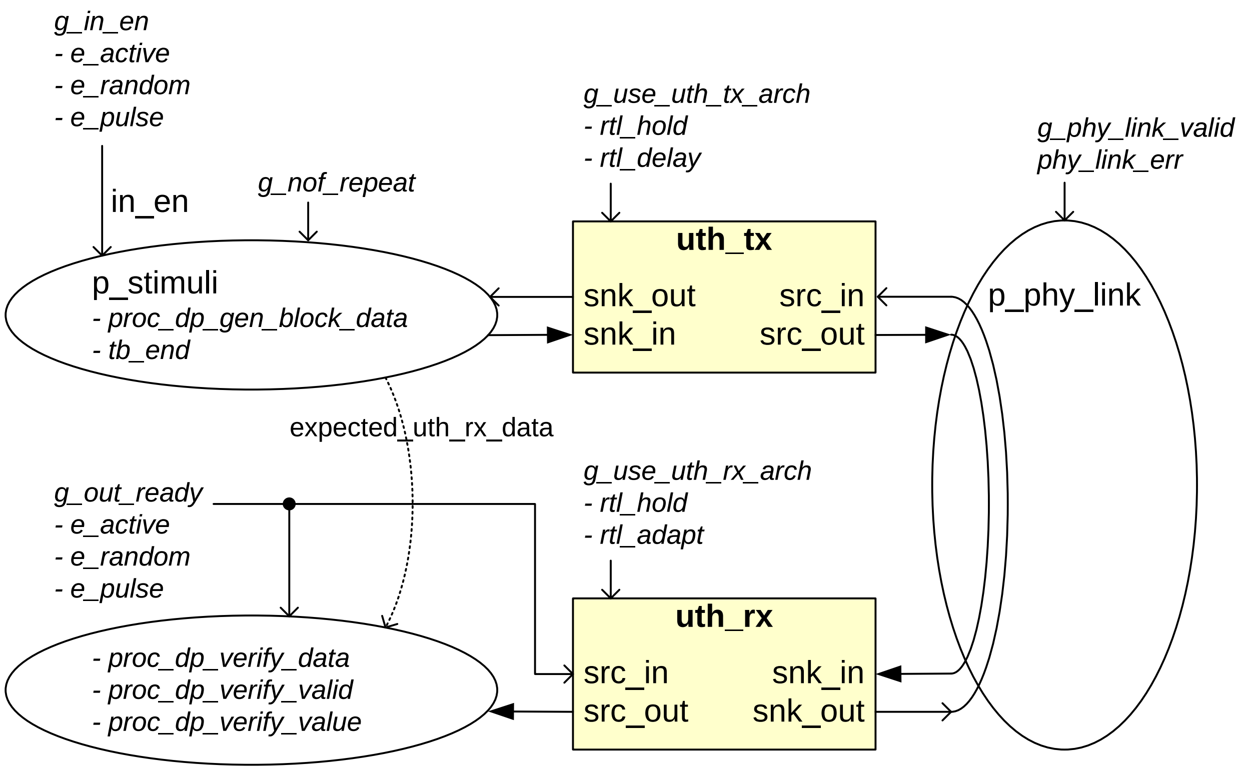

Figure 7: tb_uth testbench for uth_tx and uth_rx

The tb_tb_uth multi-testbench instantiates testbench tb_uth several times with different generic settings to perform regression tests on the uth_tx and uth_rx.