Userguide, Using TosNet in a project

This document will give an introduction to the general idea of the TosNet protocol and network, and describe how to use it in a project.
For further information, have a look at (1), (2) and (3).

1 Main idea

The idea of TosNet is to make it easy to distribute various data between a number of FPGAs. This is done over connections based on optical Toslink components, allowing noise-immune data transmission over up to 5 m. As each optical fiber can only be used for one-way communication, two fibers are needed for each FPGA, one for transmitting, and one for receiving. The FPGAs are connected in a ring, as can be seen in figure 1.

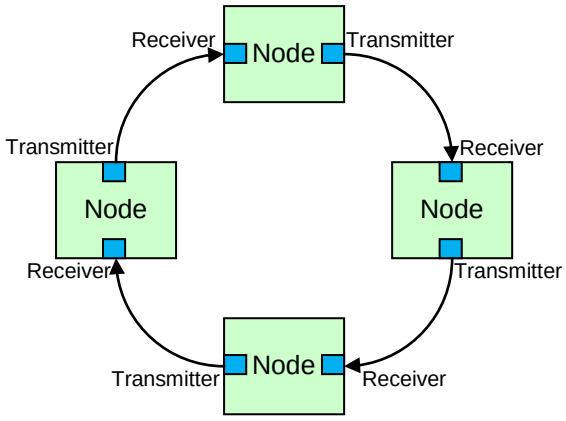

Figure 1: An example of a TosNet network.

The communication between nodes is master-slave based, and done in cycles using a shared memory block, realized with a BlockRAM in each node. The contents of this BlockRAM is then automatically mirrored between all the nodes in the network, so that when one node puts data in its own memory block, this data will be available from the same address in any other node in the network one cycle later.

1.1 Memory model

The memory block is segmented so that each node in the network has its own segment, which is further divided into eight registers, each 32 bytes large. These eight registers can be individually enabled or disabled, so that the amount of data transferred each cycle can be matched to what is actually necessary.
Of the 32 bytes available in each register, 16 bytes are “in-blocks”, used for communication from the master node to the slave node in question (the master node is the only node which should write to these, and the targeted slave node is the only one able to read them) and the other 16 bytes are “out-blocks” used for communication from the slave node to the rest of the network (only the slave node in question has write-access, but all nodes in the network can read the data)[1]. An example can be seen in section 6.

1.2 Memory interface

The interface to the BlockRAM is a standard Xilinx BlockRAM interface, consisting of an address-bus (10 bits wide), a data in and a data out bus (each 32 bits wide), a write-enable signal, and a clock signal (see (4) for more information).

[1] Generally, register access in TosNet is not enforced, so it is possible to read and write from/to all registers even though the particular node is not specified as having access. However, unless you know *exactly* what you are doing, this will most probably result in strange network behaviour, and is not at all encouraged.

The memory block is double-buffered, to avoid mixing data from one cycle with data from another cycle. This is controlled using two commit signals, to signal when all data from the current cycle have been read or written. Once a commit has been signaled, the committed parts of the registers should not be accessed until the next cycle has begun, which is signaled by a synchronization signal from the TosNet core.

1.3 Asynchronous channel

In addition to the isochronous channel, keeping the BlockRAM updated, an asynchronous channel is also available. This allows the transmission of up to 12 bytes per cycle, either as a broadcast from the master node to all slave nodes, or as a transmission to a single, specified slave node. In the latter case, the targeted slave node can reply with up to 12 bytes of asynchronous data during the same cycle. This also means that a slave node can only transmit asynchronous data in a given cycle if the master has sent asynchronous data to it during the cycle.
The interface to the asynchronous channel consists of two FIFO buffers, one for sending and one for receiving data. In both cases one of the ports (the write port for the send buffer, and the read port for the receive buffer) is simply routed through to the top of the TosNet component, and are available as a direct interface.
The data bus is 38 bits wide in both cases, with bits 31-0 used for data, bits 35-32 used for the address of the targeted slave node (in the case of a broadcast, these bits are simply all 0), and bits 37-36 used to specify how many of the bytes in the data field are valid. Further details can be seen in section 5.

2 Prerequisites

To use the TosNet protocol, the following is needed:

- Xilinx ISE 9 or newer (other versions might work too, but have not been tested)
- Working Spartan3/6 board with TosNet network interface card
- VHDL files:
    - tosnet.vhd
    - tal_top.vhd
    - tdl_top.vhd
    - tdl_app_master.vhd
    - tdl_app_net.vhd
    - tdl_app_reg.vhd
    - tdl_app_sync.vhd
    - commandpack.vhd
    - crcgen.vhd
    - crcpack.vhd
    - tpl_tx.vhd
    - tpl_rx.vhd
    - lfsr.vhd
    - enc_8b10b.vhd
    - dec_8b10b.vhd
- CoreGen XCO files: (use the appropriate files for your device, or generate new ones)
    - data_reg.xco
    - network_register.xco
    - async_fifo.xco

3 How to instantiate the TosNet VHDL module in an existing design

To use the network implementation for a design, the following steps should be followed:

1. Add all the *.vhd* and *.xco* files to the project.

2. Add the following to the top-level entity:
```
signal sig_in      : in STD_LOGIC_VECTOR;
signal sig_out     : inout STD_LOGIC_VECTOR;
```
3. Connect *sig_in* to the FPGA pin connected to the Toslink receiver, and *sig_out* to the FPGA pin connected to the Toslink transmitter. Set both of these pins to use LVCMOS33.
4. Add the component definition in list1 to the behavioral description of the top-level module.
5. Instantiate the module with the name *tosnet_inst* directly under the top-level module (see list2 for an example of this).
6. Connect the existing design to the needed signals.

4 Configuration signals

This section describes the necessary configuration variables of the TosNet node.

4.1 node_id<3:0>

The node needs to have an id assigned. This needs to be unique in the network, that is, no two nodes in the same network may use the same id. During network startup, the node with the lowest id is assigned as master node. It is recommended to set the intended master node to have id 1, to allow for the possibility of easily inserting a different master node (with id 0) during debug.

4.2 reg_enable<7:0>

This constant describes the enabled registers in the node. Each bit corresponds to a register (bit0 is register 1, bit1 is register 2, etc), and this is enabled when the bit is '1', and disabled when the bit is '0'. To enable register 1 and 3 for instance, set *reg_enable* to "00000101".

4.3 watchdog_threshold<17:0>

This constant is used to specify the amount of time with no network activity the node should wait until restarting the network setup procedure. The threshold is measured in 1.25 MHz clock cycles, that is, 800 ns. A value of "10000000000000000" thus equates to a watchdog threshold of about 104 ms (131072 * 800 ns). It should not be set lower than the cycle time for the used network.

4.4 max_skipped_writes<15:0> / max_skipped_reads<15:0>

These constants are used to specify the maximum number of consecutive network cycles, where the values of the write, respectively the read registers, are not used or accessed by the external application. This can be either due to data not being committed, or due to transmission errors. If the threshold is passed, the *system_halt* signal is asserted. If the functionality is not needed, the thresholds can be set to 0, which causes the counters to be ignored.

4.5 disable_slave

If set to high, the node will only be able to act as a master node. This can be used to reduce the amount of resources used for implementation. Removing the slave functionality is not recommended though, as it will reduce debugging possibilities.

4.6 disable_master

If set to high, the node will only be able to act as a slave node. This can be used to dramatically reduce the amount of resources used for implementation.

4.7 disable_async

If set to high, the node will not contain functionality for sending and receiving on the asynchronous channel. It will still be able to forward data on the asynchronous channel, so it is possible to mix nodes with the asynchronous channel enabled and disabled in the same network.

5 Interface signals

This section describes the interface signals used for communicating with the TosNet component. The *data_reg_* signals are directly connected to the BlockRAM, and thus work just like any other memory.

5.1 data_reg_addr<9:0> (input)

This is the address bus of the memory block, and is organized as can be seen in figure 2.

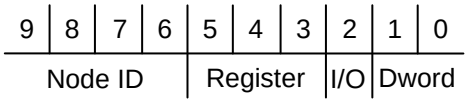

Figure 2: How to address into the memory block.

The four different parts have the following meaning:

- *Node ID*
  The ID of the node to target.
- *Register*
  The register of the specified node to target.
- *I/O*
  '0' targets out-blocks (generally used for output from a slave node - writable from owner, readable from all), '1' targets in-blocks (generally used for input to a slave node - writable from master, readable from owner).
- *Dword*
  Specifies which part of the register to target. Each address targets 32 bits, and as each in- or out-block of a register is 128 bits large, there will be four equally-sized parts.

So if for example the bitstring "0111100100" is put on the address bus, it will target the first 32 bits of the in-part of register "100" in the node with id "0111". Also see the example in section 6.

5.2 data_reg_data_in<31:0> (input) / data_reg_data_out<31:0> (output) / data_reg_we<0:0> (input) / data_reg_clk (input)

These are the rest of the BlockRAM interface signals for the memory block. To write data to the block, do the following:

1. The data to write is put on *data_reg_data_in*, the write enable signal, *data_reg_we*, is pulled high, and the address to write to is put on *data_reg_addr*.
2. *data_reg_clk* is pulled high, which performs the write.

To read data from the block, do the following:

1. The write enable signal, *data_reg_we*, is pulled low, and the address to read from is put on *data_reg_addr*.
2. *data_reg_clk* is pulled high, which performs the read.
3. The read data is now available on *data_reg_data_out*.

5.3 commit_write (input)

This signal is used to commit data in the out-blocks. Once all data for a cycle has been written, *commit_write* needs to be pulled high and held high for at least 20 ns (the TosNet core looks for a transition from '0' to '1').

5.4 commit_read (input)

This signal is used to commit data in the in-blocks. Once all data for a cycle has been read, *commit_read* needs to be pulled high and held high for at least 20 ns (the TosNet core looks for a transition from '0' to '1').

5.5 sync_strobe (output)

This signal is used to signal the end of a cycle, and the beginning of the next one.

5.6 online (output)

This signal is high when the network setup has been successfully completed, and the network is online, low otherwise.

5.7 is_master (output)

This signal is high if the node is currently functioning as master in the TosNet network, low otherwise.

5.8 packet_error (output)

This signal is high when an error is discovered in the transmission, and will be held high until a packet without errors is discovered.

5.9 system_halt (output)

This signal is pulled high if the maximum amount of skipped registers (as specified in *max_skipped_writes* and *max_skipped_reads*) is exceeded, low otherwise.

5.10 sig_in (input) / sig_out (output)

These are the signals from and to the Toslink transmitter components.

5.11 clk_50M (input)

This is the 50 MHz clock signal required by the TosNet core.

5.12 reset (input)

This is the active-high reset signal for the TosNet core.

5.13 reset_counter (output) / packet_counter (output) / error_counter (output)

These counters count (respectively) the number of resets performed, packets transmitted, and erroneous packets detected, since power-up.

5.14 async_ (input / output)

These signals are used for the asynchronous channel, and interface directly to the asynchronous FIFO buffers.
This data bus is 38 bits wide, and is organized as can be seen in figure 2.

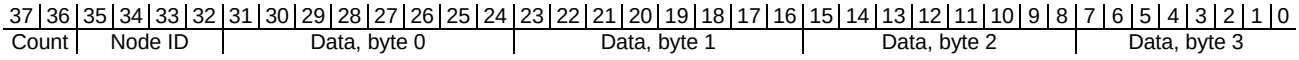

Figure 3: The structure of the data bus for the asynchronous FIFO buffers.

The different parts have the following meaning:

- *Data, byte 0-3*
  The data. All four bytes do not necessarily contain valid data.
- *Node ID*
  The id of the slave node that this data is meant for (in the case of the master node sending data to a slave node), the id of the slave node where this data came from (in the case of master receiving data from a slave node), or "0000" (in the case of a broadcast from the master to all nodes).
- *Count*
  Specifies the index of the last valid byte in the current frame. Byte 0 will thus always be valid.

6 Examples

As an example consider the following network setup:

- Node 1: Master, register 0 enabled
- Node 2: Slave, register 0 and 1 enabled
- Node 3: Slave, register 1 enabled
- Node 14: Slave, register 7 enabled

The register addresses are then as follows:

- Node 1:
  - Out register: 0x040 – 0x043
  - In register: 0x044 – 0x047
- Node 2:
  - Out registers: 0x080 – 0x083 and 0x088 – 0x08b
  - In registers: 0x084 – 0x087 and 0x08c – 0x08f
- Node 3:
  - Out register: 0x0c8 – 0x0cb
  - In register: 0x0cc – 0x0cf
- Node 14:
  - Out register: 0x3f8 – 0x3fb
  - In register: 0x3fc – 0x3ff

The access rights will be as shown in table 1.

Registers	Node 1	Node 2	Node 3	Node 14
Node 1 in	R/W	-	-	-
Node 1 out	R/W	R	R	R
Node 2 in	W	R	-	-
Node 2 out	R	R/W	R	R
Node 3 in	W	-	R	-
Node 3 out	R	R	R/W	R
Node 14 in	W	-	-	R
Node 14 out	R	R	R	R/W

Table 1: The access rights for the example network.

7 TosNet gateways

There are a number of different ways to connect a TosNet network to a higher-level system, both with regard to hardware and software. The recommended methods are described in the following sections. Currently, the gateways only support the isochronous channel, and not the asynchronous.

7.1 Ethernet gateway, generic protocol

This is the most generic and flexible method. It works by interfacing a Digi Connect ME 9210 ARM9-based, Ethernet-enabled embedded microcontroller module to the TosNet masternode, over a serial peripheral interface (SPI). A high-level system can then read and write from the TosNet network using a simple protocol.
The protocol uses a client-server scheme, where the gateway module works as the server, and responds to packets from clients over a TCP/IP connection. Each packet can contain a read request, a write request, or both.
A packet always contains a 32 bit command part. If a write is requested, the packet also needs to contain a 32 bit payload after the command part, specifying the data to write. If a read is requested, the gateway will respond with a packet containing a 32 bit payload, otherwise no response will be made.
The command part should be formatted as can be seen in table 2.

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
-	-	Use skip counter In	Commit In Pre	Do read	Commit In Post	Read Address										-	-	Use skip counter Out	-	Do write	Commit Out Post	Write Address									

Table 2: The format of the command packet, showing the usage of the various bits.

- *Use skip counter In*
  If set to ‘1’, the skip counter for the In registers is activated. If the In registers are not committed once every 1024 cycles (adjustable in the instantiation of the TosNet core), the network is halted. If set to ‘0’ the skip counter is deactivated.
- *Commit In Pre*
  If set to ‘1’, the in registers will be committed before the commands in this packet have been performed.
- *Do read*
  If set to ‘1’, this packet will perform a read from the address specified in *Read address*. If set to ‘0’, no read is performed and *Read address* is ignored.
- *Commit In Post*
  If set to ‘1’, the in registers will be committed after the commands in this packet have been performed.
- *Read address*
  The address to read from.
- *Use skip counter Out*
  If set to ‘1’, the skip counter for the Out registers is activated. If the Out registers are not committed once every 1024 cycles (adjustable in the instantiation of the TosNet core), the network is halted. If set to ‘0’ the skip counter is deactivated.
- *Do write*
  If set to ‘1’, this packet will perform a write from the address specified in *Write address*. If set to ‘0’, no write is performed and *Write address* is ignored
- *Commit Out Post*

If set to '1', the out registers will be committed after the commands in this packet have been performed.

- *Write address*
  The address to write to

If *Do write* is set, the 32 bit command needs to be followed by 32 bits of payload, containing the data to write. If no payload is received, *Do write* and *Commit out* (if set) are ignored.
If *Do read* is set, the gateway will reply with a 32 bit data packet containing the read data. If *Do read* is not set, no reply will be made.

7.2 Ethernet gateway, seDSSP protocol

It is also possible to use the Distributed Software Services Protocol (DSSP) used in for instance Microsoft Robotics Developer Studio, to communicate with the TosNet Ethernet gateway. Communication will be done through XML-serialized SOAP envelopes over an HTTP transport. This is made possible by simple embedded DSSP (seDSSP), which is a C++ implementation of part of the DSSP protocol. For more information, consult the seDSSP documentation.

7.3 USB/UART gateway

By using a board with a USB to UART bridge converter chip (currently a CP2101 chip is used), it is possible to use a standard serial COM port (through USB) for communicating with TosNet. The interface consists of a few ASCII formatted commands.
When attaching the gateway to a computer, the driver for the converter chip should auto-install, and a new COM port should be made available instantly. This port can then be used through for instance HyperTerminal, PuTTY or any other application able to access a COM port.
It should be opened with the settings specified in table 3.

Speed	115200
Data bits	8
Stop bits	1
Parity	None
Flow control	None

Table 3: The settings for using the USB/UART gateway.

The commands usable are listed below. For all commands and responses, lower-case letters are part of the command (and should be used in lower-case), while italic, upper-case letters are placeholders for a hexadecimal number, with a described functionality. It should be noted that all hexadecimal numbers should also use lower-case letters (that is, a-f instead of A-F).

- *Read*
  Reads the value of a register.

```
Command:   rNRI
Response:  XXXXXXXX
```

  The node index, *N*, should be a number in the range 0-f, whereas the register index, *R*, and sub-index, *I*, should be numbers in the range 0-7. Sub-indexes 0-3 indicate the out-blocks of a specific register, while sub-indexes 4-7 indicate the in-blocks. The response, *XXXXXXXX*, will consist of an 8-digit lower-case hexadecimal number.

- *Write*
  Writes a value to a register.

```
Command:   wNRI XXXXXXXX
Response:
```

  The node index, *N*, should be a number in the range 0-f, whereas the register index, *R*, and sub-index, *I*, should be numbers in the range 0-7. Sub-indexes 0-3 indicate the out-blocks of a

specific register, while sub-indexes 4-7 indicate the in-blocks. The value to write, *XXXXXXXX*, should consist of an 8-digit lower-case hexadecimal number, and must be preceded by a space character. No response is made.

- *Commit out*
  Commits the out-blocks.

```
Command:   t
Response:
```

  No response is made.

- *Commit in*
  Commits the in-blocks.

```
Command:   c
Response:
```

  No response is made.

7.4 PCI Express gateway

A PCI Express-based gateway also exists. This uses a Spartan3 PCI Express Starter Kit from Xilinx, and provides direct access to the shared memory block for user applications running on a PC on both Windows and Linux. A device driver enables user applications to simply map the shared memory block to user-space, and thus to access the block through for instance an integer array. It is possible to achieve write speeds of around 30 MB/s, and read speeds of approximately 2.5 MB/s. Consult the PCI Express gateway documentation for more information on this.

8 Code listings

```
component tosnet is
    Generic ( disable_slave          : STD_LOGIC := '0';
              disable_master         : STD_LOGIC := '0';
              disable_async          : STD_LOGIC := '0');
    Port(     sig_in                 : in  STD_LOGIC;
              sig_out                : inout STD_LOGIC;
              clk_50M                : in  STD_LOGIC;
              reset                  : in  STD_LOGIC;
              sync_strobe            : out STD_LOGIC;
              online                 : out STD_LOGIC;
              is_master              : out STD_LOGIC;
              packet_error           : out STD_LOGIC;
              system_halt            : out STD_LOGIC;
              node_id                : in  STD_LOGIC_VECTOR(3 downto 0);
              reg_enable             : in  STD_LOGIC_VECTOR(7 downto 0);
              watchdog_threshold     : in  STD_LOGIC_VECTOR(17 downto 0);
              max_skipped_writes     : in  STD_LOGIC_VECTOR(15 downto 0);
              max_skipped_reads      : in  STD_LOGIC_VECTOR(15 downto 0);
              data_reg_addr          : in  STD_LOGIC_VECTOR(9 downto 0);
              data_reg_data_in       : in  STD_LOGIC_VECTOR(31 downto 0);
              data_reg_data_out      : out STD_LOGIC_VECTOR(31 downto 0);
              data_reg_clk           : in  STD_LOGIC;
              data_reg_we            : in  STD_LOGIC_VECTOR(0 downto 0);
              commit_write           : in  STD_LOGIC;
              commit_read            : in  STD_LOGIC;
              reset_counter          : out STD_LOGIC_VECTOR(31 downto 0);
              packet_counter         : out STD_LOGIC_VECTOR(31 downto 0);
              error_counter          : out STD_LOGIC_VECTOR(31 downto 0);
              async_in_data          : in  STD_LOGIC_VECTOR(37 downto 0);
              async_out_data         : out STD_LOGIC_VECTOR(37 downto 0);
              async_in_clk           : in  STD_LOGIC;
              async_out_clk          : in  STD_LOGIC;
              async_in_full          : out STD_LOGIC;
              async_out_empty        : out STD_LOGIC;
              async_in_wr_en         : in  STD_LOGIC;
              async_out_rd_en        : in  STD_LOGIC;
              async_out_valid        : out STD_LOGIC);
end component;
```

List 1: The declaration for the TosNet component.

```
tosnet_inst : tosnet
Generic map(disable_slave       => ‘0’,
            disable_master      => ‘0’,
            disable_async       => ‘0’)
Port map(   sig_in              => sig_in,
            sig_out             => sig_out,
            clk_50M             => clk_50M,
            reset               => '0',
            sync_strobe         => sync_strobe,
            online              => online,
            is_master           => is_master,
            packet_error        => packet_error,
            system_halt         => system_halt,
            node_id             => "0010",
            reg_enable          => "00000001",
            watchdog_threshold  => "0000111111111111",
            max_skipped_writes  => "0000000000000000",
            max_skipped_reads   => "0000000000000000",
            data_reg_addr       => reg_addr,
            data_reg_data_in    => reg_data_in,
            data_reg_data_out   => reg_data_out,
            data_reg_clk        => reg_clk,
            data_reg_we         => reg_we,
            commit_write        => reg_commit_write,
            commit_read         => reg_commit_read,
            reset_counter       => reset_counter,
            packet_counter      => packet_counter,
            error_counter       => error_counter,
            async_in_data       => async_in_data,
            async_out_data      => async_out_data,
            async_in_clk        => async_in_clk,
            async_out_clk       => async_out_clk,
            async_in_full       => async_in_full,
            async_out_empty     => async_out_empty,
            async_in_wr_en      => async_in_wr_en,
            async_out_rd_en     => async_out_rd_en,
            async_out_valid     => async_out_valid);
```

List 2: An example of a TosNet instantiation.

References

1. **Falsig, Simon.** *FPGA-based network for distributed controllers.* Odense : The Maersk McKinney Moeller Institute, University of Southern Denmark, 2008. Available on request, sifa@mmmi.sdu.dk.
2. *An FPGA based approach to increased flexibility, modularity and integration of low level control in robotics research.* **Falsig, Simon and Soerensen, Anders Stengaard.** Taipei, Taiwan : s.n., 2010. Proceedings of 2010 IEEE/RSJ international Conference on Intelligent Robots and Systems. p. 6.
3. *TosNet: An easy-to-use, real-time communications protocol for modular, distributed robot controllers.* **Falsig, Simon and Soerensen, Anders Stengaard.** Odense : s.n., 2009. Robot Communication and Coordination, 2009. ROBOCOMM '09. Second International Conference on. p. 6.
4. **Xilinx.** XAPP463: Using Block RAM in Spartan-3 Generation FPGAs. s.l. : Xilinx, 2005.