# Sctrl Transport Protocol

Project Lab for the Minor Subject Informatics

Sebastian Millner

February 7, 2007

**Abstract**

The Sctrl Transport Protocol (STP) is a protocol to tunnel Slow Control commands via Ethernet to the Nathan system - an embedded System for controlling neural network chips. A simplified sliding window algorithm is used to warrant the execution of commands for one Nathan in the correct order. Simulation results show that it is possible to get the full benefit of a Giga Bit Ethernet connection without lowering one's sight.

## Contents

# 1 Introduction - the System

The system consists of boards called NATHAN, each with a Virtex 2 pro FPGA on it and connected to another board named BACKPLANE from whom they get their supplies and connections. The NATHANs are connected via a token ring network. The Backplane is connected to the PC via SCSI over another PC board called DARKWING. The DARKWING is connected to the token ring network, too. The connection between the PC and the system has a transferate of just about 3 MB/s. The protocol used for communication is called slow control(sctrl) protocol. It has been implemented to be reliable, while the transfer rate was no issue in this times. As a faster connection was needed, different solutions have been thought through resulting in Giga Bit Ethernet. Right now, the implementation is that far, that Ethernet packages can be stored in the SDRAM on the NATHAN, but the bridge between Ethernet package and the sctrl protocol is still missing. As a new backplane was needed, it got its own Virtex 2 pro FPGA with a point to point connection to each NATHAN. STP will be implemted within the backplane FPGA. A scetch of the system can be seen in figure 1.

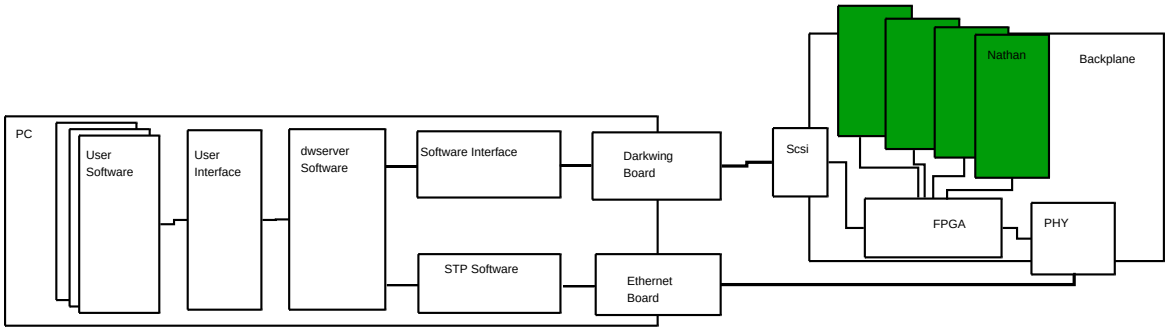


Figure 1: Overview of the System

# 2 Requirements Specification

## 2.1 Interface

There are four interfaces, this protocol has to work on: The user interface and the system interface on software level, the wishbone interfaces of the Ethernet Core and Slow Control(Sctrl) interface to the NATHANs.

### 2.1.1 User Interface

At this moment, the user is able to do single read and write commands and Blockread and -write commands. The use of block commands depends on the application. The best way is to stick to the old interface to reduce the change for the user. This is possible for the commands, but one will have to concentrate on blockcommands to get full benefit of the speed of the new connection. As each single command needs a single acknowledge to work on - a return statement in the software - each command has to be send in an own Ethernet package. It is not possible to have more than one command in progress for a single NATHAN this way. One command in an Ethernet package means we have to fill the payload up to 40 byte resulting in a bandwidth efficiency of less than 10 %. It is neccessary to switch to the use of block commands if huge amounts of data have to be transmitted.

The software of STP will have to dock on the Dwserver as can be seen in figure 1. The dwserver administrates message queues to deliver the commands from the user to the system.

### 2.1.2 System interface

The software has to be able to send raw Ethernet packages and receive the STP-packages. This can be done via the c-libraries libnet for transmission and libpcap for receiving. The benefit of libpcap in this case is, that it captures the packages as early as possible to avoid buffer overflows from copying. The problem is, that it has to be run as root.

### 2.1.3 Wishbone Interface to the Ethernet Core

In the FPGA, the Ethernet Core communicates with the SDRAM as master over the SDRAM interface. That's our docking point. The STP - slave interface has to be similar to the SDRAM interface.

Controlling of the Ethernet Core used to be done over a sctrl-Slave interface. This is not actual, but still implemented here because it was my stand of knowledge, when I started composing the system. In the new Interface, the core dirctely gets addresses where to store and load data while there is no extern management of the intern handling of these addresses needed anymore. The interface from STP to the core gets a lot simplier this way and so does the STP-unit. The change in the interface whas necessary because the wishbone-slave-interface of the former ethernetcore, which whas builded to handle Fast Ethernet is to slow for Giga Bit Ethernet. The old system used special registers called bufferdescriptors for communication between the core and systems like the STP-unit. Each descritor has one bit, signing whos allowed to write on it and fields like address and length. If you want to receive data for example, first have to look if you can write and than write the address where the data should be stored to the buffer. Now you can change the mutex bit to set the writemode to the core. When the mutex allows you to write again, the data should be stored in the memory

### 2.1.4 Slow Control Interface

The Sctrl-commands have to get from the STP-module to the modules on the NATHANs. In the beginning of this project, it was thought to directly send the commands over the given tokenring network. This would be the easiest and the most FGPG space efficient way, but also the one with the lowest bandwidth. The decision was now to make a point to point connection to each NATHAN with parallel working Sctrl-Masters for each NATHAN. In fact there are problems with the FPGA slices this solution would consume. One solution would be a slimmed Sctrl-Master.

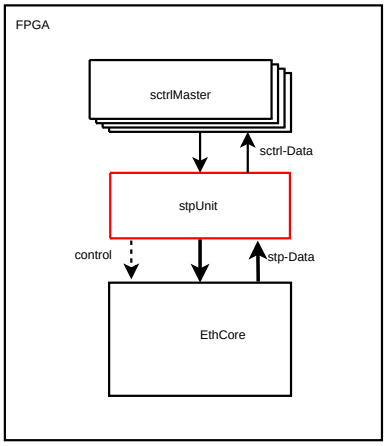


Figure 2: Interfaces in the Hardware
The control arrow from the stdpUnit to the ethernetcore symbolizes the control flux via the sctrl-slave interface.

## 2.2 Constraints

The constraints have been divided into three main parts: The hardware, functional and software constraints.

### 2.2.1 Hardware Constraints

1. The connection between each NATHAN and the Backplane enables 200 Mbits bandwidth.
2. Disordered commands will cause errors.
3. Latency is no problem.
4. The hardware implementation has to be simple not to consume too much slices.
5. There are 44 kByte blockrams on the FPGA which are shared with the other modules.
6. Commands have different length - 76 bit with data, 44 without.

### 2.2.2 Functional Constraints

1. The enduser should be able to control the system via Ethernet
2. The user should be noticed when an error occurs.
3. Errors should be corrected if possible.
4. Very old packages may be received.
5. The protocol should be independent from the type of the transported command.
6. There must not be any dead ends.
7. It must be possible to identify the packages as early as possible.
8. The content of an Ethernet package must be between 46 and 1500 Bytes long.

### 2.2.3 Software Constraints

1. The software should be system independent.
2. The software should be object oriented.
3. The code must be commented in detail.

## 2.3 Tests

Several test will be performed. As worst case scenario all NATHANs will be bombarded with single commands resulting in lots of acknowledgments and buffers on the edge. The second scenario is to block read-write on each NATHAN to get the maximum bandwidth. Finally, we will make tests with a block write and a block read write on one single NATHAN while we try to disturb the transfer with single commands to the other NATHANs.

# 3 Protocol Definition

## 3.1 Protocol Composition

The protocol is a direct payload of an Ethernet package and has two layers. The first layer has an header consisting of package type and payload. The second layer is the payload of the first and can have different types.

1. STP-Sctrl. This Package transports the Sctrl-commands. It consists of a header with fields NATHAN number, sequence number,length - the number of sctrlcommands - and a field of Sctrl-Commands. Right now, Sctrl-commands have different sizes, depending on their type. For easier handling, Sctrl-Commands have to be normed to one size in here. This is necessary for the buffer management.

2. STP-Answer. Answers from the NATHANs are send via this package. Each Answer has a field data and a field NATHAN. Several Answers are stored into one package.

3. STP-Ack. When STP-Sctrl-packages are worked out, an STP-Ack is send. An ack is send too, when an package is abandoned or a ping was send. This is why it has a field 'type' next to the fields NATHAN and 'sequence'. The status of the input buffer is stored, too.

4. STP-Ping. The function of this package is to check the connection between the pc and the System. This package does not contain anything and will be directely answered with an ack.

5. STP-Error. This Package is send, when a NATHAN is blocked and one command could not be executed. It is not implemented yet and should contain NATHAN number, sequence number and command number.

## 3.2 Protocol Function

For each NATHAN, there are two output fifos in the software interface. The first fifo (A) contains the packages to be send, and the second (B) the ones that are currently in progress. In the fifos, Packages are ordered by the STP-Sctrl sequence numbers while STP-Ping will be treated as if it had sequencenumber zero. The size of B is similar to the number of packages the hardware is able to buffer for each NATHAN. The software is allowed to send all packages in Buffer B. If an STP-Acknowledge is received, the package with the sequence number of the ack is marked acknowledged. Buffer B works like a fifo. If the first package is marked acknowledged, it can be dumped and another package can be put from A into B. With an acknowledge, the system gets the status of the input buffer and with this a map between buffer number, sequence and NATHAN. So we know, which packages are buffered and can mark them. When a package neither receives an acknowledge nor is buffered for a certain time, it will be send again. In our simulation the status buffered for packages in B is not implemented yet, so the time out has to be long enough cause else packages will be send again while they are in progress and not lost.

The principle used in this protocol is sort of sliding window.

# 4 System Model

As said above, the STP-unit lies between the Ethernet Core and the sctrl-Master-Interface. Figure 4 gives an overview which components are in the model and how they work together. For better understanding, I will describe the way of a single write and a single read before after having a look on the detailed function of the single components.

## 4.1 Active Modules

### 4.1.1 Software

In addition to the fifos descripted in 3.2 two more fifos are implemented. One for collecting the commands from the user and one for receiving the answeres from the System.

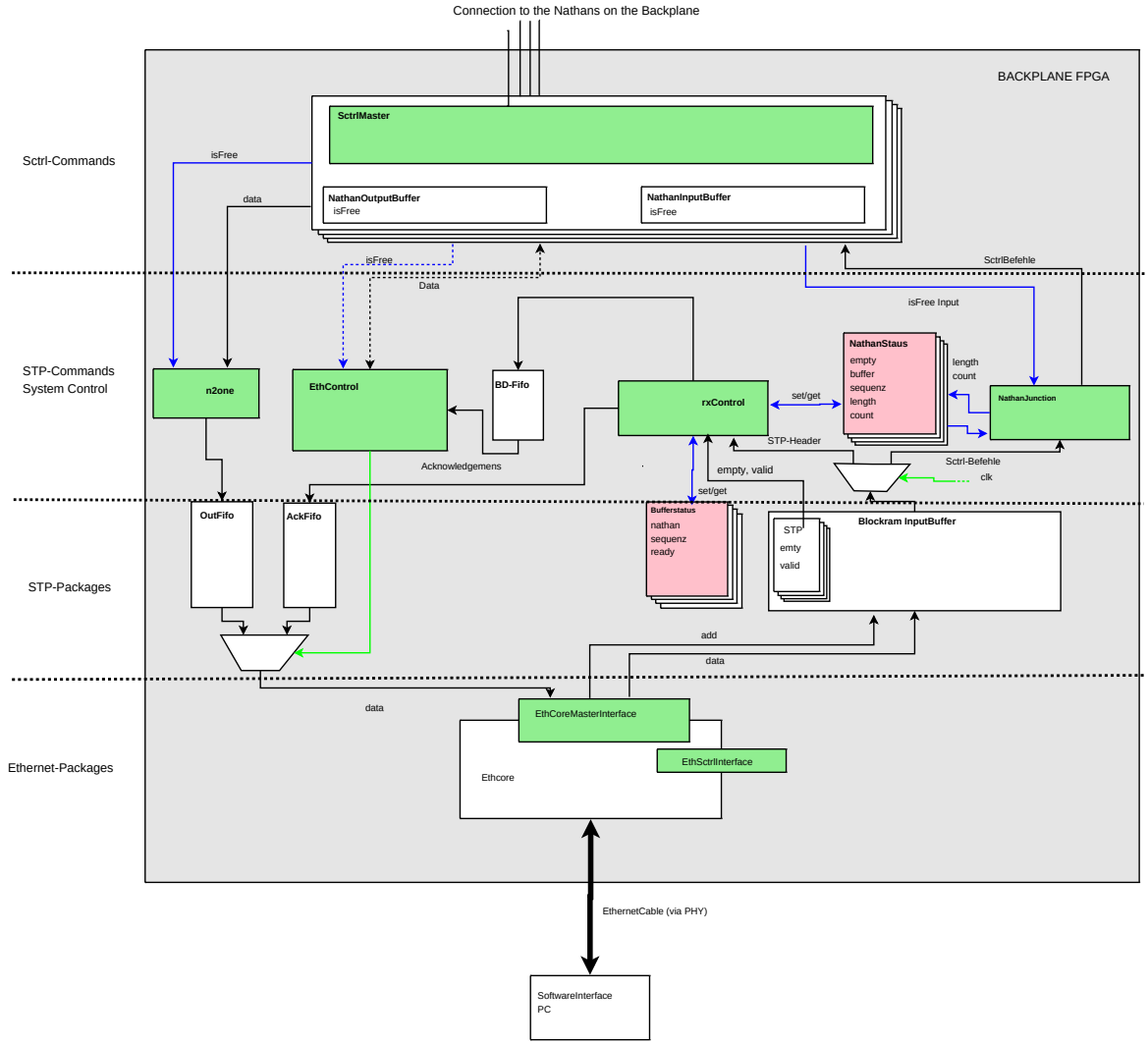


Figure 3: Block diagram of the model

### 4.1.2 RxControl

This module administrates the blockram input buffer. It traverses the Buffer and reads the header of packages if the buffer is not empty. If the package is sort of sctrl, it gets NATHAN number and sequence number and reads the NATHANStatus register for the current NATHAN. If the NATHAN is not ready and the counter number equals the number of commands, the buffer is marked ready, an acknowledge for the buffer in the NATHANStatus register is send, the buffer is marked empty and put into the buffer fifo. If the sequence number of the package in the current buffer is the next expected and the NATHAN is ready, NATHANStatus is set to the current buffer and bufferStatus is marked to NATHAN and sequence. If the sequence number is to low, an acknowledge for NATHAN and sequence of the buffered package is send and the buffer is freed again. Packages with a sequence number beyond the receiving window will be abandoned. For other packages the bufferStatus will be set to NATHAN and sequence.

#### 4.1.3 NathanJunction

The NATHAN junction traverses the NATHANStatus registers. If the status of the current NATHAN is 'not empty', the command counter number is smaller than the number of commands and the input buffer for the current NATHAN is empty, a command will be fetched from the blockram and put into the input buffer. Than we go on with the next NATHAN.

#### 4.1.4 ScrtlMaster

Each NATHAN has its own sctrl Master to whom it is connected point to point. The master takes the command from the inputBuffer. If it is a write command it sends it directly to NATHAN and acknowledges the inputbuffer when it receives an acknowledge from NATHAN. If it is an read it waits until the outputbuffer is empty and sends the command to the NATHAN than. The inputbuffer is acknowledged when output data has been received.

#### 4.1.5 n2one

This component empties the NATHANOutputBuffer and puts the data together with the NATHAN number into the OutputFifo.

#### 4.1.6 EthControl

Transmission control and ethernet core control is done by this state machine. It communicates with the ethernetCore via its sctrl slave interface. The first state is RXCHECK. In this state it looks up if there is a buffer address in the buffer fifo. If there is, it transmits a read command on the rx buffer descriptor to the Ethernet core and the next state is RXADDRESS, else it just goes over to TXCHECK. The system remains in RXADDRESS until data from the core is received. If the bufferdescriptor is ready, the next bufferaddress is written to the buffer descriptor via sctrl write and the next state is RXACTIVATE. In other cases the next state will be TXCHECK. In RXACTIVATE, a sctrl command for activation of the rxbufferdescriptor will be send to the core. TXCHECK checks, if there is something to transmit - that means that the ackfifo or the outputfifo is not empty. If so, it asks the core if the tx buffer descriptor is ready and goes over to TXLENGTH, else to RXCHECK. It stays in TXLENGTH until data from core is received. If the bufferdescriptor is not ready, the nextstate is RXCHECK. Else if the outputfifo is not to full and the acknowledgefifo is not empty, the offset for the acknowledmentfifo and the length of an acknowledgment will be send to the txbufferdescriptor. Else if the Output fifo is full enough or too long not empty, the offset for the outputfifo and the length of the package - minimum 10 usual output fifo length, maximum is the maximum number of SctrlAnsweres that fit into one Ethernet package.

### 4.2 Passive Modules

The storage devices in this unit are fifos, blockrams and registers. They connect the different state machines. NATHANInputBuffer and NATHANOutputBuffer work with acknowledgments to manage synchronisation between SctrlMasters, n2one and NATHANJunction. The BlockRamInputBuffer and the status register have control bits to tell who is allowed to write - i. e. the empty bit in case of the NastanStatus registers.

### 4.3 The way of a single write command

To simplify the situation, we assume, the system has been reseted and no other packages are on the way. The next sequence number expected for NATHAN 0 is the first - one. Our single write has to be packed exclusively in one Ethernet package because there are no other commands. We have to fill the rest of the package because it has to have at least 44 Bytes payload as this is the minimum Ethernet Payload. We can put our constructed Ethernet package in the send buffer for NATHAN 0.

Next step, the software will realize, that the send buffer is not empty, whilst the working buffer for NATHAN 0 is not full. Our package will be taken out of the send buffer and put into the working buffer,

marked ready for sending. Now the package will be send via the Ethernet adapter and marked send - it still remains in the working buffer. The working buffer for each NATHAN has the same size as the reserved buffer space for each NATHAN in the hardware. If the package is not acknowledged after a certain timeout, it will be send again.

The next component working on our package is the Ethernet core receiving the package via the Ethernet cable. Without loss of generality it can be assumed, that the buffer descriptor for receiving is ready. The core will take the address from the buffer descriptor and store the payload of the Ethernet package - namely our stp package at this address in the blockram input buffer. Assume the target address of our command is 0.

Now the RxControl component realizes, the buffer on address 0 has changed. It reads the header of the stp package and realizes, that its type is sctrl. It gets the NATHAN number and the sequence number - 0 and 1 in our case. The NATHANStatus register for NATHAN 0 tells RxControl, that NATHAN 0 is ready and that the next sequence number expected is 1. The buffer address and the number of sctrl commands of our package will be stored in NATHANStatus. The command counter is set to zero and the sequence number is incremented. Finally the empty bit is set false. Sequence and NATHAN of our package will be stored in the BufferStatus register, too. This is done to avoid a reading of the header each time.

*NathanJunction* reacts on the false empty bit in *NathanStatus* by reading our scrtl-command from the blockram and putting into the input buffer for NATHAN 0. The counter is incremented and as it is now equal to the number of commands, no other commands will be read.

The sctrlMaster of NATHAN 0 takes the command and sends it via the token ring to NATHAN 0. NATHAN 0 acknowledges the recipience of the command and the data is written to the module.

It's *rxContol*'s turn again now. As the counter equals the number of commands in *NathanStatus* for NATHAN 0, the next time, the buffer of our package is checked, *rxControl* will realize, that the buffer is worked out. An acknowledge for NATHAN 0 sequence 1 will be put in the *AcknowledgementFifo* and transmitted later on. The address of our buffer will be put into the *bufferFifo* to tell the *EthernetCore*, that this buffer is free again. Finally, *NathanStatus* and *bufferStatus* will be set empty.

The software receives the acknowledgment from the system and marks the package in the working buffer acknowledged. If the first package in the working buffer is acknowledged, it can be abandoned. Our package is abandoned and releases buffer space for other packages. The write progress is done.

### 4.4 The way of a single read command

A read command takes the same way but SctrlMaster is ready not until the data from the NATHAN has be stored in the *NathanOutputBuffer*. n2one takes the data from the NATHAN OutputBuffer and puts it into the OutputFifo together with sequence number and the counter number of the aboriginal read command.

If the *OutputFifo* is not empty for a certain time or if it is full enough and the network is not busy transmitting acknowledgments, an STPAnswer package will be transmitted and the read command is worked out.

## 5 Simulation

A simulation has been written in Java to verify the concept.

### 5.1 Introduction

It would be a great effort to implement the system directly in vhdl or verilog and debugging it then. The simulation has been done to optimize the model, to find logical errors and to find the best parameters. At first it was planed to implement the system in c++ or in systemC but for practically reasons, Java has been chosen finally - programming in Java is much faster than in c++ and I did not have any experience in programming systemC code. As Java is totally object oriented, it offers to program very close to the model. Each instance in the hardware is an object in the simulation.

## 5.2 Principles

To simulate parallelism and proceeding time, each state machine in the simulation has a method called step and each storage device has a method called update. Time is discretized into cycles. Every logic operation that happens in one cycle is supposed to happen at the same time. In this case, one cycle is the time, the EthernetCore needs to store one byte in the buffer, so it is equivalent to the 125 MHz cycle of the network. Each simulation cycle all step functions are called, followed by the update functions. The state of the storage devices does not change until the update functions are called. As the statemachines only communicate over the storage devices, the system state for each maschine is the same during one cycle.

The simulation of most machines is behavioral and no direct states can be seen in the code. The only exception is the ethControl machine which has clearly defined states.

Update functions are implemented with a buffer. They have been implemented into the working simulation system and do not have any effect to the interfaces of the storage devices. The statemashines have no idea, that the data is not stored directely.

As test bench a stub of the software interface has been implemented which is connected to the system via an instance called cable. It is implemented with two fifos - one for each direction. To model a realistic environment, the cable can lose packages or disorder them. The probability for this can be set with to parameters. The software interface simply has all the fifos described above and does the flow control. It is still simplified because it only accepts packed stp-packages and returns stpAnsweres. Features like the ping package are not used at all. The information about the current bufferstatus, given with each acknowledge and ping is not used at all.

All modules and the test bench are initialized and connected in the top class.

Parameters are initialize from a class called SystemConst.

## 5.3 Appliance

To run a test just set the wished parameters in the SystemConst class and run the top class, where a main function is implemented. In the main function you first have to initialize the system and then run the specified test. The way the tests are implemented it is not possible to run several tests aligned because the sequence numbers start from 1 for each test and the current sequence numbers are not different after a test has been run. An exception are firsttest and swtest, which use the local variable sequence which can be incremented after each test.

On each run, the important system constants, the transmitted packages and the elapsed cycles are written in log.txt which can be used for analyzes over several test. Each cycle, the free space of the outputbuffer and the acknowledgment fifo is written in data.txt. This file is overwritten each time a new test is started.

If you change something in the system and want to verify if it still works you run firsttest at first by calling it it the mainfunction in the top class. If it works, you go over to two firsttests in a row. Now you can try swtest and then the first mode of test(int) with one package and few NATHANs. Increase the number of packages and the number of NATHANs. Go over to the second test. If all tests are successful - what successful means will be declared in the next paragraph - your verification is done.

## 5.4 Tests

As denoted above, there are several different kinds of tests.

### 5.4.1 firstTest

This test simply tries to write the number 5 on NATHAN 0 module 0 address 0 and reads this place again. It displays the received packages. There should be one acknowledge as one package has been transmitted and the answer from the read command. As the working time is independent from the status of the accomplishment in this test you can rise the number of loops.

#### 5.4.2 swTest

The software stub will be tested with this test. It does the same as first test, but uses the software interface to transmit the package to the system and gets the answer in return from the software. You wont get an acknowledgement because the software is taking over flow control now and handles the acknowledgements as descripted above.

#### 5.4.3 orderTest

The first mode of test(int) is order test. In this test, single reads and writes for each NATHAN are send to the system - each in its own package, so each will get its own acknowledgement. There will be ordered numbers written to and read from the first address of the first module of each NATHAN. The answers for each NATHAN should be ordered, too. If so, the test is successful. The test is over, when all transmitted packages are acknowledged. This does not mean, that all packages have already been transmitted from the stp-unit. To be sure to get all packages, you would have to run the system for some more cycles. This test floods the connection with acknowledgments and is the extreme case described above. The acknowledgments for the small packages obstruct the answers from being send which thwarts the whole operation. Filling bits in acknowledgments and single commands thwart it, too.

#### 5.4.4 utilizationTest

In difference to order test, here only one package is transmitted to each NATHAN, transporting the same commands from orderTest. This test is made to get the maximum performance. It is mode 2 of test(int).

#### 5.4.5 Realistic Maximum Package Size Test

Here the same commands are packed in to packages with a maximum size, that is realistic in comparison with the actual ethernet maximum package size. At least one package is transmitted to each NATHAN. This is mode 3.

#### 5.4.6 One Nathan

In this test one NATHAN gets orderTest-commands packed in packages with the maximum size whilst the other NATHANs get single commands. This test is done as realistic working scenario. The packages for the other NATHANs are called noise packages in the simulation. This is mode 4.

#### 5.4.7 One Nathan Write

One NATHAN gets write commands and the others get noise commands. This test can be executed with mode 5.

# 6 Conclusion

## 6.1 Simulation Results

### 6.1.1 Order Test

Sending 100 write commands to each NATHAN resulting in 200 commands at all for each, we can get a transfer rate R of R = 85 Mbits per second. This transfer rate is calculated like this:

$$R = \frac{NumberOfCommands * BitSizeOfCommand * NumberOf\text{NATHAN}s * ClockRate}{ElapsedCycles} \quad (1)$$

Figure 4 shows the free space in the transmission fifos for the order test. As can be seen, the system is busy sending acknowledgments all the time, so the output fifo is not emptied before it is nearly full. This is why you get this sawtooth look. The acknowledgment fifo is not filled more than just about 64

items. This is because no more than 64 unacknowledged packages are transmitted by the software in this simulation. The fifo needs to be bigger because it could happen, that a packages is transmitted twice when it is timed out in the software.

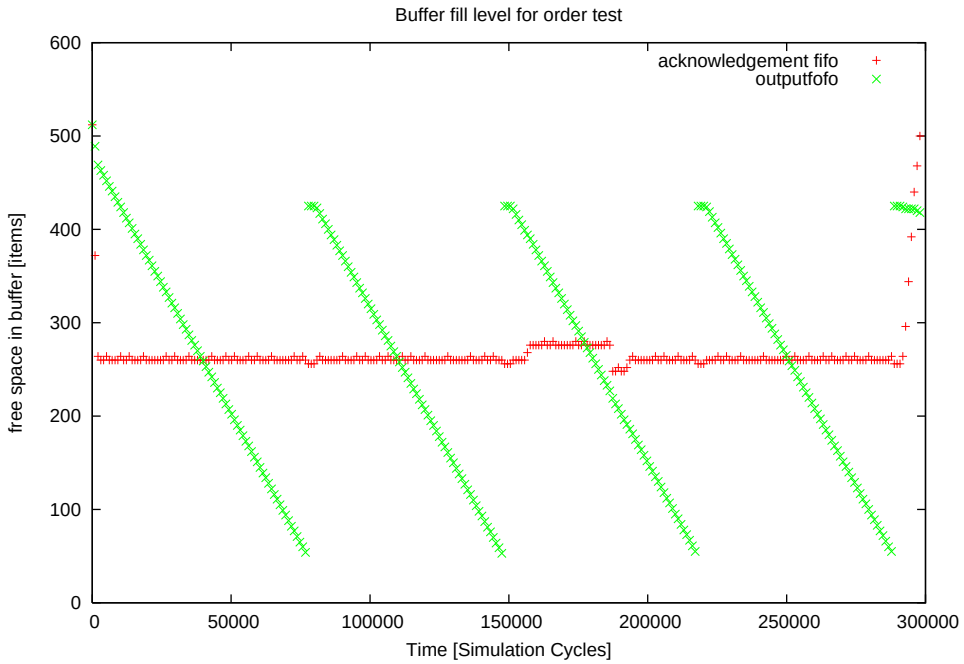


Figure 4: Buffer fill level for order test

### 6.1.2 Ulitization Test

We are still transferring 100 commands to each NATHAN. In this experiment, we get a transfer rate of R = 900 Mbits per second, which is nearly the one Gigabit per second. The space left in the fifos is illustrated in figure 5. At the beginning, it takes some time to store the packages in the input buffer. This is why the speed the buffer is filled increased from cycle 0 to just about 3500. First there is no package. Than the first packages arrived and one NATHAN is writing into the output buffer. With proceeding time, more packages arrive, so more NATHANs fill the buffer. After a certain number of cycles, the first package is worked out and an acknowledgment is send. Whilst this is executed, the output fifo is filled. We get a triangle structure as can be seen in the plot in figure 5.

### 6.1.3 Realistic maximum Package Size Test

Now 1000 commands are transfered to each NATHAN. We get a rate of 900 Mbits per second. Figure 6 shows the result for the buffer fill levels.

### 6.1.4 One Nathan Write

When writing 10000 write commands to one NATHAN and 50 noise commands to the others, we get a transfer rate of R = 500 Mbits per second. The noise is chosen in a way that the single transmission of the noise would not take longer than the single transmission of the write commands. Figure 7 illustrates the results.

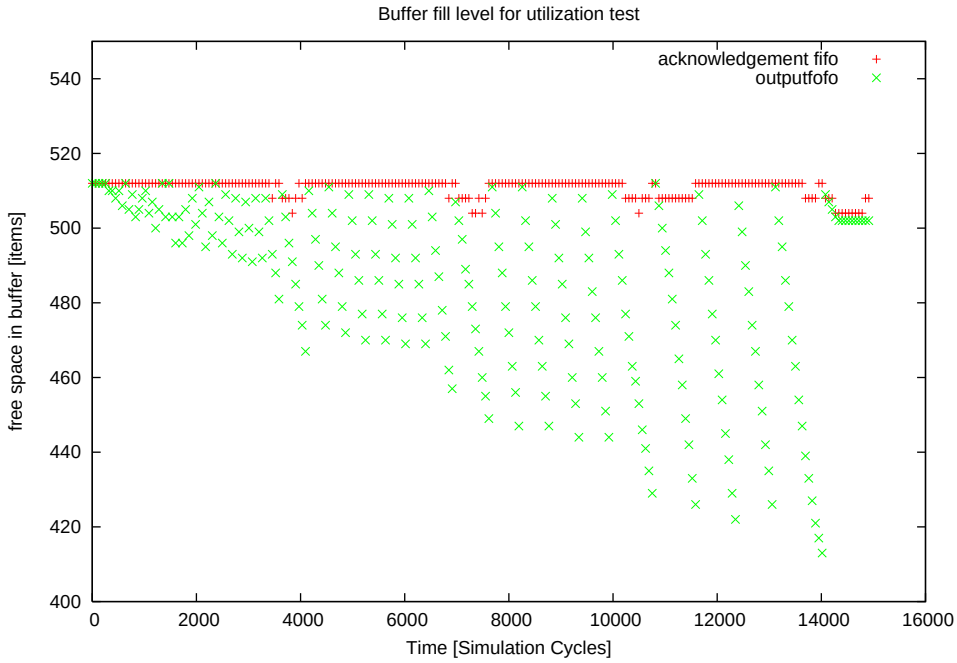


Figure 5: Buffer fill level for utilization test

## 6.2 Quality Management

Regarding our functional constraints from the beginning of the project and our test results, we can evaluate the system.

### 6.2.1 Remote Controllability

The System is not fully controllable via ethernet right now. The user is not able to reset the system neither one single NATHAN board.

### 6.2.2 Error Messages

Error messages are only transmitted, when a package is abandoned. The software does the rest. If a package is lost, it does not get an acknowledge and will be send again. One could realize, that one NATHAN module is suspended, when there is no ack for one package after a certain time.

### 6.2.3 Error Correction

All correctable errors are corrected. Disordered Packages will be ordered again by sequence numbers because only the next expected sequence number is accepted by rxControl.Lost packages will be send again. If one NATHAN is hung up, the procedure would be a reset for this single NATHAN. This can be done by the software, when reset is implemented.

### 6.2.4 Old Packages

Packages that are that old, that there is an sequence number overflow between them and the current package, could cause undetectable errors if they are in the reception window of the current number. This is very unlikely and can be nearly barred by choosing more bits for the sequence number.

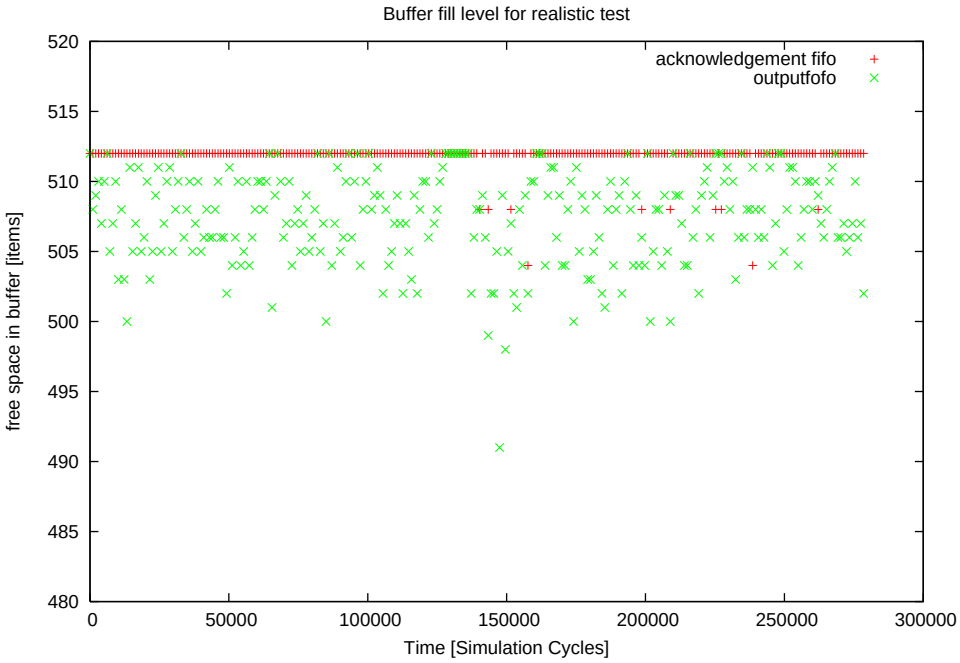


Figure 6: Buffer fill level for realistic test

### 6.2.5 Payload Independence

The payload is only important for the sctrl-Master-unit. The package processing does not rely on the payload.

### 6.2.6 Dead Locks

There are no Dead Locks, if the parameters are set right. You must be sure to leave the timeout for resend big enough because else the system will be occupied with acknowledgments all the time.

### 6.2.7 Package Identity

Packages can be marked directly in the type field of the ethernet header.

### 6.2.8 Package Size

Package size is not a problem at all. Minimum and maximum package size of Ethernet packages have been included in the simulation of the System.

### 6.2.9 Transfer Rate

The transfer rate lies between 85 and 900 Mbits and is 500 Mbis in a normal working situation. We can get the full benefit of our Gigabit Ethernet connection.

## 6.3 Viability

As we only have 44 kByte Buffer on the FPGA, our input buffer is bounded. If we use two buffers of 1.5 kilobyte packages for each NATHAN, we already have 40 kByte engaged. This is not possible, so we have to use smaller packages. If we restrict the package size to 6 slowcontrol packages, we could use 512 bytes as buffer size, and one blockram  2 kB for 4 buffer. Then 16 of the 22 blockrams should do the job.

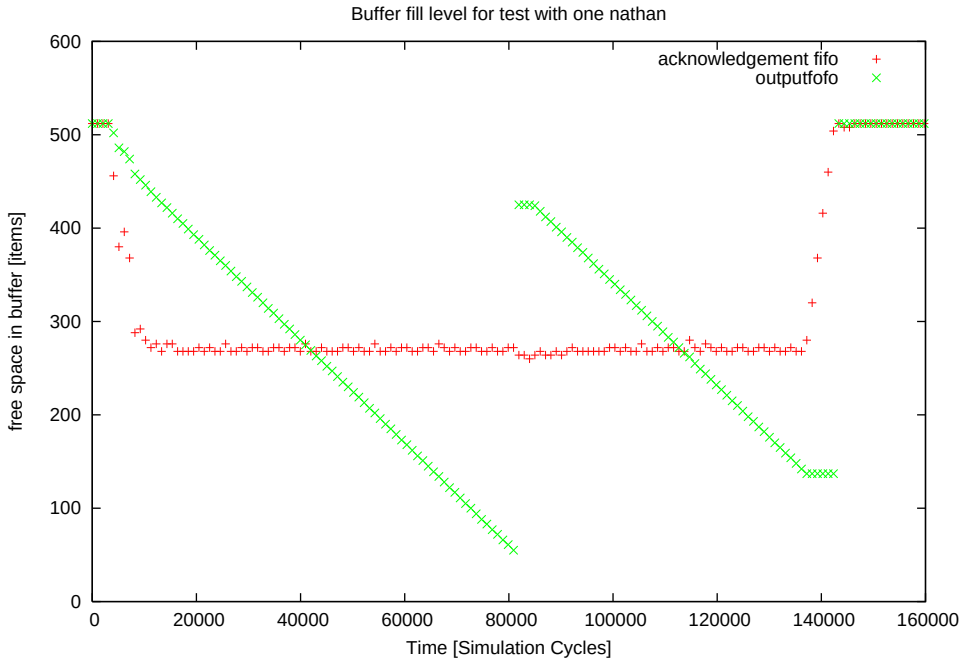


Figure 7: Buffer fill level for one NATHAN write

### 6.4 Future

The first thing to be done in the future would be updating the interface to the ethernetcore as it is not actual anymore. This will make the system easier because we do not need a complex ethcontrol module anymore.

Then the reset functionality should be implemented in the software model.

Next Step would be the hdl implementation of the system.

## 7 synopsis

I tried to use technics I learned in software engineering in this project. This led me to a very abstract view at the beginning. Modeling and user feedback was needed, to get the reference to the system again. Implementing the simulation I found lots of errors in the model I made and corrected them. If this process would have been done in hdl it would have taken me ten times longer.

The results of the simulation are quite satisfactory and I look forward to an implementation in hdl for final verification.