# UDP/IPv4 FPGA command protocol

Introduction 2

Theory of operation 2

Schematic overview 2

On timeouts and lost packets 2

UDP packet format 3

Maximum size 3

Alignment and endianness 3

Layout 3

Document revision history 4

Supported commands and their replies 4

wait-for-pps instruction prefix (1.0) 5

accessing the EPCS flash-memory (1.0) 6

read/write of memory or memory-mapped registers (1.0) 7

write of bit field in memory or memory-mapped registers (1.2) 8

read/write a hardware FIFO (1.1) 9

read-modify-write bitwise operations (1.0) 10

# UDP/IPv4 FPGA command protocol

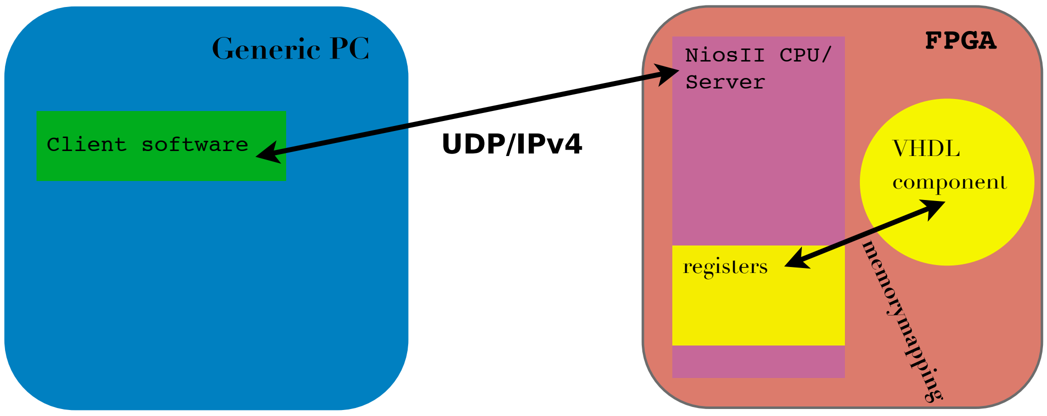
## Introduction

This document will serve as reference for those developers who wish to implement the UniBoard UDP-based command protocol in the programming language of their own choice which is not Erlang.

In order to control the FPGAs which are mounted on the UniBoard a simple, UDP based, protocol and message format was designed. The choice for UDP is an obvious one, given the restrictions on code-size and processing power: the goal is to have command-processing software running on a 125MHz (preferably slower) CPU with a total of 128kB of memory (preferably 16kB), to leave as many FPGA resources as possible for useful components. The fact that UDP is a connection- (and hence state-)less protocol makes access control simpler (there is none) as well as controlling/monitoring a UniBoard from multiple programs simultaneously much simpler. Restarting a crashed controller program (or a crashed FPGA) does not affect the other end of the communication.

## Theory of operation

### Schematic overview



The principles behind the operational model have been kept as simple as possible. A VHDL engineer designs a component and defines some control/status registers. These registers must be memory mapped into the NiosII CPU’s memory space. Software running on the NiosII will act as a server, by reacting to incoming packets on its 1 Gbit ethernet interface and processing the commands it finds in there. The combined replies will be sent back to the client.

### On timeouts and lost packets

UDP is a lossy protocol. No guarantees are made with respect to delivery of a packet nor will there be any notification should a packet be lost. The client software must be prepared to deal with this: theoretically either the command packet or its reply can get lost. The NiosII firmware executes each command packet it receives.

# UDP/IPv4 FPGA command protocol

## UDP packet format

### Maximum size

The NiosII server currently supports only the minimum ethernet frame length of 1500 octets (bytes, usually). Larger packets will be (truncated/rejected) by the network card on the FPGA.

### Alignment and endianness

The UniBoard command packet carries binary data only. Because of the NiosII's 32 bit little-endian architecture and (implicit) 32 bit aligned memory accesses[1] it was decided that all entries in the packet should be multiples of 32 bits in little-endian format and that all addresses found in the commands should be 4 byte/32 bit aligned and also in little-endian format. The firmware will send back and error reply for a such a non-aligned access.

### Layout

The UniBoard command packet payload starts immediately following the UDP/IPv4 header. Command and reply packets are identical in this respect: the first 32 bits are the Packet Sequence Number (PSN). When a client sends a command packet it will receive the replies, eventually, in a packet which starts with the same PSN.

| 32 bits | Variable length, must be >0 bytes |
|---|---|
| PACKET SEQUENCE NUMBER | COMMANDS/REPLIES (depending on command or reply packet) |

The PSN is a 32 bit number which, combined with the sender's IP address and originating UDP port number will determine the unique triplet based on which the NiosII server decides wether to execute a packet or send a cached reply - should a reply with this triplet key (still) exist in its cache[2].

The NiosII server does not place any constraints on the PSN - it's sole use is forming the key triplet. It is advisable though, that subsequent controlling processes do not generate the same PSN sequence when sending commands. It may lead to unexpected behaviour, typically depending on the actual PSN sequence generated.

[1] Non-aligned accesses are allowed on the NiosII memory bus; behind the scenes they are translated into an aligned access plus some shift/mask operations by the CPU

[2] This is true only for the `unb_os` firmware. Other, smaller, versions (`unb_osx`, `unb_osy`) do not exhibit this behaviour

# UDP/IPv4 FPGA command protocol

## Document revision history

- version 1.2 October 2012
- version 1.1 November 2011
- version 1.0 September 2011

## Supported commands and their replies

This section will list the commands the NiosII server recognizes and the replies they generate.

In general each command starts with an OPCODE field, indicating the requested function. An actual OPCODE is never 0x0. The OPCODE 0x0 is reserved to mean "end of command sequence". An OPCODE is typically followed by a "NUMBER OF OPERANDS" field, followed by the actual operands:

| 32 bits | 32 bits | Variable length |
|---|---|---|
| OPCODE | NUMBER OF OPERANDS | DATA FIELD |

Commands including an address usually adhere to the convention that if the operation was successful the reply starts with the ADDRESS (optionally followed by the results), or with the binary-not of the ADDRESS in case of (detectable) failure. The binary-not means that each bit in the original ADDRESS is flipped.

# UDP/IPv4 FPGA command protocol

## wait-for-pps instruction prefix (1.0)

The NiosII server suspends execution of commands in the current packet until a PPS tick is seen. The wait should timeout in a period between 1.0s and 1.5s – the upper bound set by the watchdog interval: the FPGA will be automatically reset if the FPGA does not acknowledge the PPS tick for a period longer than 1.6s. Return value is a 32bit word consisting of the ASCII character codes of “1PPS” if the PPS tick was detected or “0PPS” if not. The command does not necessarily have to be followed by other commands.

### Command:

| 32 bits |
|---|
| 0xFFFFFFFF |

### Reply:

| 32 bits |
|---|
| 0x48808083 or 0x49808083 |

## accessing the EPCS flash-memory (1.0)

Each FPGA on the UniBoard has EPCS flash-memory attached to it. After a power up it will boot from this persistent storage. The NiosII server software supports reading/writing of this memory area, if the flash-controller is built into the FPGA's design. Access to the flash-memory is via a rather low baud rate serial device so do not expect very fast transfers.

A nasty property of the flash-memory is that you cannot write '1's to it. In order to (properly) write arbitrary binary data the flash-memory it must first be erased (sets the content to all '1's). Then write your data and all the '0's will be written and the '1's, well, they were already there. Unfortunately, reads and writes are done in pages (256 bytes) whereas the erase is done is by section. The amount of pages per section is heavily dependent on the actual type of EPCS flash-memory attached. In this version of the UniBoard there are 64 sections of 1024 pages each (EPCS 128).

The addressable range of the flash-memory area is a flat memory space which starts at 0x0 and extends to the size of the flash-memory as measured in units of bytes.

**flash write:** write one page of data to the FPGA's EPCS flash region at the indicated address.

Command:

| 32 bits | 32 bits | 256 bytes |
|---|---|---|
| 0x6 | ADDRESS | ONE PAGE OF CONFIGURATION DATA |

Reply:

| 32 bits |
|---|
| (NOT) ADDRESS |

**flash read:** read one page, 256 bytes, of data from the EPCS flash region at the indicated address. If the read fails, NOT ADDRESS starts the reply and *no* data follows.

Command:

| 32 bits | 32 bits |
|---|---|
| 0x7 | ADDRESS |

Reply:

| 32 bits | 256 bytes \| 0 bytes |
|---|---|
| (NOT) ADDRESS | ONE PAGE OF CONFIGURATION DATA |

**flash erase:** erase the section (usually multiple pages!) of the flash region where ADDRESS points into. It is suggested to use the start address of the section to erase. For this to work efficient the section size must be known (it is device dependent) since then the start address of section *n* can be easily computed.

Command:

| 32 bits | 32 bits |
|---|---|
| 0x8 | ADDRESS |

Reply:

| 32 bits |
|---|
| ADDRESS or (NOT ADDRESS) |

## read/write of memory or memory-mapped registers (1.0)

These commands will most likely make up the bulk of the monitoring/control. These commands allow the client software to read or write locations in the full 32-bit NiosII address space.

Careful notice should be taken that the NiosII firmware does not do ANY bounds checking/access control whatsoever. Reading from or writing to addresses that are not wired may or may not hang, crash or burn. Overwriting the firmware itself is not unthinkable so try to make sure you don’t do that.

**read:** reads N consecutive 32-bit locations starting at START ADDRESS. Returns START ADDRESS followed by N*32-bits if successful, or NOT START ADDRESS in case of failure to read.

### Command:

| 32 bits | 32 bits | 32 bits |
|---|---|---|
| 0x01 | N | START ADDRESS |

### Reply:

| 32 bits | N * 32 bits \| 0 bits |
|---|---|
| (NOT) START ADDRESS | DATA |

**write:** writes the data from the command packet into N consecutive 32-bit locations starting at START ADDRESS. Returns START ADDRESS if successful, or NOT START ADDRESS in case of failure to write. It is unclear if writes will ever, detectably, fail. Assume that at some point writes may fail and return the error reply.

### Command:

| 32 bits | 32 bits | 32 bits | N * 32 bits |
|---|---|---|---|
| 0x02 | N | START ADDRESS | DATA |

### Reply:

| 32 bits |
|---|
| (NOT) START ADDRESS |

## write of bit field in memory or memory-mapped registers (1.2)

In order to set a bit field consisting of more than one bit inside a 32-bit word without affecting the values of the bits not part of the field one must use at least two bitwise instructions to make sure the desired bit pattern/value does correctly appear in the bits of the bit field.

In the command set up to and including version 1.1 there was no support of doing this in a single read-modify-write instruction: the client software had to translate a write to a bit field into two separate bitwise commands, causing the target address to be written to twice. For hardware components reacting to the write strobe of the address this is disastrous.

To that effect this instruction has been introduced in the firmware. It takes a base address, the bit field's mask value and an array of values as arguments. For each value it will read to register's value into a temporary variable, apply a bitwise AND with the bitwise NOT of the mask to reset the field to all zeroes. Then the temporary variable will be bitwise OR'ed with the value which has been bitwise AND'ed with the mask such that bits outside the masked field are zeroed. Finally the new value will be written back to the target address.

Careful notice should be taken that the NiosII firmware does not do ANY bounds checking/access control whatsoever. Reading from or writing to addresses that are not wired may or may not hang, crash or burn. Overwriting the firmware itself is not unthinkable so try to make sure you don't do that.

**write:** writes the data from the command packet into N consecutive 32-bit locations starting at START ADDRESS. Returns START ADDRESS if successful, or NOT START ADDRESS in case of failure to write. It is unclear if writes will ever, detectably, fail. Assume that at some point writes may fail and return the error reply.

Command:

| 32 bits | 32 bits | 32 bits | 32 bits | N * 32 bits |
|---|---|---|---|---|
| 0x0B | N | START ADDRESS | MASK | DATA |

Reply:

| 32 bits |
|---|
| (NOT) START ADDRESS |

## read/write a hardware FIFO (1.1)

It is not uncommon for hardware to communicate via a FIFO. Reading from a FIFO means repeatedly reading from the same (hardware)address and writing, consequently, means repeatedly writing to the same (hardware)address. The standard READ/WRITE commands read from/write to consecutive, incrementing addresses. Implementing a FIFO read of N words using these commands would require the programmer to send N identical one-word READ commands, keeping the address to read from constant.

For efficiency reasons it was decided to add dedicated FIFO READ/WRITE commands that would repeatedly read from/write to the same address (i.e. do not increment the source or destination address).

**fifo read:** read a number of 32-bit words from the FIFO located at address FIFO ADDRESS. The firmware will reply with either FIFO ADDRESS + N*32 bits of data following or the binary NOT of FIFO ADDRESS if the read (detectably) failed.

Command:

| 32 bits | 32 bits | 32 bits |
|---|---|---|
| 0x09 | N | FIFO ADDRESS |

Reply:

| 32 bits | N * 32 bits \| 0 bits |
|---|---|
| (NOT) FIFO ADDRESS | DATA |

**fifo write:** write the indicated number of 32-bit words to the FIFO located at address FIFO ADDRESS. The firmware will reply with either FIFO ADDRESS or the binary NOT of FIFO ADDRESS if the write failed.

Command:

| 32 bits | 32 bits | 32 bits | N * 32 bits |
|---|---|---|---|
| 0x0A | N | FIFO ADDRESS | DATA |

Reply:

| 32 bits |
|---|
| (NOT) FIFO ADDRESS |

## read-modify-write bitwise operations (1.0)

Sometimes it is handy to apply a read-modify-write sequence; i.e. applying a mask to register. The commands in this section allow you to do this efficiently.

All commands read the existing contents of the locations starting at START ADDRESS, compute new values from the bitwise operation indicated by OPCODE with the old values and the mask for that location (from the MASK 'array'). Finally they write the results back to the locations. The return value is START ADDRESS if all RMW sequences were performed successfully, NOT START ADDRESS in case of failure.

### Command:

| 32 bits | 32 bits | 32 bits | N * 32 bits |
|---|---|---|---|
| OPCODE | N | START ADDRESS | N * 32bit words of MASK |

| OPCODE | 0x03 | 0x04 | 0x05 |
|---|---|---|---|
| BITWISE OPERATION | AND | OR | XOR |

### Reply:

| 32 bits |
|---|
| (NOT) START ADDRESS |