UDP/IP WITH VHDL

Written by: Jussi Nieminen
22.10.2009
Tampere University of Technology

Contents

1 INTRODUCTION

UDP/IP block is a network interface that is capable of sending and receiving UDP (User Datagram Protocol) packets over ethernet. The block also includes ARP (Address Resolution Protocol) funtionality. All the source codes are written in VHDL (Very high speed integrated circuit Hardware Description Language) and licensed under LGPL (Lesser General Public License).

1.1 Intended use and restrictions

This block is originally designed for a Network-on-Chip monitor application to get high data bandwidth from Altera's DE2 board to a single PC. (At least higher than with UART.) Future applications in mind possibility to receive UDP packets sent from a PC was also included. This means that the block is desinged for predetermined communication between two agents using UDP packets only. IP level broadcasting or network masking are not supported. Correct functionality when connecting this block to a network full of agents and protocols is not guaranteed. In addition the size of the used ARP table (two entries) is not suitable for use in larger networks and there's no timeouts in cases like ARP request being never answered.

So to wrap things up, without modification this block can be used in special communication within two agents or in small special networks using proper UDP packets only. Other types of packets are always simply discarded. If you need something more advanced, this block is a good starting point for further development.

1.2 ARP block

As mentioned, the UDP/IP block also includes ARP functionality. The internet is a wonderful place, and with little searching ready-made ARP blocks by Ashley Partis and Jorgen Peddersen [1] came up. The files were modified to meet our needs (and match our coding style), but the basic functionality is the same.

2 USAGE

2.1 Application side

This section describes how the UDP/IP block is connected to and used with an application.

2.1.1 Interface

Figure 1 shows the interface with an application. Thicker arrows are wider busses while thinner ones stand for single bit signals. Most of the signals are hopefully self-explanatory. The rx_erroneous signal means that the ethernet controller has detected a possibly data corrupting situation (e.g. collision), but the frame was nevertheless received. So it's relayed to the application, which can decide what to do with it.

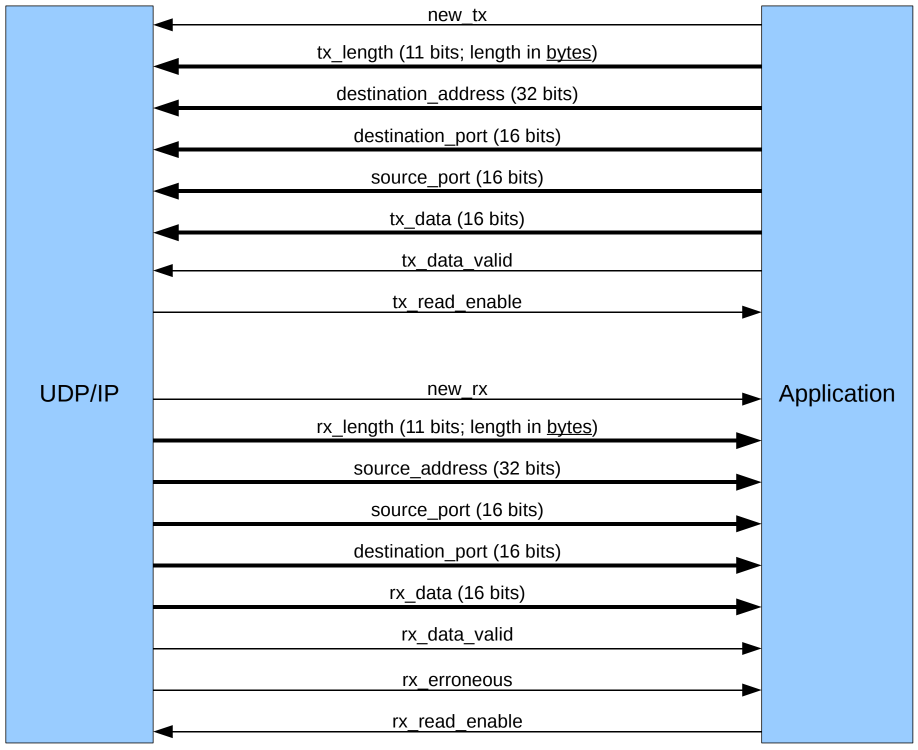

Figure 1: *Interface between an application and the UDP/IP block.*

2.1.2 About the length and data busses

As shown in the interface, the length bus is 11 bits wide. This corresponds with the maximum possible ethernet frame size (1518 bytes). IP packets could be much longer, but there is no support for packet fragmenting and thus the maximum possible data length is maximum ethernet frame size minus headers and ethernet checksum. The application is responsible for keeping the packet size below maximum.

$$\begin{aligned} l_{data} &\leq l_{max} - l_{eth} - l_{ip} - l_{udp} - l_{CRC} \\ &= 1518 - 14 - 20 - 8 - 4 \\ &= 1472 \text{ bytes} \end{aligned}$$

Length information is given before any data so the application must know beforehand how much it's going to send. This is due to the fact that we want to use ethernet controller chip's internal data buffers to save FPGA resources, and for that (at least some if not all) ethernet chips need the length beforehand. The application must make sure, that it really sends or receives data as much as the length value tells. **Remember that the length is given in bytes even though data is written two bytes at the time!**

The 16-bit data bus consists of two bytes instead of a single 16-bit data word. Endianess of the data bus is different from an ethernet frame (*n downto 0* versus *0 to n*), so if you want to have a sequence 0xABCD on a data frame, you will have to write it as 0xCDAB to the data bus. If there is a transfer with odd length, the last byte will use the bits 0 to 7 while bits 8 to 15 can have random values. The figure 2 desperately tries to explain all this.

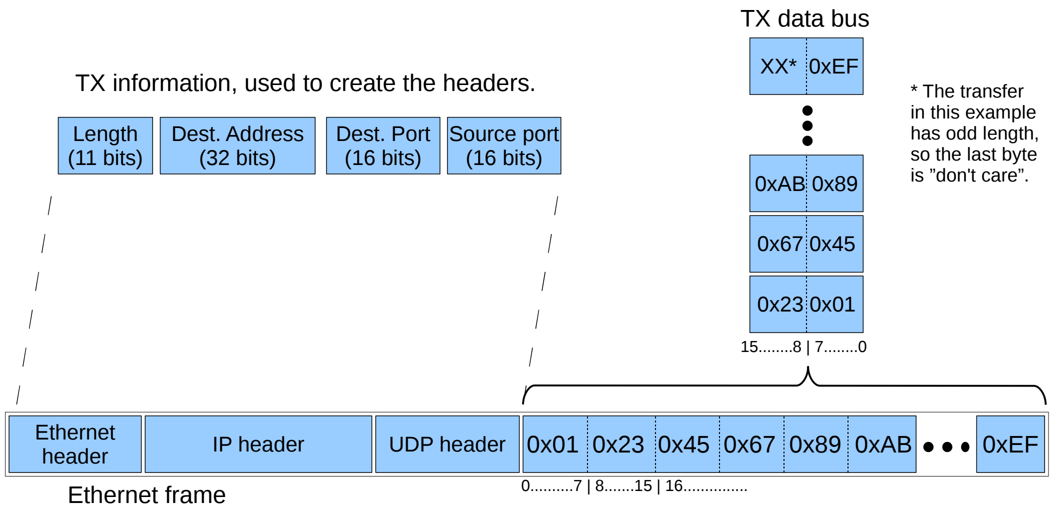

Figure 2: *Example of how the ethernet frame is formed.*

2.1.3 How to get bits running

Timing for transferring and receiving data is shown in figure 3. A new transfer is started by setting up the new_tx signal, all the needed information (TX length, destination address and ports) and the first two bytes of data with data_valid signal. When UDP/IP block is ready, it will start reading data by raising the tx_read_enable signal. All the application's signals must remain stable from the moment new_tx is risen to the first time the read enable is up. It might take a while before the UDP/IP block starts to read if the destination MAC address is not yet in the ARP table. The first time the read enable is up means that all the other information (address, ports etc.) has been received and those signals no longer have to remain the same.

Receiving works exactly the same way, the UDP/IP block sets all the needed information, first bytes of data and new_rx signal and application starts to read by raising up the rx_read_enable signal. The information signals are allowed to change (thus are considered invalid) once the read enable signal is risen for the first time.

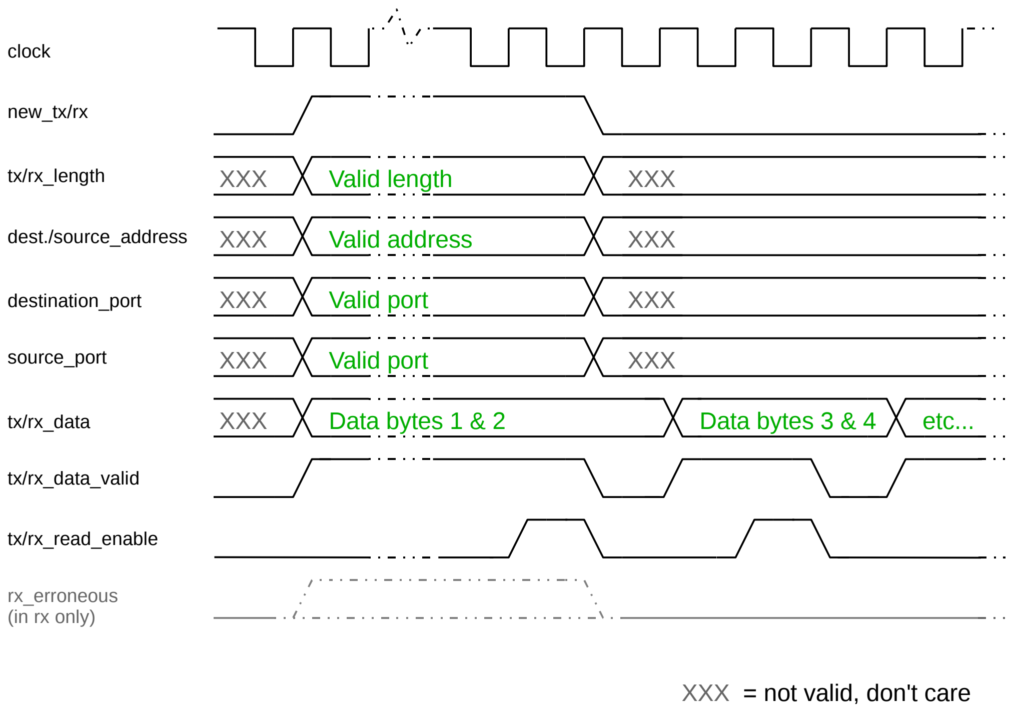

Figure 3: *Timing of transferring or receiving data.*

2.2 Ethernet side

UDP/IP block's communication with an ethernet controller block is similar as with an application. Information signals are a bit different but timing follows the same principle. In fact, when headers have been written and it's time to write the data the UDP/IP block simply connects its input and output data signals so that the application is really talking straight to the ethernet controller block. More information about this can be found from section 3.

2.2.1 Interface and timing

Figures 4 and 5 contain the interface and timing information. The ethernet frame type value is part of an ethernet header and it identifies the protocol that is carried in that frame. UDP/IP block only accepts frames with frame types 0x0800 or 0x0806 standing for IP or ARP frames respectively. Frames with other types are read from the ethernet controller block but discarded without notifying the application.

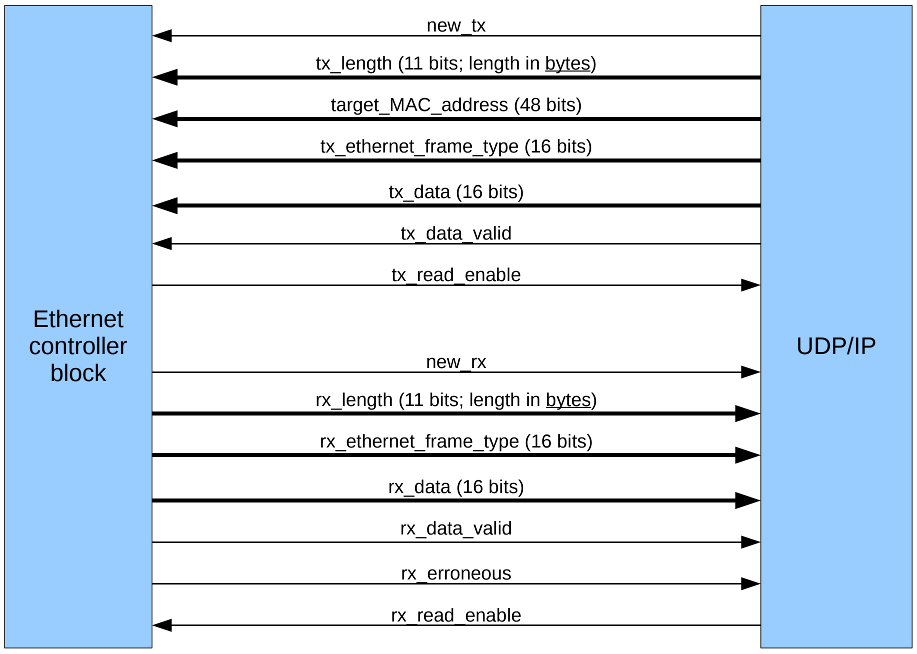
Ethernet controller block
UDP/IP
new_tx
tx_length (11 bits; length in bytes)
target_MAC_address (48 bits)
tx_ethernet_frame_type (16 bits)
tx_data (16 bits)
tx_data_valid
tx_read_enable
new_rx
rx_length (11 bits; length in bytes)
rx_ethernet_frame_type (16 bits)
rx_data (16 bits)
rx_data_valid
rx_erroneous
rx_read_enable

Figure 4: *Interface between an ethernet controller block and the UDP/IP block.*

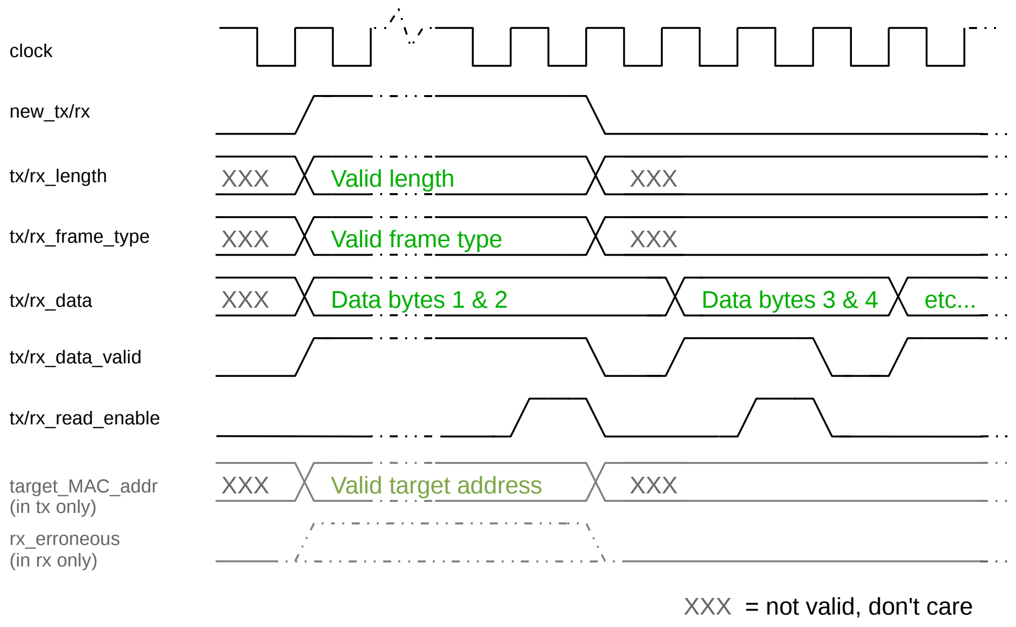

Figure 5: *Timing for communication between UDP/IP and ethernet controller block.*

3 IMPLEMENTATION DETAILS

This section briefly describes how the UDP/IP block is implemented. If you aren't making any changes to the source code you hopefully don't have to care about the following stuff.

3.1 Structure of the UDP/IP block

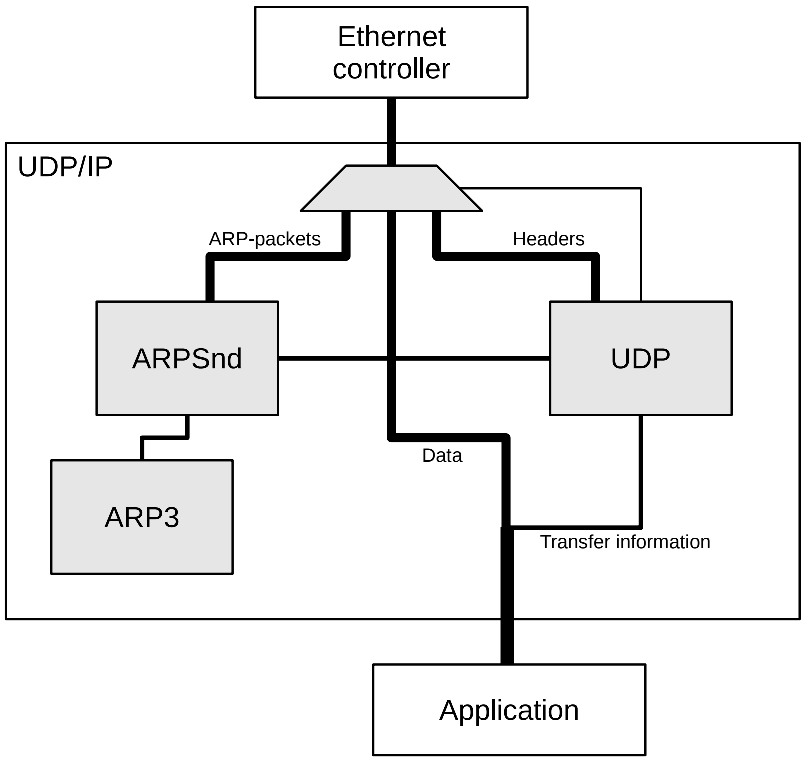

Figure 6: *Block diagram of the system.*

Figure 6 shows the construction blocks of the system. Block named UDP is responsible of running the show: it commands the ARP side and selects the multiplexer input. Its other main task is to form or interpret the UDP/IP headers depending on the direction of the transfer.

The ARP functionality [1] is divided to two blocks. The ARP3 (the ”3” comes from the file name, don't know what it stands for) block includes the ARP table and is responsible of updating it. The ARPSnd block is responsible of sending and receiving ARP queries and it also communicates with the UDP block.

The (de)multiplexer selects one of the three sources/destinations for data going/coming to/from the ethernet controller. UDP sends and receives headers and ARPSnd ARP packets. The third data bus goes directly to the application. So e.g. during an outgoing transfer, after the UDP block has finished sending headers it switches the mux to relay data from the application straight into the ethernet controller.

3.2 State machines of the UDP block

The UDP block contains most of the control logics, and it uses two main and two sub state machines. Main state machines *Tx state* and *Rx state* are shown in figure 7 and the sub state machines helping in reading or writing headers are shown in figure 8. The header state machines are used when the main state machines are in states write/read_IP/UDP_headers.

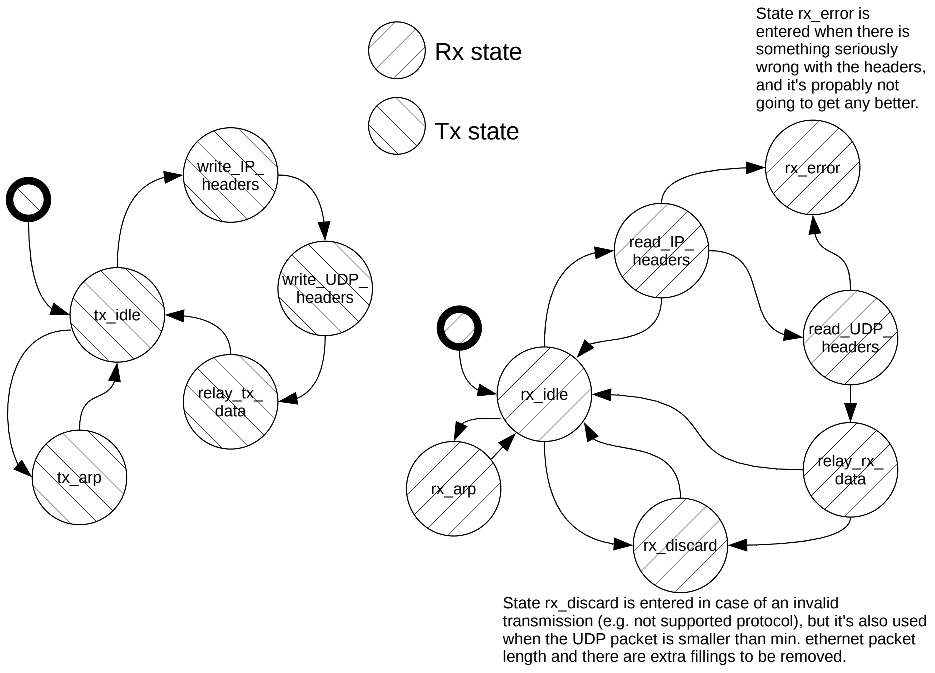
Rx state
Tx state
State rx_error is entered when there is something seriously wrong with the headers, and it's propably not going to get any better.
write_IP_ headers
write_UDP_ headers
tx_idle
relay_tx_ data
tx_arp
read_IP_ headers
rx_error
read_UDP_ headers
rx_idle
relay_rx_ data
rx_arp
rx_discard
State rx_discard is entered in case of an invalid transmission (e.g. not supported protocol), but it's also used when the UDP packet is smaller than min. ethernet packet length and there are extra fillings to be removed.

Figure 7: *State diagrams of the two main states.*

3.3 Sending and receiving

Figures 9 and 10 illustrate how sending and receiving data really happens. I'm not an expert on UML, so don't mind any unintended semantics. These diagrams are only trying to show the order in which events take place.

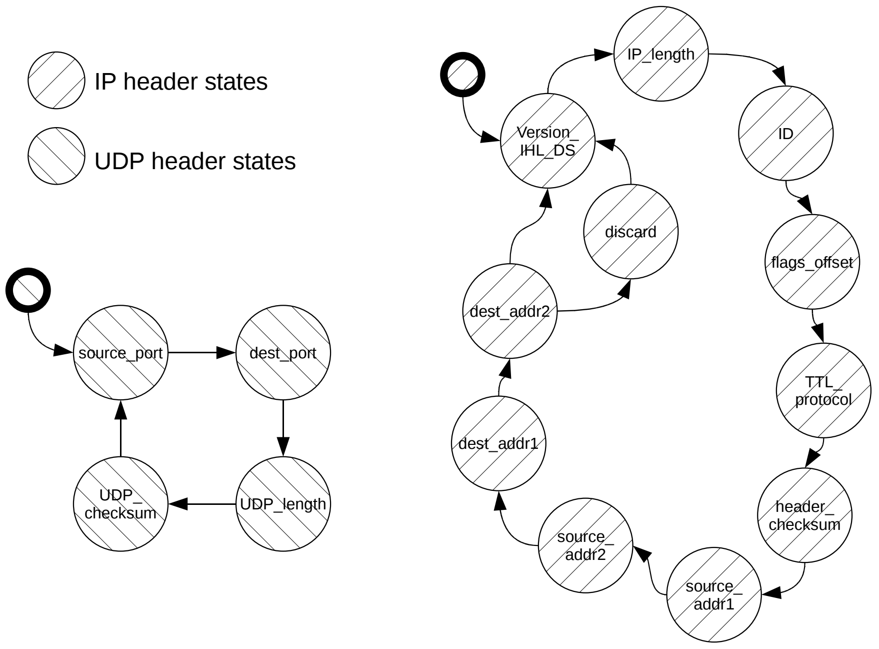

Figure 8: *State diagrams of the header manipulation states.*

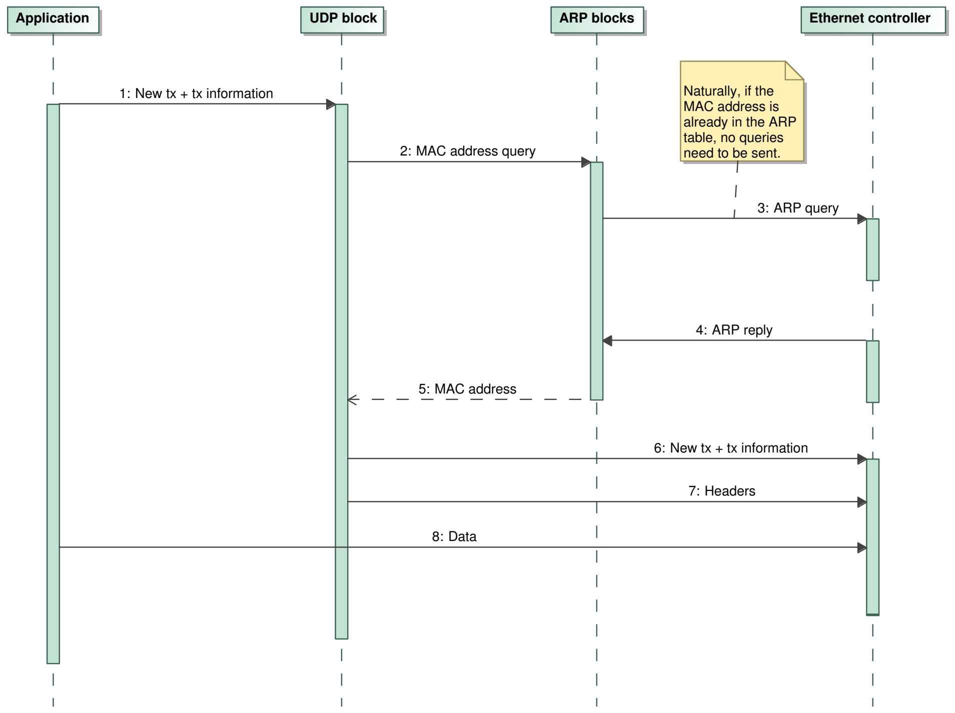
Application
UDP block
ARP blocks
Ethernet controller
Naturally, if the MAC address is already in the ARP table, no queries need to be sent.
1: New tx + tx information
2: MAC address query
3: ARP query
4: ARP reply
5: MAC address
6: New tx + tx information
7: Headers
8: Data

Figure 9: *Sequence diagram about sending a packet via UDP/IP.*

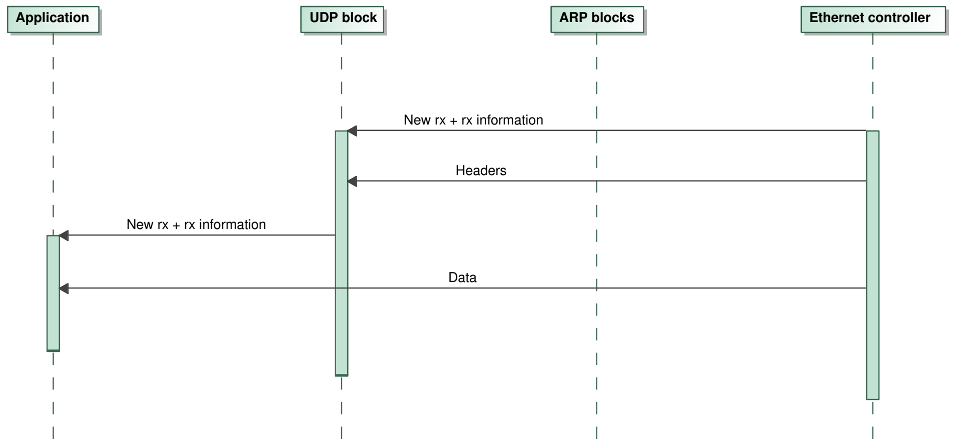

Figure 10: *Sequence diagram about receiving a packet via UDP/IP.*

References

[1] Ashley Partis, Jorgen Peddersen. VHDL IP Stack Project. University of Queensland, Australia, 2001. URL: http://www.itee.uq.edu.au/~peters/xsvboard/index.html