Making the SEcube™ Speaking Ethernet

Project Documentation

Release: February 2020

Proprietary Notice

The following document offers information, which is subject to the terms and conditions described hereafter.
While care has been taken in preparing this document, some typographical errors, error or omissions may have occurred. We reserve the right to make changes to the content and information described herein or update such information at any time without notice. The opinions expressed are in good faith and while every care has been taken in preparing this document, some typographical errors, error or omissions may have occurred. We reserve the right to make changes to the content and information described herein or update such information at any time without notice. The opinion expressed are in good faith and while every care has been taken in preparing this document.

Authors

Alberto CARBONI albertocarboni95@yahoo.it
Alessio CIARCIÀ alessiociarc@gmail.com
Jacopo GRECUCCIO jacopogrecuccio1994@gmail.com
Paolo PRINETTO *(President, CINI Cybersecurity National Lab)* paolo.prinetto@polito.it
Gianluca ROASCIO *(CINI Cybersecurity National Lab)* gianluca.roascio@polito.it
Antonio VARRIALE *(Managing Director, Blu5 Labs Ltd)* av@blu5labs.eu
Lorenzo ZAIA lorzai93@gmail.com

Trademarks

Disclaimer

Contents

1 Overview

1.1 Ethernet

Since its advent in 1973, the Ethernet protocol has been the bearer of technological innovation, spreading progressively and very quickly. With Ethernet, we mean the leader technology in the LANs (Local Area Network), with also implications in WANs (Wide Area Network) and MANs (Metropolitan Area Network). It was marketed in 1980 and standardized few years later by IEEE with its 802.3 (1983), which defines the levels 1 and 2 of the ISO-OSI stack. The standard include cable variants and physical signaling. The systems communicating over Ethernet divide the data flow into fragments which are called *frames*.

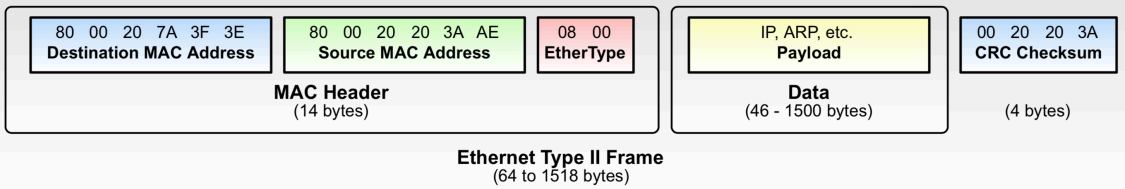

Figure 1: Ethernet Type II Frame

The most common frame type is the Ethernet Type II, the one shown in Figure 1. A frame contains the Source and the Destination MAC addresses (or Ethernet addresses) of 6 bytes each, the EtherType field (2 bytes), the Payload and the CRC error checking field, which allows discarding the damaged frames at reception. Ethernet does not provide re-transmission of packets in case of failure; the task is delegated to the higher levels of the ISO-OSI stack (e.g., TCP). As specified, the frame size can vary from a minimum of 60 up to a maximum of 1518 bytes. With the presence of the VLAN tag it can be extended up to 1522 bytes. The EtherType field hosts the identifier related to the upper level protocol encapsulated in the payload: for instance, `0x800` identifies IPv4, `0x806` indicates ARP (Address Resolution Protocol).

1.2 The Project

The present project aims to provide the **SE*cube*™ DevBoard** with Ethernet compatibility, enabling its network communication capabilities. In particular, the board has a LAN9211 Ethernet controller welded on, and connections with the **SE*cube*™ Chip** are viated through its FPGA[1], with a design that takes into account different aspects.

[1]For reference, see https://www.secube.eu/site/assets/files/1152/secube_sdk_v1_4_1_wiki_rel_009.pdf, **Appendix A - SE*cube*™ Data Sheet**

2 LAN9211 Controller

2.1 General description

The LAN9211 by Microchip™ is an Ethernet general-purpose platform-independent controller for embedded applications, implemented on a single chip. The large number of features and high speed make it suitable for high performance and high throughput 16-bit applications. It is fully compatible with the IEEE 802.3 10BASE-T and the 802.3u 100BASE-TX standards. The chip includes the MAC and PHY hardware modules, easily adapting them to a 16-bit slave host bus interface; it also includes an architecture with FIFO reception and transmission buffers. The controller, as previously stated, is welded on the **SE*cube*™** DevBoard and connected to the **SE*cube*™** . However, it is not bonded directly to the MCU, but to the FPGA of the device.

2.2 Components

2.2.1 10/100 PHY

The PHY (PHYsical transceiver) refers to the hardware component required for implementing the first level (physical layer) of the ISO-OSI stack. It deals with connecting a level-2 device (data link layer), which is often referred to as a MAC, to a physical broker. More in detail, the Ethernet PHY is a chip that implements the *send* and *receive* hardware functions, acting as a link between the analog domain (electrical cable transmission specification) and the digital domain (link layer signaling). As in the description, the PHY in the LAN9211 controller is fully compatible with the 10BASE-T and 100BASE-TX specifications; it can therefore be configured to operate at 10Mbps or 100Mbps and in half duplex or full duplex modes. Among the various possibilities, it supports auto-negotiation for the automatic choice of speed and mode. It also includes an interface for reading and writing internal configuration registers (CSR). This interface is required because the registers linked to the PHY module are not memory-mapped, but only accessible in an indirect way. This interface is called MII, Media Independent Interface, and is defined by the 802.3 standard.

2.2.2 10/100 MAC

The MAC (Media Access Control) module controls the transmission and the reception of packets by taking advantage of two separate paths to obtain the best performance. A third path is exploitable to access the internal configuration registers. Like the PHY module, it is fully compatible with 10Mbps and 100Mbps Ethernet. The module also contains two buffers not directly accessible to the host, which carry out the transmission and reception functions linked to the MAC logic.

2.2.3 RX/TX FIFO

They represent the conduits within which packets and information between the MAC module and the host interface pass. They raise the degree of tolerance to disparate latencies to limit the overrun, adapting to the software (or hardware) limitations of the host.

2.2.4 Other components

For what concerns the other characteristic components of the LAN9211 Controller, those are:

- an interrupt controller, capable of generating interrupts of different polarities and buffer type;

- a general-purpose timer that can be used to issue the interrupt;
- a GPIO interface;
- a serial interface to EEPROM, programmable directly from host via bus, suitable for containing the mac address of the controller;
- various settings regarding low-power modes.

2.3 Transmission of a frame

According to the technical specifications[2], the frame must arrive to the TX data FIFO preceded by a specific header (Figure 2), which identifies it and provides its characteristics. Note that this header is not required by the hardware, but only by the LAN9211 software driver.

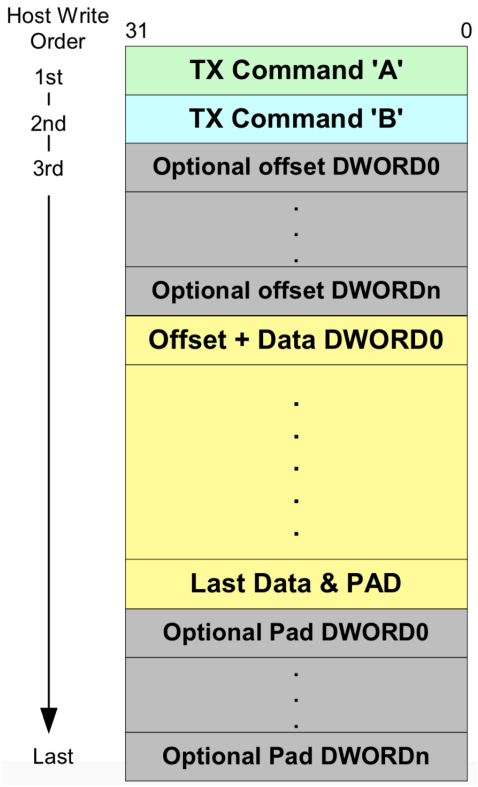

Figure 2: TX Buffer Format

TX commands A and B are both on 32 bits. The TX command A contains 5 fields of concern:

1. Buffer End Alignment: is a 2-bit field which controls the device behavior with a DMA controller. A zero is written here because the DMA is not used for the purpose;
2. Data Start Offset: is a 5-bit field which indicates to the device the number of bytes offset from the base address where the data actually begin. A zero is written in this field;
3. First Segment: is a 1-bit indication that the data trasfer marks the beginning of a single packet. This field is set;

[2] For additional details, please refer to the LAN9211 Reference Manual: http://ww1.microchip.com/downloads/en/DeviceDoc/00002414A.pdf

4. Last Segment: is a 1-bit field marking the data transfer as the end of a single packet. Packets are sent is a single data transfer operation, so also this bit is always set. This is the typical behavior of operating systems based on Linux, for example, in which system kernel buffers are used to hold packet data and almost always contain enough for the largest possible packet size (1514 bytes);

5. Buffer size: is a 11-bit field specifying the number of data bytes in the current data transfer not including the two command words (TX command A and TX command B);

The TX command B contains 2 major and 3 minor fields of concern:

1. Packet Tag: is a 16-bit field which contains a unique ID which can read back from the TX status word after the packet has been sent. We have chosen to use a simple counter so that the packets sent have a unique code, and that the variable can be easily queried in order to draw the total number of packets gone out;

2. Packet Length: contains the overall number of bytes in the current packet;

3. TX Checksum Enable: if set, the TX checksum offload engine will calculate a L3 checksum for the associated frame. This bit is reset;

4. Add CRC Disable: when set, the automatic addition of the CRC is disabled. This bit is reset;

5. Disable Ethernet Frame Padding: when set, this bit prevents the automatic addition of padding to an Ethernet frame of less than 64 bytes. This bit is reset;

2.4 Reception of a frame

As for the reception, the frame arrives as it is (Figure 3). The 4-byte CRC checksum is included in the arrival length and present at the end of each received frame.

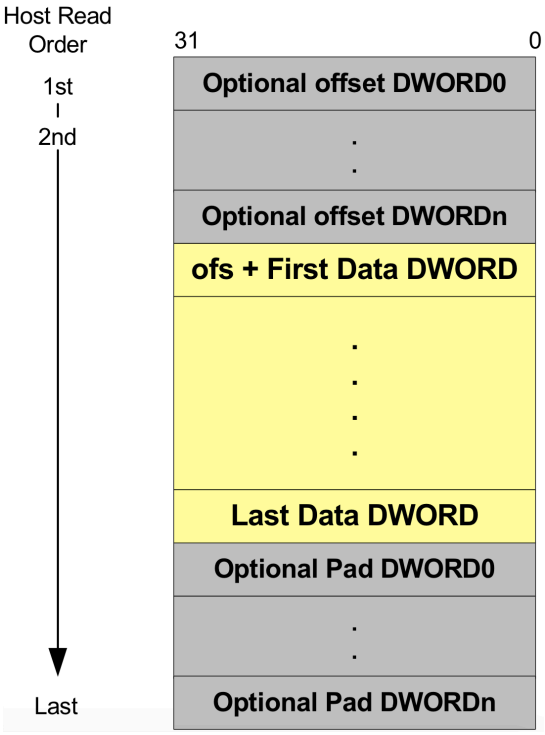

Figure 3: RX Buffer Format

3 IP core

3.1 Architecture

As we said, the LAN9211 Controller is reached by the **SE*cube*™ Chip** through a parallel interface which is connected to its internal FPGA. Thus, there is the need of architecting a custom IP core which manages the communication.
The IP core is compatible with the IP Manager FPGA configuration already provided[3], therefore it maintains the external interface defined that architecture. Internally, the core consists of 2 components:

1. The Main Controller;
2. The Programmable Input/Output (PIO) Controller.

These are reviewed one by one. The top-level schematic is shown in Figure 4.

3.1.1 Main Controller

Directly connected to the IP Manager, its action begins with the decoding of the opcode sent by the CPU to distinguish what operation to do. These include reading and writing from and to the configuration registers and sending and receiving Ethernet frames. It gets all the signals outgoing from the IP Manager, as shown in the diagram. As will be shown in the transactions, it is responsible for routing the packets to be sent by the data buffer to the LAN9211 controller, passing through the PIO Controller.

3.1.2 PIO Controller

The Programmed Input/Output (PIO) Controller manages the connection with the LAN9211 host bus. It interacts with it using the six signals of the schematic. The name is inspired by the LAN9211 manual; this always refers to operations such as PIO: PIO read (read from the controller registers or from the reception FIFO) and PIO write (write to the controller registers or to the transmission FIFO). It is therefore the PIO controller that routes a packet from the LAN controller and vice versa. Furthermore, it deals with the management of package formatting.

[3] Available at link https://www.secube.eu/resources/open-source-projects/, **IP-core Manager for FPGA-based design**.

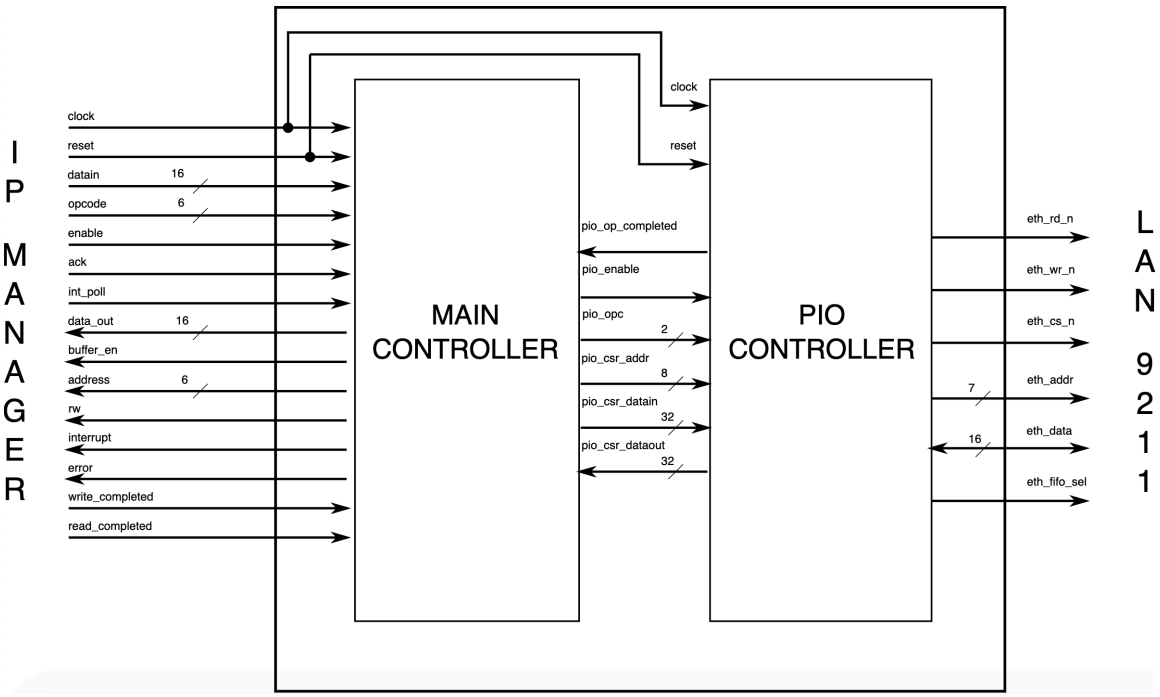

Figure 4: IP core Schematic

3.2 Transactions

Both the transactions are opened by the IP Manager specifying the address of the register to be reached, according to the direct addressable register map provided by the manufacturer[4].

3.2.1 CSR write transaction

The CSR write transaction is depicted in Figure 5. The 32-bit word is written to the data buffer in two stages (the data buffer is 16-bit wide). The transaction continues with the management of the lock/unlock mechanism typical of a polling transaction. Subsequently, the IP core writes the data to the specified LAN9211 address. Finally, the transaction is closed.

3.2.2 CSR read transaction

The CSR read transaction is depicted in Figure 6. After receiving the target address, the IP core keeps the CPU locked using, even in this case, the lock/unlock mechanism; then it sets the control signals for reading the register of the LAN9211 specified. Once the data is ready within the IP core, the CPU is unlocked and the write back operation to the CPU takes place.

[4] Please refer to the LAN9211 Data Sheet, Chapter 5, Table 5-1: DIRECT ADDRESS REGISTER MAP

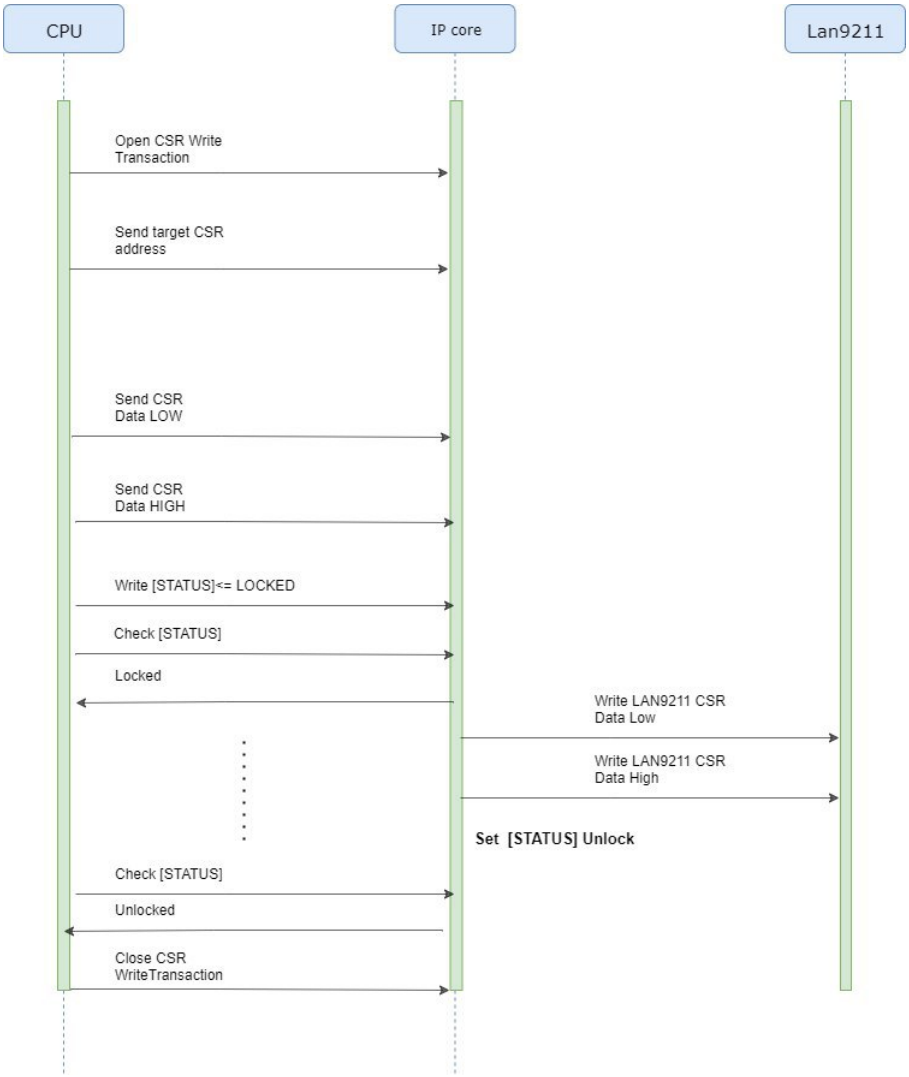

Figure 5: CSR Write Transaction

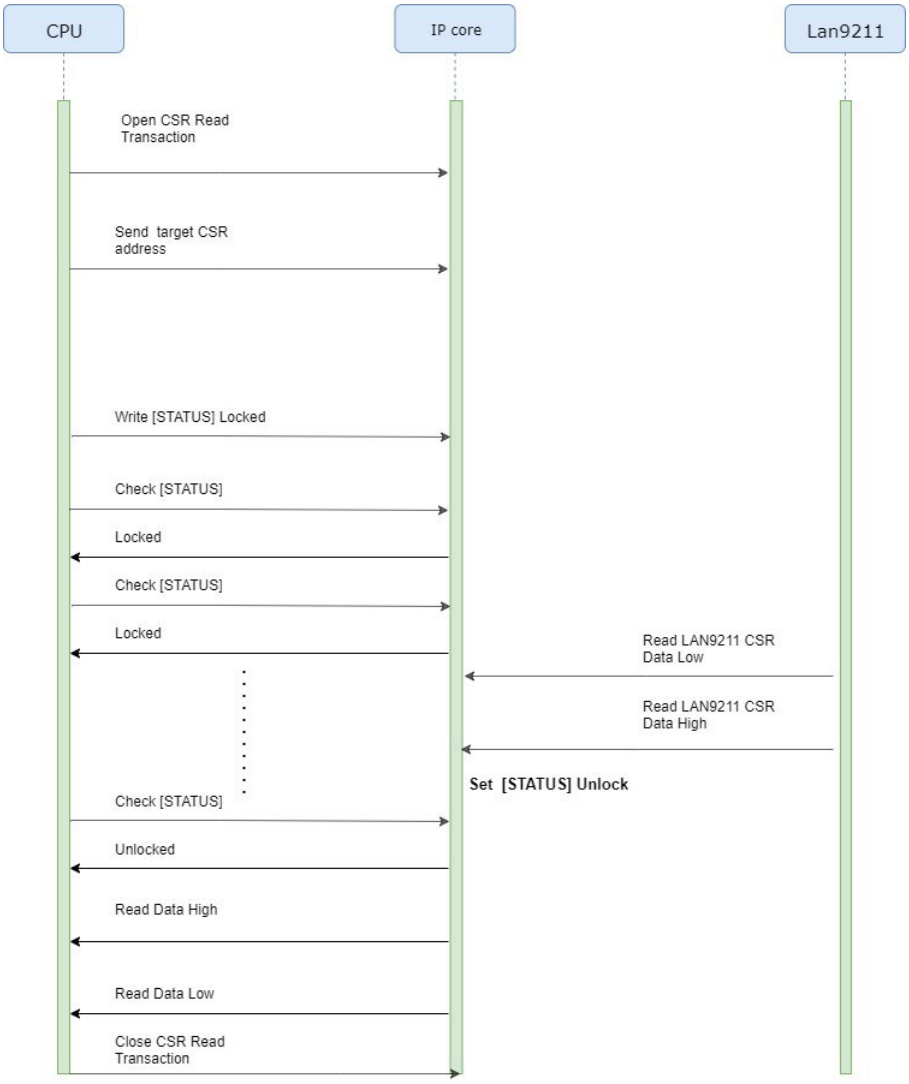

Figure 6: CSR Read Transaction

4 API

In this Section, the communication protocol provided by the Application Programming Interface is explained. This is divided into *high-level* and *low-level* API; the former is used to configure the system and to start a transmission, the latter to take care of the communication with the IP Manager. All the macros for each addressable register are defined in the "lan9211.h" file, which is provided in the project material.

4.1 Low-level API

The low-level API consists in two functions which allow the upper layers to write and to read LAN9211 Controller's internal registers. Each register is 32-bit wide and it is addressable via a 16-bit value between `0x50` and `0xFC`[5].

```
void ETH_SMB_PIOWrite(uint16_t lan9211_reg, uint32_t data)
```

This function is used to write a word of 32 bits into a LAN9211 register. The following parameters are required:

- `lan9211_reg`: the address of the required register;
- `data`: the value to be written.

```
void ETH_SMB_PIORead(uint16_t lan9211_reg, uint32_t *data)
```

This function is used to read a word of 32 bits from a LAN9211 register. The following parameters are required:

- `lan9211_reg`: the address of the required register;
- `data`: the pointer to the address where the data will be stored.

4.2 High-level API

They provide the following services to the user:

- Initialize the LAN9211 controller;
- Send an Ethernet frame;
- Receive an Ethernet frame;
- Read/Write CSR registers;
- Read/Write PHY registers;
- Read/Write MAC registers;
- Dump all LAN9211 registers.

[5] Please refer to the LAN9211 Data Sheet, Chapter 5, Table 5-1: DIRECT ADDRESS REGISTER MAP

4.2.1 Basic Functions

With these functions ,it is possible to use the LAN9211 Controller with a basic configuration chosen to meet the architectural constraints.

```
uint8_t lan9211_init()
```

This function initializes the LAN9211 controller with a default configuration, which includes the following steps:

1. Perform an HW soft Reset;
2. Assure that the internal PHY is running;
3. Set Automatic Flow Control;
4. Turn on GPIO leds;
5. Disable and clear interrupts;
6. Enable flow control and pause frame time;
7. Set default MAC Address;
8. Initialize TX parameters;
9. Initialize RX parameters;
10. Enable TX and RX;
11. Initialize PHY parameters;
12. Start auto negotiation.

```
int lan9211_sendFrame(const uint8_t *frame, uint16_t length);
```

This function is used to send an Ethernet frame. The following parameters are required:

- `frame`: the pointer to the frame to be sent;
- `length`: the length of the frame.

It returns an integer value, which reports the status of the execution:

- Negative value: an error occurred in the transmission of the frame;
- 0: no frame sent;
- Positive value: length of the transmitted frame.

The following steps are performed:

1. Add padding (if necessary);
2. Generate TX command A and TX command B;
3. Send the two commands followed by the frame;
4. Check for errors.

```
int lan9211_receiveFrame(uint8_t *buffer);
```

This function is called to receive a frame. The following parameters are required:

- `buffer`: the pointer to the address where the received frame will be stored.

It returns an integer value, which reports the status of the execution:

- Negative value: an error occurred in the reception of the frame;
- 0: no frames collected;
- Positive value: length of the received frame.

The following steps are performed:

1. Read `RX_FIFO_INF` register to check if there is used space (occupied by the incoming frame);
2. Check the status port for errors;
3. Read the whole frame and store it.

```
void CSRregsDump()
```

This functions reads all the main PIO registers and store their value on a struct variable called `regs`.

4.2.2 Advanced Functions

These functions can be used to change the system configuration. The use of this function could compromise the functioning of the system, hence use them only after understanding the hardware architecture.

```
void SetMAC_Reg(uint8_t idx, uint32_t data)
```

This function is used to write an internal register of the MAC. The following parameters are required:

- `lan9211_reg`: the address of the required MAC register;
- `data`: the value to be written.

```
void GetMAC_Reg(uint8_t idx, uint32_t *data)
```

This function is used to read an internal register of the MAC. The following parameters are required:

- `idx`: the address of the required MAC register;
- `data`: the pointer to the address where the data will be stored.

```
void SetPHY_Reg(uint8_t idx, uint32_t data)
```

This function is used to write to a PHY register of the LAN controller. The following parameters are required:

- `idx`: the address of the required PHY register;
- `data`: the value to be written.

```
void GetPHY_Reg(uint8_t idx, uint32_t *data)
```

This function is used to read a PHY register of the LAN controller. The following parameters are required:

- `idx`: the ID of the required PHY register;
- `data`: the pointer to address where the data will be stored.

```
void lan9211_CSR_write(uint16_t lan9211_reg, uint32_t data)
```

This function is used to write a value into a CSR (Control and Status Register) of the LAN9211 Controller. The following parameters are required:

- `lan9211_reg`: the address of the required PIO register;
- `data`: the value to be written.

```
void lan9211_CSR_read(uint16_t lan9211_reg, uint32_t *data)
```

This function is used to read the value of a CSR (Control and Status register) of the LAN9211. The following parameters are required:

- `lan9211_reg`: the address of the required PIO register;
- `data`: the pointer to the address where the data will be stored.

5 User Manual

The following Section is committed to explain how to use the Ethernet IP core, starting from its insertion in the development environment. A brief test procedure of the functionalities is also provided.

5.1 Core installation

To start with, you need to add the core to the environment, that is the IP Manager architecture. Hence, you need a custom Device-Side **SE*cube*™** project opened in Eclipse, ready to work with the FPGA available on the **SE*cube*™** . To do so:

1. Download the files related to the IP Manager from the **SE*cube*™** Open Projects site[6].

2. Create a new Device-Side project by following the steps that are listed in the related documentation[7], Section 5.2.4. Be sure to add all the necessary files to communicate with the FPGA.

3. Create a new Lattice Diamond™ project as specified into Section 5.3.1 of the IP Manager project documentation. When you are asked to import the VHDL files, be sure to include the files named “CONSTANTS.vhd”, “DATA_BUFFER.vhd”, “IP_MANAGER.vhd”, which you downloaded at point 1. Moreover, you must add all the VHDL files related to Ethernet core project. These files are located inside the folder named ”VHDL”. In more detail, a second IP core that turns on the LEDs of the DevBoard has been added for the test phase. The .vhd files related to this are present in the same folder. This will cooperate with the primary IP core responsible for Ethernet.

4. Once the file are added to the project, right click the ”LPF Constraint Files” folder, select ”Add » Existing File...” and select the file ”eth.lpf”; this file contains the pin configuration to correctly setup the connection between the FPGA and the LAN9211 controller. Select the option ”Copy file to Implementation’s Source directory” and click ”Add”. Finally, right click the new added file and select ”Set as Active”. Please note that, if you want to keep your default constraint file, you can only copy the lines beginning with LOCATE COMP ”fpga_eth_” adding these to your .lpf existing file.

5. Now synthesize the project (check that no timing errors are present) and produce the files containing the bitstream by following the steps contained in Sections 5.3.2 and 5.3.3 of the IP Manager documentation. You should have obtained two files with name ending with “_algo.c” and “_data.c”, which will be used later.

6. Proceed by adding everything to the Device-Side project, as explained in Section 5.4 of the IP Manager documentation. When you are asked to substitute the content of the two arrays in the file “TEST_FPGA.h”, use the files obtained previously.

Now the bitstream describing the architecture is statically saved in the firmware to be programmed into the device. To use its functionalities, you need to include the files containing the correlated API, with the following steps:

1. To import the necessary files in your Device-Side project, select ”File » Import...”, then “Filesystem” and press “Next”.

[6] https://www.secube.eu/site/assets/files/1164/secube_fpga_ip_manager_release.zip
[7] https://www.secube.eu/site/assets/files/1164/wiki_fpga_-_rel006_201910.pdf

2. Browse to the directory where the API libraries for Ethernet are located, inside the folder named ”C”.

3. Select the files ”lan9211.c”, ”lan9211.h”, ”eth_smb.c”, ”eth_smb.h”, ”eth_smb_util.c”, ”eth_smb_util.h”, ”blinker.c” and ”blinker.h”. You might want to set also “Destination Folder” to “SEcubeDevBoard/Application/src” and then press “Finish”.

Now the core is correctly inserted in the environment. You can now create your own program, in the file ”main.c”, and use the core to send and receive data over Ethernet, as described in the API section.

5.2 Send and receive your first Ethernet frame

A test to prove the integrity of the system is proposed in this Subsection. This is a simple test exploiting ARP (Address Resolution Protocol) and its functionalities. What you need is:

- Your **SE*cube*™** devboard ready to be programmed;
- A Windows/Linux PC (or Laptop or whatever device) equipped with an Ethernet card;
- The software Wireshark[8]. Wireshark is an open-source traffic analyzer used for many purposes including network troubleshoot, software and protocols development, education. It can be very useful in this context as it is able to intercept the complete communication on a network interface.

5.2.1 Linux

The test consists in sending an ARP request packet from the **SE*cube*™** to the the network card of the PC, addressed to its IPv4 address. We expect the PC to reply with an ARP reply packet telling its MAC address. The last byte of this physical address will be printed on the **SE*cube*™** devboard leds.

First of all, make sure you have correctly followed all the steps described in the previous section. Then, follow the steps below to correctly setup the code for your **SE*cube*™** device.

1. In the ”main.c” file of your project, include ”lan9211.h” and ”eth_smb_util.h”:

```
#include "lan9211.h"
#include "eth_smb_util.h"
```

2. In the `main()` function, declare the following local variables:

```
uint8_t rx_frame[1518];
int retval = 0;
```

   The first is the buffer where your received packets will be stored, while the second is an integer to capture the return value of a function as explained later.

3. After the `device_init()` function, add a call to the `lan9211_init()` function.

```
lan9211_init();
```

   This will take care of preparing the device to communicate with the LAN9211 Controller. To ensure that the configuration is successful, **it is mandatory to connect an Ethernet cable to both sides of the link, the SE*cube*™ DevBoard and the network adapter card of the PC you are using**, otherwise the system will be locked in its initialization phase until a cable is plugged. Make sure to connect a cable when your device is running.

[8] Wireshark is available at the following link: https://www.wireshark.org/download.html

4. Add one call to the `lan9211_sendFrame()` function, passing the frame to be sent as the first parameter and the frame length as the second parameter. Set `simple_arp_packet` as first parameter, which is found in the file ”eth_smb_util.h”. This is a 42-byte ARP request packet: the sender’s MAC address is 00:aa:bb:cc:dd:ee, the destination MAC is the broadcast address (ff:ff:ff:ff:ff:ff). This attempts to resolve the specified IPv4 address (indicated as ”target IP address” in the comments) by receiving the MAC address associated with it as a response. The current IP address set as destination is 10.10.10.10; you may change this at your convenience.
   The line to be added will therefore be:

```
lan9211_sendFrame(simple_arp_packet, sizeof(
    simple_arp_packet));
```

5. Add finally the following lines to your code:

```
for (;;) {
    retval = lan9211_receiveFrame(rx_frame);
    if (isArpReply(rx_frame, retval)) {
        LEDS_on(rx_frame[11]);
        break;
    }
}
```

   These are the ones performing the reception of the ARP reply packet and showing the source MAC address on the LEDs.

Now the system is ready to be built and the SE*cube*™ programmed. To set a static IP address using your Linux operating system, follow the steps below:

1. Go to Settings > Network and select your Ethernet interface. This can be under the name ”Wired” or others depending on your system;
2. Click the gear icon to configure it;
3. Select the IPv4 tab;
4. Under ”IPv4 Method” select ”Manual”;
5. In the ”Address” field type 10.10.10.10, or a different one according to the one you decided to use as the destination IP address in the `sample_arp_packet`;
6. In the ”Netmask” field type 255.255.255.0;
7. in the ”Gateway” field type 10.10.10.1 or whatever you may like more;
8. Apply the changes.

The procedure may be a little different for different versions of Linux. You might prefer to set a static IP using terminal.
Now you can turn on the SE*cube*™ and set up a link-to-link connection with the PC. The MAC address of the Ethernet interface can be checked opening a terminal and typing `ifconfig`. Only the last byte is printed on the leds.
To see if packets are received on the PC side, open Wireshark once correctly installed. You will be asked for the network adapter card to be analyzed, choosing from the available ones. The capture of the packets on the selected interface starts automatically. If you want to stop just click on the

red square at the top (stop capture); to restart capturing, click on the blue fin (start capture). When the capture is active, restart the **SE*cube*™** . After about forty seconds, the time required for reprogramming the FPGA, Wireshark will begin to detect traffic on the interface. The ARP packet sent with **SE*cube*™** should now be visible. You can apply a filter in the top bar to make the packets that meet the indicated parameters appear on the screen. For example, to see the ARP packet sent by the **SE*cube*™** , you can apply a filter to the MAC address by typing `eth.addr == 00:aa:bb:cc:dd:ee`, i.e., the source Ethernet address of the ARP packet.
If all the steps have been carried out correctly, the system should reply to the ARP request packet with an ARP reply. The LEDs of the **SE*cube*™** DevBoard light up correspondingly with the last byte of the MAC address of the PC network interface.

5.2.2 Windows

The test consists in sending an ARP request packet from the **SE*cube*™** to the the network card of the PC, addressed to its IPv4 address. This time, differently from Linux, we do not expect a response from Windows for a problem with the static IP address setting. This comes from the use of APIPA (Automatic Private IP Addressing), an IPv4 address starting with 169.254 automatically assigned to the network interface when DHCP is not enabled, as in the case of a static IP configured. The problem is that this automatic address is the one preferred over the static IP configured manually. This means that the PC is not able to answer with an ARP reply to our ARP request, directed to 10.10.10.10, because to the network interface is now assigned another IP address.
The test now plans to print the MAC address of the PC coming from an ARP request packet, not from the ARP reply one. In fact, once a link connection is established, the PC sends several broadcast ARP requests trying to resolve the MAC address related to its default gateway IP address. The last byte of this physical address will be printed on the **SE*cube*™** devboard LEDs.
First of all, make sure you have correctly followed all the steps described in the previous section. Then, follow the steps below to correctly setup the code for your **SE*cube*™** device.

1. In the ”main.c” file. include the ”lan9211.h” and ”eth_smb_util.h”:

```
#include "lan9211.h"
#include "eth_smb_util.h"
```

2. In the main function declare the following local variables:

```
uint8_t rx_frame[1518];
int retval = 0;
```

   The first is the buffer where your received packets will be stored, while the second is an integer to capture the return value of a function as explained later.

3. After the ”device_init” function, add a call to the ”lan9211_init” function.

```
lan9211_init();
```

   This will take care of preparing the device to communicate with the LAN9211 Controller. To ensure that the configuration is successful **it is mandatory to connect an Ethernet cable to both sides of the link, the SE*cube*™ DevBoard and the network adapter card of the PC you are using**, otherwise the system will be locked in its initialization phase until a cable is plugged. Make sure to connect a cable when your device is running.

4. Add one call to the `lan9211_sendFrame()` function, passing the frame to be sent as the first parameter and the frame length as the second parameter. Set `simple_arp_packet` as first parameter, which is found in the file ”eth_smb_util.h”. This is a 42-byte ARP request

packet: the sender's MAC address is 00:aa:bb:cc:dd:ee, the destination MAC is the broadcast address (ff:ff:ff:ff:ff:ff). This attempts to resolve the specified IPv4 address (indicated as "target IP address" in the comments) by receiving the MAC address associated with it as a response. The current IP address set as destination is "10.10.10.10", but this is not important since we are not expecting an answer. The line to be added will therefore be:

```
    lan9211_sendFrame(simple_arp_packet, sizeof(
       simple_arp_packet));
```

5. Add finally the following lines to your code:

```
for (;;) {
    retval = lan9211_receiveFrame(rx_frame);
    if (isArpRequest(rx_frame, retval)) {
        LEDS_on(rx_frame[11]);
        break;
    }
}
```

These are the ones performing the reception of the ARP request packet and showing the source MAC address on the LEDs.

Now the code is ready to be built and the **SE***cube*™ programmed.
Now you can set up a link-to-link connection between the **SE***cube*™ and the PC. The MAC address of the Ethernet interface can be checked opening a terminal and typing `ipconfig \all` when the connection is active, under the name "Physical Address" of the related Ethernet interface card.
To see if packets are received PC side, open Wireshark once correctly installed. You will be asked for the network adapter card to be analyzed, choosing from the available ones. The capture of the packets on the selected interface starts automatically. If you want to stop just click on the red square at the top (stop capture); to restart capturing, click on the blue fin (start capture).
When the capture is active, turn on or restart the **SE***cube*™ . After about forty seconds, the time required for reprogramming the FPGA, Wireshark will begin to detect traffic on the interface. The ARP packet sent with **SE***cube*™ should now be visible. You can apply a filter in the top bar to make the packages that meet the indicated parameters appear on the screen. For example, to see the ARP packet sent by the **SE***cube*™ , you can apply a filter to the MAC address by writing `eth.addr == 00:aa:bb:cc:dd:ee`, i.e., the source Ethernet address of the ARP packet.

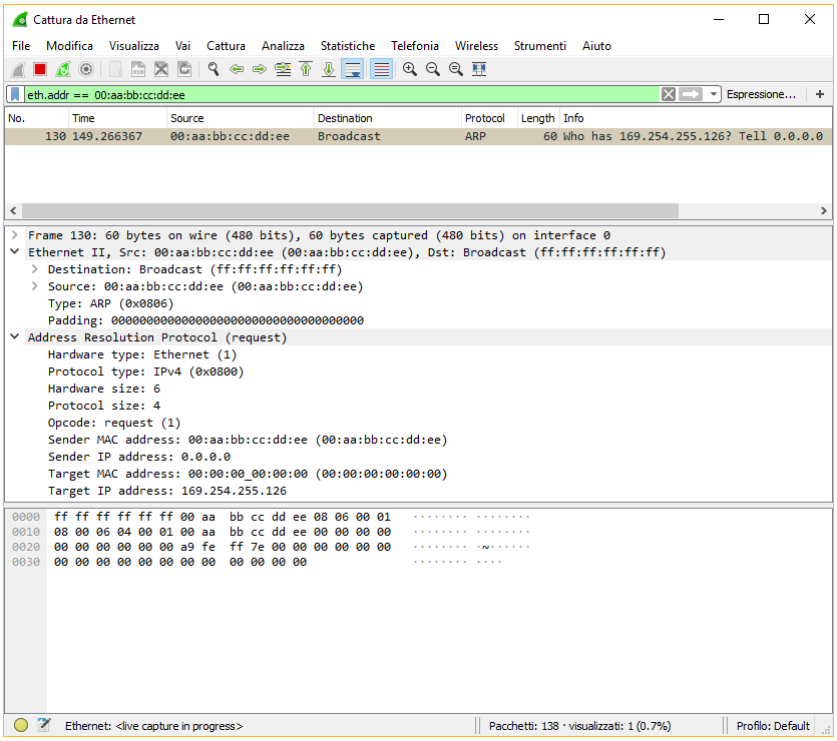

Figure 7: Wireshark: Reception of the ARP Request Packet

The Figure 7 shows the capture performed using Wireshark. The packet can be inspected in deep checking all the information needed. In this way the tool can be used to verify the correct reception of the frames by the selected Ethernet interface, therefore the correct functioning of the **SE*cube*™** on the sending side.
If all the steps have been carried out correctly the leds of the **SE*cube*™** DevBoard light up correspondingly with the last byte of the MAC address of the PC network interface.