Politecnico
di Torino

Polytechnic of Turin

A.Y. 2022/2023

Master degree course
Electronic engineering

Transaction-Level Ethernet PHY modelling in SystemC

Advisor	Co-advisor	Candidate
Luciano Lavagno	Dario Soldi	Loris Panaro

Abstract

The availability of Advanced Driver Assistance Systems (ADAS), the pressing demand for connectivity and increasingly complex and content-rich infotainment systems, have had, as first consequence, a considerable increase in data traffic and in the demand for solutions capable of ensuring high throughput and low latency. Among these technological advancements, the adoption of Ethernet as a communication protocol within automotive systems has emerged as a game-changer. Ethernet, a well-established standard in the realm of computer networking, offers numerous advantages in terms of bandwidth, scalability, and flexibility, making it an attractive solution for in-vehicle communication networks.

However, being an innovative communication protocol in the automotive industry, it leads to brand new challenges: to define validation methodologies and strategies that allow to easily test the integrability of Engine Control Unit (ECU) interconnected on Ethernet and to perform high specific simulations on a virtual environment among all the involved peripherals of the vehicle.

The aim of this master thesis is to deeply analyze how the Ethernet communication protocol can be implemented on an automotive environment according to OPEN Alliance specifications, to develop virtual Ethernet PHY device TJA1101B provided by NXP Semiconductor using SystemC 2.3.3 library and SCML2 2.8.0 library (provided by Accelera and Synopsys respectively) and to simulate its behaviour in the Synopsys Virtual Development Kit (VDK) tool. The project has been carried out in collaboration with Punch Softronix and Polytechnic of Turin.

Acknowledgments

I would like to thank my relator, professor Luciano Lavagno, who has shown great willingness both during the teaching period and during this thesis period. Moreover, I am truly thankful for the invaluable opportunity he has provided me with, allowing me to get in touch with Punch Torino and to develop this thesis together with its group.
I would like to extend a special thank to Punch Torino for the cooperation of this thesis, which gave me the possibility to get in touch with its business reality and to better understand how a real life project comes into being; in particular, a sincere thank goes to Dario Soldi, member of Punch Torino and co-advisor, whom I hold in high esteem, and with whom I have shared this thesis journey. He has consistently been extremely helpful with assistance and clarifications at every stage of the work and pushed me to do better every day.
A personal thank goes to my girlfriend Clara, exquisit person who deeply knows me and with whom I have shared the most meaningful experiences of my life. Involving her in such a moment of joy only fills my heart with happiness.
In addition to that, I would like to thank all my friends who I have known during Master Degree and during High School, whose friendship has last over the years.
Last but not least, I would like to thank and dedicate this achievement to my father Massimo and my mother Luciana. Thanks to them, I have been able to start and accomplish my university carreer, which helped me growing as an engineer and as a person.
My special thanks go to all of them.

Contents

Acknowledgments **ii**

1 Digital Twin and SystemC modeling **1**
1.1 Digital Twin creation and behaviour 1
1.1.1 Examples of Digital Twin applications 2
1.1.2 Creation process of the Digital Twin 2
1.1.3 Data acquisition of the Digital Twin 4
1.2 SystemC language 6
1.2.1 Overview 7
1.2.2 Module hierarchy 7
1.2.3 SystemC ports and signals 8
1.2.4 SystemC processes 10
1.2.5 SystemC events 12
1.2.6 SystemC time modeling 12
1.2.7 SystemC simulation scheduler 13
1.2.8 Transaction Level Modeling 15
1.2.9 SCML 2.8.0 library 17
1.3 Comparison between Digital Twin and SystemC model 18

2 General overview of Automotive Ethernet **19**
2.1 Limitations of CAN and LIN protocols 19
2.2 Ethernet: a new suitable solution 19
2.3 From standard Ethernet to Automotive Ethernet 20
2.3.1 Standard Ethernet 20
2.3.2 Automotive Ethernet 25
2.4 100BASE-T1 technology 26
2.4.1 Cabling and wiring 26
2.4.2 Noise reduction 27
2.4.3 Echo reduction 28
2.4.4 Bit encoding for EMI constraints 29
2.4.5 Link startup 31

2.5 1000BASE-T1 technology . . . 32
2.5.1 Interface circuitry . . . 32
2.5.2 PAM5 modulation and wiring . . . 33
2.5.3 Link startup . . . 34
2.6 TC10 specification for Sleep/Wakeup . . . 36

3 Punch Electronic Platform and ASIC specifications 38
3.1 Punch Electronic Platform specifications . . . 38
3.2 TJA1101B overview and pinout . . . 39
3.3 TJA1101B PHY . . . 40
3.4 SMI interface and registers . . . 41
3.5 Pinout . . . 42
3.6 Functional block and diagram . . . 45
3.7 SMI registers . . . 46
3.7.1 SMI frame . . . 47
3.8 Hardware configuration . . . 47
3.9 MII signal encoding . . . 48
3.10 RMII signal encoding . . . 49
3.11 Reverse MII . . . 50
3.12 Loopback . . . 51
3.13 MDI interface circuit . . . 52
3.14 Finite state machine . . . 53
3.14.1 DUT operations allowed . . . 54
3.14.2 Relevant timing . . . 55
3.15 Undervoltage detection methods . . . 56
3.16 Overtemperature and Temperature warning methods . . . 56
3.17 Interrupt handling . . . 56
3.18 PHY wakeup concept . . . 57
3.19 PHY sleep concept (LPS) . . . 58

4 Model requirements 59
4.1 TJA1101B hierarchy . . . 60
4.2 Pinout . . . 61
4.3 MII and MDI interface . . . 61
4.4 SMI interface . . . 62
4.5 SMI registers . . . 62
4.6 Internal and external loopback . . . 63
4.7 Remote loopback . . . 64
4.8 Ethernet frame class description . . . 65
4.9 Finite state machine . . . 65
4.10 Undervoltage detection methods . . . 66

4.11 Overtemperature and temperature warning detection methods 66
4.12 Interrupt handling 67
4.13 Local/Remote wakeup 67
4.14 Low Power Sleep (LPS) 67
4.15 Ethernet sniffers 67
4.16 Live demostration 69

5 Test environment **70**
5.1 Testbench structure 70
5.2 Test list 72
5.3 Undervoltage detection 73
5.4 Overtemperature and temperature warning detection 73
5.5 Interrupt event 74
5.6 Local/Remote wakeup handling 74
5.7 LPS detection 75
5.8 Results of listed tests 76
5.9 Ethernet sniffer dumps 77
5.10 Code coverage 81
5.11 Functional verification 82
5.12 Live sniffing demostration 83
5.13 Virtual Development Kit implementation 83

6 Final conclusions **87**

Bibliography **88**

List of Tables

3.1 Pins functionality 44
3.2 SMI registers numbering and functionality 47
3.3 SMI frame structure 47
3.4 Pins for hardware configuration 48
3.5 Media Independent Interface (MII) encoding for transmission 48
3.6 MII encoding for reception 48
3.7 Reversed Media Independent Interface (RMII) encoding for transmission 49
3.8 RMII encoding for reception 50

5.1 State transitions list 72

List of Figures

1.1 Digital Twin creation . . . 3
1.2 Enabling technology for data collection . . . 6
1.3 RTL description and TLM differences . . . 15
1.4 TLM socket representation . . . 16

2.1 OSI model layers representation . . . 21
2.2 Data-link frame format . . . 21
2.3 Ethernet types relationship . . . 22
2.4 IP frame format . . . 22
2.5 TCP frame format . . . 24
2.6 UDP frame format . . . 25
2.7 100BASE-TX connection . . . 27
2.8 100BASE-T1 connection . . . 27
2.9 TJA1101B PHY application circuit for noise reduction . . . 27
2.10 Block diagram of interface circuitry 100BASE-T1 . . . 28
2.11 Echo cancellation block diagram . . . 29
2.12 Generation of the 2T ternary pair vector . . . 30
2.13 From MII to MDI conversion process . . . 30
2.14 100BASE-T1 link startup process . . . 31
2.15 Block diagram of interface circuitry 1000BASE-T1 . . . 33
2.16 PAM5 modulation. Source [2] . . . 34
2.17 1000BASE-T1 link startup process . . . 35
2.18 TC10 Sleep/Wakeup mechanism . . . 37

3.1 TJA1101B overview . . . 40
3.2 Ethernet MII-MDI peripheral . . . 41
3.3 SMI interface and control blocks . . . 41
3.4 TJA1101B pinout . . . 45
3.5 TJA1101B block diagram . . . 46
3.8 Reverse MII configuration . . . 51
3.9 internal loopback . . . 51
3.10 External loopback . . . 51

3.11 Remote loopback 52
3.12 Media Dependent Interface (MDI) interface circuit 52
3.13 Finite state machine diagram 53
3.14 TC10 wakeup implementation 58

4.1 Pinout implementation Green:implemented Orange: abstracted Red: omitted 61
4.2 TLM representation of MII and MDI interface 62
4.3 SMI TLM memory structure 63
4.4 Block diagram of the virtual internal and external loopback 64
4.5 Block diagram of the virtual remote loopback 64

5.1 Test environment 70
5.2 Test process flowchart 71
5.3 Undervoltage on V_bat 73
5.4 Overtemperature event 73
5.5 Interrupt handling 74
5.6 Local wakeup 74
5.7 Remote wakeup 74
5.8 LPS code group detection 75
5.9 Timeout on $t_{to(req)sleep}$ 75
5.10 Acknowledge timer behaviour 76
5.11 Transition to NORMAL with no $t_{to(ack)sleep}$ timeout 76
5.12 State tracing using waveform viewer 76
5.13 TCP packet sniffing using wireshark 79
5.14 UDP packet sniffing using wireshark 81
5.15 Code coverage representation 81
5.16 Virtualizer analysis settings 85
5.17 Pin and SMI registers tracing 85
5.18 Pins and sockets interface 86

Acronyms

DT

Digital Twin.

ESD

Electric Static Discharge.

CMC

Common Mode Choke.

LPF

Low Pass Filter.

EMC

ElectroMagnetic Compatibility.

OSI

Open System Interconnection.

SMI

Serial Management Interface.

MII

Media Independent Interface.

RMII

Reversed Media Independent Interface.

MDI

Media Dependent Interface.

PCS

Physical Coding Sublayer.

PMA

Physical Medium Attacher.

MAC

Media Access Control.

UV

Under Voltage.

DUT

Device Under Test.

VDK

Virtual Development Kit.

CAN

Controlled Area Network.

LIN

Local Interconnect Network.

LAN

Local Area Network.

TCP

Transmission Control Protocol.

UDP

User Datagram Protocol.

IP

Internet Protocol.

PoE

Power over Ethernet.

UTP

Unshielded Twisted Pair.

STP

Shielded Twisted Pair.

RF

Radio Frequency.

EMI

ElectroMagnetic Interference.

TLM

Transaction Level Modeling.

SoC

System on Chip.

PEP

Punch Electronic Platform.

ASIC

Application Specific Integrated Circuit.

LPS

Low Power Sleep.

WUR

WakeUp Request.

WUP

WakeUp Pulse.

FSM

Finite State Machine.

ECU

Engine Control Unit.

AUTOSAR

AUTomotive Open System ARchitecture.

PLL

Phase Locked Loop.

ADAS

Advanced Driver Assistance Systems.

OSCI

Open SystemC Initiative.

API

Application Programming Interface.

DC

Direct Current.

AC

Alternate Current.

PFI

Ported Fuel Injection.

SENT

Single Edge Nibble Transmission.

SCML

SystemC Model Library.

FPGA

Field Programmable Gate Array.

CHAPTER 1

Digital Twin and SystemC modeling

The aim of this chapter is to give a general overview about Digital Twin (DT) technology to explore the concept, development, and far-reaching implications of DTs across various domains. In the following sections, a deep analysis will be performed into the core principles, technologies, and real-world applications that have led to the emergence of digital twins. After that, it will be discovered how DTs are creating exciting opportunities for precision, optimization, and innovation. In addition to that, challenges and considerations that come with putting them into practice will be explored. Finally, it will be discussed about SystemC Hardware Co-Design, why it is going to be deeply and more often used in the industry and how it will be exploited for the purpose of modeling the TJA1101B Ethernet PHY in the virtual environment.

1.1 Digital Twin creation and behaviour

DT technology is a cutting-edge concept that involves creating a virtual replica or representation of a physical object, system, or process in the digital realm. This virtual counterpart, known as the "digital twin", is an exact digital simulation that mirrors the real-world entity's characteristics, behavior, and functionality. It is dynamically updated in real-time, allowing it to reflect changes and updates in the physical counterpart accurately. The core idea behind DT technology is to create a bridge between the physical and digital worlds. By having a DT, engineers, designers, and operators can gain valuable insights into the performance, status, and interactions of the physical entity, such as the decision making at different levels, from the early production strategy of the product to the more technical operations.
This new level of abstraction has led to a re-creation of the industry product developing process, which is now more focused on the digital world and based on:

- Virtual environment
- Physical space (the real world)

- Communication and exchange of data between the two

1.1.1 Examples of Digital Twin applications

Due to the capability to provide real-time data analysis of the involved environment, DT has become a keystone for production, manteinance, optimization and performance analysis of the product. Some use-cases are here reported[21]:

- Manufactoring: industries can create a virtual model of the manufactoring process such that it is possible to constantly monitor the correctness of the production and identifying errors in a faster way
- Automotive: electronic peripherals and mechanical parts can be abstracted to observe their physical interaction
- Construction: construction companies can test different conditions and worst cases to see how a building will response to a given stimuli (earthquake, flood, and so on), or how the building asset will be in real time
- Energy: energy companies can create the virtual model of their energy assets to test in real-time the quality of thei energy systems, increasing time-to-market and safety of their products
- Healthcare: patient's body can be modelled to better design medical devices and equipments, together with diagnosis accuracy to reduce unnecessary procedures
- Transportation: models of city transportation systems are now built virtually to test their behaviour in different scenarios, such that engineers can optimize their systems before they are implemented
- Medicine: DT technology in the medicine field can be used for educational purposes such that students can interact with a virtual replica of the human body and can explore it without any risk on a real patient; in addition to that, scientists can study more accurately new hypotheses and speed up the development of new medicines
- Smart Cities: a city replica can be used to simulate the traffic flow for identifying bottlenecks and improve the traffic timing, or to analyze energy consumption and optimize its efficiency

1.1.2 Creation process of the Digital Twin

The creation of the virtual model of a product is a long process, and it must be accurately designed in order to take advantage of its features. It starts from deciding

how complex the model should be: the more the model is rich of details the more accurate will be the simulation of the replica, but this results in the software overhead due to the interal implementation; on the other hand, the model can be a simplistic version of the physical produt to enable faster simulations, but it will lose a huge amount of the technology involved. The tradeoff is left to the implementers according to their needs and specifications.
A possible approach is identified by Deloitte University[1], and it is composed by 6 steps:

- Imagine
- Identify
- Pilot
- Industrialize
- Scale
- Monitor

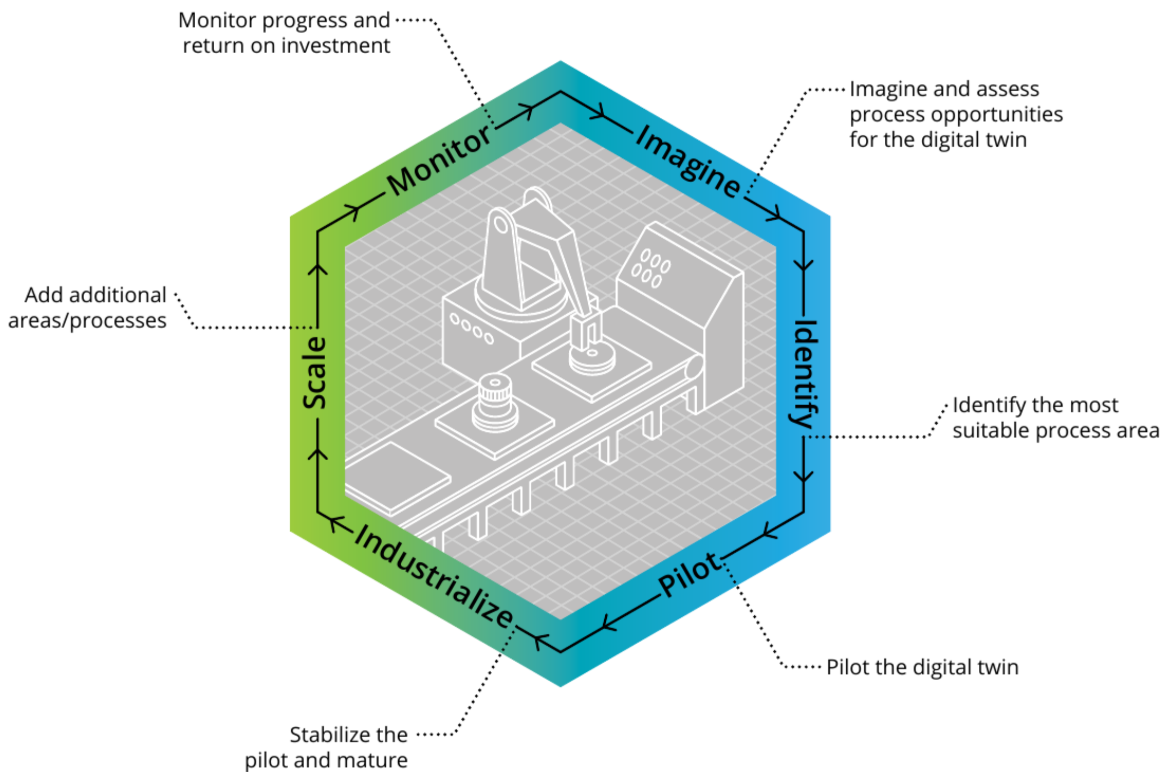


Figure 1.1: Digital Twin creation

Fig.1.1 shows the steps listed above.

[1]Source [31], pp12

- Imagine: first step tries to identify a set of scenarios in which the model can be implemented to bring the DT value to the company
- Identify: identify the DT that offers success in its implementation and benefits to the industry production
- Pilot: it can be considered as the representation of an use-case by starting from the data analysis provided by different company divisions
- Industrialize: start the industrial production of the product once the virtual model is available
- Scale: adjust the twin for different processes interconnected with the pilot
- Monitor: constantly monitor the behaviour of the solution, in order to allow continous improvement of the twin

1.1.3 Data acquisition of the Digital Twin

As said, one of the key aspect for DT development is data, which must be collected in near real-time to ensure accuracy, efficiency and adaptability of the virtual model. In this section, it will be analyzed how DT data are collected using different methods[32]. However, it is important to highlight different requirements on the data collection:

- **Comprehensive data**: comprehensive data gathering refers to the process of collecting and compiling a wide range of relevant information from various sources to create a comprehensive and complete dataset. This means that the virtual model should collect data both from the physical world and virtual world: it must consider certain scenarios and uncertain scenarios, because low-probability events should be included in the dataset, otherwise the model will not behave correctly when responding to real world stimuli.
- **Knowledge mining**: knowledge mining is the process of extracting valuable insights and knowledge from large volumes of data. In order to derive a real entity behaviour, it is necessary to extract information by a raw data collection and build a useful model: the challenge remains understanding which data is useful for extracting information over a very large dataset, which data is unuseful and which data is redundant.
- **Seamless data fusion**: seamless data fusion is the process of combining information from multiple sources or sensors to create a unified and coherent representation of the data. It is required when dealing with different data sources in order to build an accurate model: different data must be integrated such that they can be verified, corrected and validated by each other.

- **Real-time data interaction**: real-time data interaction is useful for:
  - updating virtual model parameters from the physical world
  - real-time diagnosis and mantainance of the physical entity
  - illustrating deficiencies and calibrating the virtual model
- **Iterative optimization**: it is a cyclic process through which new incoming data is compared with old data to generate new information until the best possible solution is found.
- **Data universality**: a high universality of data is required to solve problems related to transferring a DT across different scenarios that have different constraints on data collection.
- **On-demand data usage**: a virtual model often has to provide different data types, so it is difficult to allow and implement a generic set of data operations.

To fulfill the requirements cited above, a set of seven principles has been developed:

- **Complementary principle**: it corresponds to comprehensive data gathering, where physical and virtual world feed each others; physical world reflects the dynamic effects of the real entity, and virtual world generates rare events which cannot be easily measured and collected by the physical world
- **Standardization principle**: it corresponds to data universality, such that data is collected under templated structures to be exchanged among different applications
- **Timeliness principle**: it corresponds to real-time data interaction, where data has to be manipulated and compressed to speed up the communication in a time-manner, and make it real-time
- **Association principle**: it corresponds on knowledge mining, and it associates different aspects of the DT to extract useful information
- **Fusion principle**: it corresponds to seamless data fusion, and it aims to fully merge data coming from different sources and minimize uncertainties related to the different parts of the entity (both digital and physical)
- **Information growth principle**: it corresponds to iterative optimization, where new data should be continously merged with old data through iterative fusions; if the new information is valid, the information quantity can be increased or decreased deciding wheter to accept the new information or to discard it
- **Servization principle**: it corresponds to on-demand data usage, and it is based on the encapsulation of data resources into on-demand data structures

Referring to Fig.1.2, it is possible to see how what kind of technology could be a suitable solution for every different method of data collection that is in conformity with the associated principle listed above:

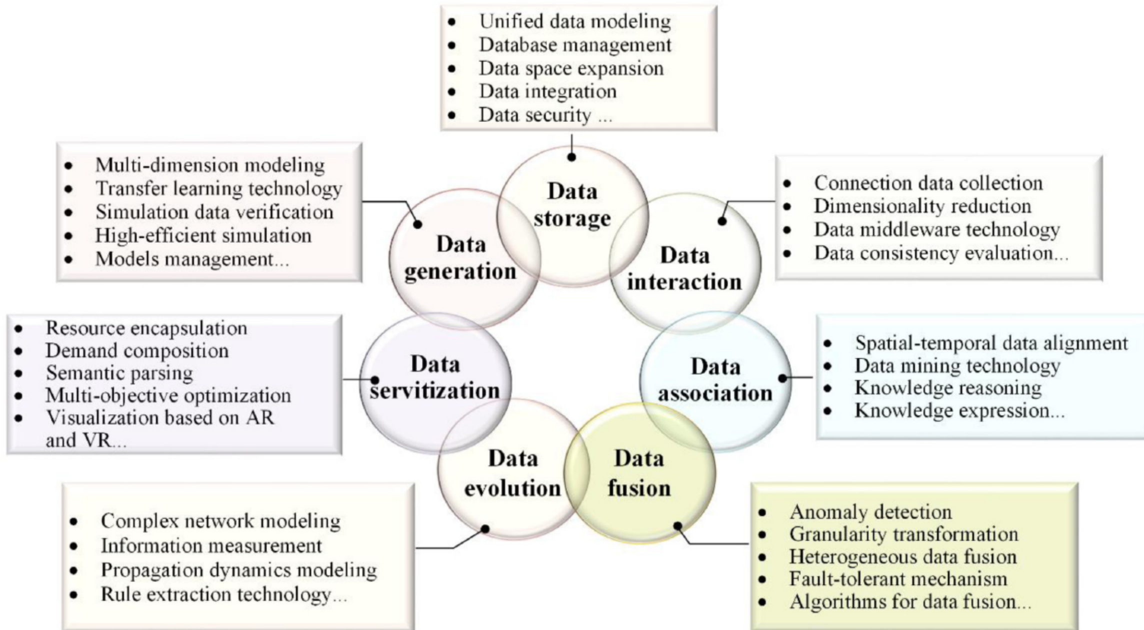


Figure 1.2: Enabling technology for data collection

1.2 SystemC language

SystemC is a versatile tool that enables engineers model complex electronic systems using high-level algorithms and constructs. It has been developed in the year 1999 by a team of ARM, CoWare, Synopsys and CynApps; after that, Open SystemC Initiative (OSCI) was created in year 2000 and it is responsible for continously develop and improve the language. In 2005, SystemC has become a IEEE standard and in 2011 it added the support for Transaction Level Modeling (TLM). It was born to address the growing complexity in hardware design and verification by means of a higher level of abstraction.
Inside this thesis context, a DT model is to be intended as a virtual ECU whose behaviour is emulated at instruction level: this means that microprocessor instructions are simulated one by one by means of the SystemC virtual environment. The virtualization process allows to abstract hardware interfaces and to develop the software to be flashed in the microprocessor before the hardware is available (enabling concurrent developing between hardware and software), and without changing it when the hardware product is then accessible. In addition to that, the virtual model enables the simulation of possible faults that may occur in the ECU and it can be easily integrated with other models developed using Simulink or OpenModelica.

1.2.1 Overview

SystemC is a C/C++ class library developed to model hardware architecture, System on Chip (SoC) interfaces and system-level design[29], and it is a cycle-based language model. The reasons behind the choice of using C and C++ language for SystemC library can be multiple: nowadays, the enormous hardware complexity has led to the developing of faster executable specifications for verification and validation of the involved system[2] and only C/C++ languages could provide fast code execution and adequate abstracion levels; in addition to that, the hardware design world needed for a standardized modeling language in order to provide interoperability design tools and services.
Similarly to VHDL hardware language, SystemC offers the possibility to design structural hierarchies using ports, signals and modules but also to model concurrent behaviour using processes and to model binary numbers and bits using specific data types. Everything will be discussed in the following sections.

1.2.2 Module hierarchy

Almost every large design can be splitted down in several submodules to manage complexity and readability of the overall code. Just as VHDL provides *entity* for modeling a hardware block, SystemC provides *sc_core::sc_module* class to encapsulate design components and all the required methods for modeling that specific block: the designer is able to create his own design hierarchy by instantiating submodules, ports, channels and processes for simulation; another significant advantage of creating a module hierarchy is that it is possibile to hide internal data and algorithms from other modules, forcing to use public interfaces that allow to abstract the module interal behaviour.
The module declaration must be included in the class header file (*.h* file extension), and it has to comprise ports declaration, methods declaration and members declaration; the implementation of the module (the *architecture* of VHDL) must be included in the *.cpp* file.
Modules can be declared using SystemC macro *SC_MODULE*, or using the class *sc_module_name* which holds the name for the current module; to construct the module, SystemC provides two ways of accomplish the task: it provides *SC_CTOR(module_name)* macro or the *SC_HAS_PROCESS(module_name)* macro, where the difference between the two is that using *SC_CTOR* macro, no additional parameters can be passed to the constructor, while using *SC_HAS_PROCESS* it is possible to pass further parameters to the custom module constructor. Here are reported generic examples of how to declare and construct a module:

```
1 #include "systemc.h"
2
```

[2]Source[29], pp1

```
3 SC_MODULE(module_name) {
4     /*declaration of public, private and protected members*/
5     SC_CTOR(module_name) {
6         /*module constructor implementation*/
7     }
8 }
```

```
1 #include "systemc.h"
2
3 class example_class : public sc_core::sc_module {
4     SC_HAS_PROCESS(example_class);
5     example_class(sc_module_name module_name /*additional parameters*/);
6 }
```

It is important to notice that class *sc_module_name* has its private member of type *const char **, so the module name can be passed by value between quotation marks in the class instantiation.

1.2.3 SystemC ports and signals

Module ports are module object instances that can be seen as hardware pins, and they are responsible for passing data from/to the module itself. They can be input ports, output ports or both input and output ports; a substantial difference with respect to VHDL pinout is that SystemC ports are templated objects: this means that they can be of any type (C++ built-in type, SystemC defined type or user defined type) such that it is possible to guarantee a higher level of abstraction and flexibility.
Here it is shown how to declare module ports:

```
1 #include "systemc.h"
2
3 SC_MODULE(module_name) {
4     sc_in<port_type>      _input_port;
5     sc_out<port_type>     _output_port;
6     sc_inout<port_type>  _input_output_port;
7
8     SC_CTOR(module_name) {
9         ...
10    }
11 }
```

Input ports can be read using *port_type sc_in::read()* method but they cannot be written; output port can be written using *void sc_out::write(port_type value)* method but they cannot be read; inout ports can be both read and written.
To be able to connect pins of different modules, just as VHDL and Verilog, it is necessary to use signals: they represent physical wires responsible for carrying data among modules. As SystemC ports, SystemC signals are templated objects but the direction of data flow is determined by the ports they are connected to. A very unique feature of

ports and signals is the possibility to trace their switching activity: using the SystemC function *sc_trace* it is possible to register the simulation time and value each time a port or signal changes in the simulation by writing these information into a *.vcd* file; when the simulation ends, *.vcd* files can be loaded into a waveform viewer program and the device internal values can be analyzed properly. Also SystemC events can be traced, and it will be discussed lately.
An example of signal usage is here reported:

```
#include "systemc.h"
#include "mod1.h"
#include "mod2.h"

SC_MODULE(Top) {
    sc_signal<signal_type> s;

    mod1 *module1 = new mod1("mod1_name");
    mod2 *module2 = new mod2("mod2_name");

    mod1.in_port(s);
    mod2.out_port(s);

    SC_CTOR(Top) {
        ...
    }

    int sc_main() {
        /*main implementation*/
    }
}
```

In this example, signal *s* is connected to output port of instance module2 and to input port of module1, and it will carry data coming out from module2 to module1.
Referring to the previous example, it is shown how to trace ports and signals:

```
#include "systemc.h"
#include "mod1.h"
#include "mod2.h"

sc_trace_file *trace_file = sc_create_vcd_trace_file("VCD_file");

SC_MODULE(Top) {
    sc_signal<signal_type> s;

    mod1 *module1 = new mod1("mod1_name");
    mod2 *module2 = new mod2("mod2_name");

    mod1.in_port(s);
    mod2.out_port(s);

```

```
16     SC_CTOR(Top) {
17         ...
18     }
19 }
20
21 int sc_main() {
22     Top t("Top");
23
24     sc_trace(trace_file, t.mod1.out_port, "port_name");
25     sc_trace(trace_file, t.mod2.in_port, "port_name");
26     sc_trace(trace_file, t.s, "signal_name");
27
28     sc_close_vcd_trace_file(trace_file);
29     return 0;
30 }
```

sc_create_vcd_trace_file allows the designer to open a file of type *sc_trace_file* (file extension *.vcd* will be assigned automatically) where to write the switching activity of desired ports, signals and events.

1.2.4 SystemC processes

Processes in SystemC are used by the library to model different parts of a specific system that execute concurrently and interact with each other. SystemC processes can be seen by the end-user as normal processes and threads in normal programming, but they are not actually part of an operating system call; instead, they are abstractions that represent concurrent activities in a hardware description. The concept of concurrency in SystemC is just an illusion given by the SystemC kernel[24]: it is true that it simulates concurrency, but when multiple processes are called to begin their execution at the same simulation time, only one process is executed at a particular time. The illusion of concurrency is given by the fact that the simulation time remains the same until every process has finished its execution. SystemC provides three process types: methods, threads, cthreads; their behaviour will be explained in the following subsections.

SystemC methods

Methods can be seen as a normal C/C++ function call that does not require any parameter and does not return any value. Its execution begins every time a value in its sensitivity list changes, and it does not cost any simulation time (i.e. it is executed in SC_ZERO_TIME). A function to be used as a method must be registered inside the module constructor, and its implementation must be provided separately. An example of a method registration is here provided:

```
1 #include "systemc.h"
2
```

```
SC_MODULE(my_module) {
    ...
    SC_CTOR(my_module) {
        SC_METHOD(module_method);
        sensitive << any_port << any_signal << any_event;
    }

    void module_method();
}
```

SystemC threads

Unlike methods, SystemC threads can be seen as software threads that begin their execution and never return in order to mantain the interaction with the external environment. To do so, a systemC thread typically contains an infinite loop and a *wait()* statement to suspend its execution to allow the kernel to schedule other processes and to advance the simulation time. Thread execution will continue when a signal in the sensitivity list has changed. Threads are a good solution when the designer has the need to rely on a higher level of abstraction without wasting hardware or software resources; however, a systemC thread is not a synthesizable construct and it is only used for simulation purposes. Just as methods, threads must be registered inside the module constructor:

```
#include "systemc.h"

SC_MODULE(my_module) {
    ...
    SC_CTOR(my_module) {
        SC_THREAD(module_thread);
        sensitive << any_port << any_signal << any_event;
    }

    void module_thread();
}
```

SystemC cthreads

Cthread stands for "clocked thread", and it is a subclass of systemC thread: cthreads are triggered on one edge of one single clock, and do not accept a sensitivity list; instead, they only accept a clock object and its triggering edge. Cthreads can be registered as follows:

```
#include "systemc.h"

SC_MODULE(my_module) {
    ...

```

```
6       sc_in_clk clock;
7
8       SC_CTOR(my_module) {
9           SC_CTHREAD(module_cthread, clock.pos()); //module_cthread is triggered on the rising edge of the clock object "clock"
10      }
11
12      void module_cthread();
13  }
```

Cthreads can be used when it is required a synthesis result, because their behaviour meets the requirements provided by synthesis tools.

1.2.5 SystemC events

Events are systemC objects that can cause the triggering of methods and threads: they can be customized by using the *sc_ event* class or using the default events provided by *sc_ port*, *sc_ signal* and others. Custom events using the *sc_ event* class can be notified with *notify()* method: the notification can be immediate (*event_ name.notify()*) or delayed after a specific amount of simulation time (*event_ name.notify(delay_ value)*); in addition to that, an event that has to be notified with a time delay can be cancelled using the *cancel()* method. This can be useful when modeling timers with a reset signal.

1.2.6 SystemC time modeling

In this section, it will be discussed how the simulation time can be handled and the classes that can be used to model it.

sc_time

This is s systemC class provided to handle objects that model the simulation time: they can be used to notify events after the specified time, to suspend the simulation time for a specific value, and so on. It is composed by an double number and the time unit, which are:

- SC_FS : femtoseconds
- SC_PS : picoseconds
- SC_NS : nanoseconds
- SC_US : microseconds
- SC_MS : milliseconds
- SC_SEC : seconds

sc_time_stamp

sc_time_stamp() is a useful method that returns the current simulation time (which is in turn a *sc_time* object), and it does not accept any parameter: it can be useful for debugging purposes, since an *sc_time* object can be displayed on a terminal or other terminal emulators.

sc_start

This method is responsible to start the simulation: when it is called, all processes will start at the same simulation time even if there has not been any triggering specified in the sensitivity lists. It can be called without any parameter, or it is possible to pass a *sc_time* object to specify the maximum time the simulation can run:

```
sc_start(); //simulation runs until every process has terminated
sc_start(const sc_time& time_object); //simulation runs for the specified time
sc_start(double max_time, sc_time_unit time_unit); //simulation runs for the specified time
```

1.2.7 SystemC simulation scheduler

According to the language reference manual provided by IEEE[15], the scheduler main purpose is to start or resume one of the user processes. The kernel scheduler is event-driven, this means that it schedules one of the runnable processes when an event occurs in a specific simulation instance. IEEE1666 standard language model defines five situations in which a process can be executed by the scheduler[3]:

- after the initialization phase (if it is made runnable)
- when *sc_spawn()* function is called during the simulation
- when an event occurs, if present in the sensitivity list
- when a timeout occurs
- when a call to a process control member function occurs

The scheduling algorithm is composed by several phases, and it is described in the following subsections.

[3] Source[15], pp19

Initialization phase

The first step executed by the scheduler is the so called *initialization phase*, which performs three consecutive steps[4]:

- execute the update phase without delta notification
- fill the set of runnable processes (methods and threads) of an object instance, excluding clocked threads and methods for which *dont_initialize()* function has been registered
- run delta-notification phase and start evaluation phase

Evaluation phase

The evaluation phase is responsible for executing a user process[5]: it selects one of the process instances from the runnable process set and it removes the process from the set; after that, the process is executed without interruption until it returns or it calls the *wait()* function.

Update phase

After the evaluation phase, every pending calls to function *update* is executed such that every port and signal values is updated; when there is no pending call left, the scheduler moves to the delta-notification phase[6].

Delta-notification phase

During the delta-notification phase, the scheduler determines which process is sensitive to event notifications or timeouts and add it to the set of runnable processes; finally, it removes that notifications and timeouts from the set of delta-notification[7].

Timed-notification phase

In addition to delta-notification phase, SystemC offers a timed-notification phase in which, if pending timed notifications or time-outs exist, the simulation time advances to the specified notification or timeout, add notification sensitive processes to the runnable processes set and remove the previous notifications[8].

[4] Source[15], pp22
[5] Source[15], pp23
[6] Source[15], pp25
[7] Source[15], pp25
[8] Source[15], pp25-26

1.2.8 Transaction Level Modeling

TLM provides an abstraction layer that captures essential communication and interaction details while minimizing the intricacies of individual components, such as pin events and exact time representation[6].

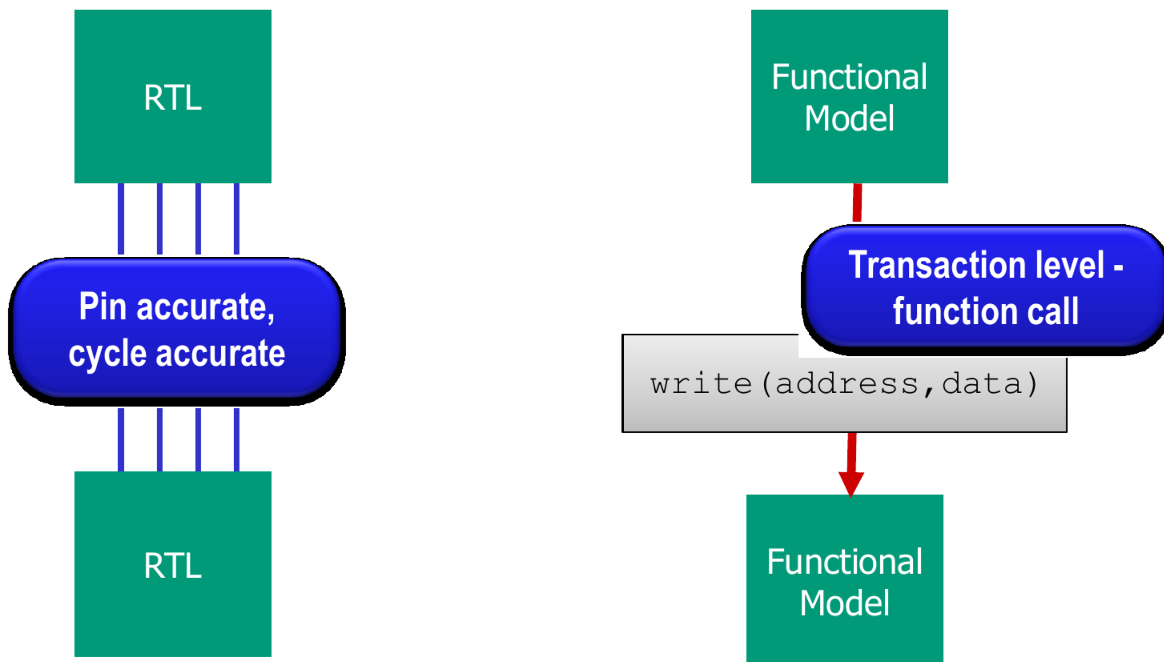


Figure 1.3: RTL description and TLM differences

TLM allows to represent only the key aspects of the internal architecture[9] such that the designer can use the TLM model as golden reference for hardware verification and use software execution to obtain faster simulations that RTL simulations.

Initiator and target sockets

To interconnect different components inside the design and to allow high level communiation among them, TLM uses two socket types to representation interconnections:

- Initiator socket: it is responsible for initiating a transaction (data exchange)
- Target socket: it is responsible for receiving a transaction, and to implement the behaviour of the transaction

[9]Source[6], pp7

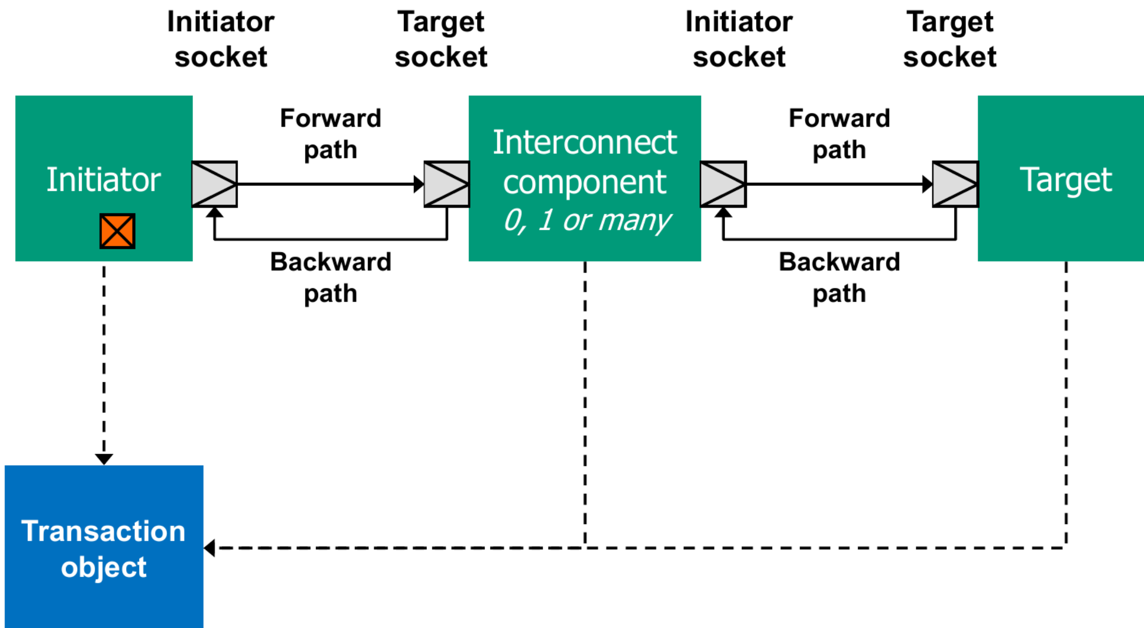


Figure 1.4: TLM socket representation

TLM transports

To allow the communication among initiatior and target sockets, TLM offers two types of transport calls: blocking and non-blocking transports. The main difference between the two is that non-blocking transport keeps track of the transaction phases, meaning that a non-blocking transaction can execute in multiple phases[10], while a blocking one executes in one delta cycle. From an initiator socket point of view, the call for a transaction involves only the call to *b_transport* or *nb_transport*, while from a target socket point of view the functions related to a transaction must be registered using the *register_b_transport* or *register_nb_transport* and they have to be implemented separately.

Generic payload

The content of a TLM transaction is represented by a payload object: in particular, the generic payload allows the abstraction of memory busses and the full interoperability among the involved sockets[11]. Generic payload options can be set by the end-user using payload methods provided by TLM library: command, address, data, byte enables, Direct Memory Interface (DMI), streaming width, data width, response status are set by the initiator socket and get by the target socket.

---

[10]Source[6], pp23
[11]Source[6], pp63

1.2.9 SCML 2.8.0 library

The objective of this thesis project is to replicate the behaviour of the TJA1101B device using the SystemC simulation framework. Subsequently, the source code of the model will be under the ownership of Punch Torino company, which will build the virtual environment on the VDK. VDK tool is provided by Synopsys group, and it allows comprehensive debugging and model analysis. In order to trace and debug memory contents to be displayed in the Virtualizer environment, Synopsys provides an open-source SystemC compliant library known as SystemC Model Library (SCML)[13]: in this project, it will be used the 2.8.0 library version. SCML memory objects have been created to hide TLM APIs details and to create a simple mechanism to describe a memory map of a device. Allowed memory objects are:

- Memory: top-level object that acts as a data storage
- Memory alias: represents sub-regions inside the memory map and can be associated with different behaviours
- Register: represents the *memory* word
- Bitfield: represents *register* single or consecutive bits

SCML memory objects are compliant with TLM transport calls for communication among different components: in this sense, a memory object has blocking and non-blocking transports and their behaviour can be hidden to the end-user or it can be customized by the designer using the SCML memory callbacks. Memory callbacks represent the behaviour associated to a *memory*, *register* or *bitfield* access[12]: many types of memory callbacks are provided by the library but, for sake of semplicity, only some relevant regular callback registration are here reported[13]:

- *set_callback*: it registers a function that will be called whenever a read or write access takes place
- *set_read_callback*: it registers a function that will be called when a read access takes place; the values of the memory contents will be restored after the callback returns
- *set_read_no_store_callback*: it registers a function that will be called when a read access takes place; the values of the memory contents will not be restored after the callback returns (a memory address could be modified)
- *set_write_callback*: it registers a function that will be called when a write access takes place

[12]Source[13], pp25
[13]Source[13], pp26

- *set_write_ignore_restriction*: it registers a memory object to ignore a write access (useful to model read-only memories)

When initializing a memory object, it takes an argument which represents the memory object name that will be displayed, as said, in the Virtualizer environment.

1.3 Comparison between Digital Twin and SystemC model

Analyzing carefully the characteristics of a DT model and of a SystemC model, it is possible to highlight several differences: first and foremost, a DT is a virtual representation of the physical object to be monitored and analyzed, while a SystemC model simulates a digital hardware circuit at various abstracion levels; furthermore, a DT focuses on replicating the behavior and attributes of a specific physical entity, often including real-time data and other sources of information to provide accurate and up-to-date insights into the behavior of the physical entity, while the SystemC model primarily relies on predetermined models and parameters to model the behavior of digital hardware and software systems. In this sense it is not correct to say that the SystemC virtual model of the TJA1101B device will be the DT of the real device, because it can be seen as a black-box that, given certain inputs, it will produce proper outputs but there are no information about the physical hardware and there are no real-time data sources from sensors for updating the model itself. Even if DT and SystemC environments are two different worlds, the idea of combining SystemC and DT is emerging to create a bigger and more accurate ecosystem[7]: in fact, the current form of the DT lacks the capability to integrate an embedded system. To achieve the combination of these two, it would be required a *top-down* approach[14] for developing feasible standards and distributed simulation/co-simulation APIs in order to embed SystemC in the overall environment; another key feature that has to be inserted in the SystemC library is the capability of the model encapsulation, because up to now, a systemC model can be integrated in a non-real time environment only.

[14] Source[7], pp25

CHAPTER 2

General overview of Automotive Ethernet

In this chapter it will be described the context of the Ethernet protocol in an automotive environment, highlighting the main differences with respect to Controlled Area Network (CAN) and Local Interconnect Network (LIN), which are the currently most used protocols in the automotive industry.

2.1 Limitations of CAN and LIN protocols

While CAN and LIN have served as the backbone of automotive communication systems for many years, their inherent limitations have become increasingly apparent as the automotive industry pushes the boundaries of innovation.
CAN, although reliable and widely adopted, offers limited bandwidth and struggles to keep up with the vast amounts of data generated by the growing number of sensors, cameras, and other components in modern vehicles. Additionally, CAN rigid data frame structure and fixed message length hinder its ability to efficiently handle large volumes of data required for emerging applications such as high-resolution cameras, sensor fusion, and over-the-air updates.
On the other hand, while LIN excels in cost-effectiveness, simplicity, and low-power applications, its low-speed nature and limited bandwidth make it unsuitable for high-performance and data-intensive functionalities required in advanced automotive systems. LIN is primarily designed for basic sensor inputs, actuator control, and simple peripheral device communication.

2.2 Ethernet: a new suitable solution

Recognizing the need for a more robust and versatile communication protocol, the automotive industry has turned to Ethernet, the same technology that powers the internet and Local Area Network (LAN) worldwide. Ethernet offers several key advantages that make it an attractive alternative to CAN and LIN in automotive applications, such as:

- Bandwidth: Ethernet supports data rates of up to 10 gigabits per second (10 Gbps) and beyond, far surpassing the capabilities of CAN and LIN
- Standardization and interoperability: ethernet benefits of robust and well tested standards (such as IEEE 802.3 standard), and also of extensive research, development, and innovation in the field of networking; this leads to continuous improvements in terms of performance, security, and reliability, which are essential for critical automotive applications
- Versatility and integration: ethernet offers a full suite of networking protocols, such as Internet Protocol (IP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP), enabling seamless integration with existing network infrastructure. This opens the door to connectivity with external systems, such as cloud services, fleet management platforms, and intelligent transportation systems, facilitating advanced data analytics, remote diagnostics, and software updates. In addition to this, Ethernet us able to carry Power over Ethernet (PoE): this simplifies wiring and reduces the complexity of the vehicle's electrical architecture. PoE enables the consolidation of power and data cables, reducing weight, cost, and installation complexity, while also supporting the growing demand for electric vehicles and their charging infrastructure

2.3 From standard Ethernet to Automotive Ethernet

This section aims to describe the standard ethernet protocol, its main features and why it cannot be used in automotive environment as it is, but it requires relevant changes to be a suitable solution for a vehicle integration.

2.3.1 Standard Ethernet

Standard ethernet relies on the Open System Interconnection (OSI) model[20].
OSI model is an ethernet standard validated in 1984 by International Organization for Standardization (ISO) to produce standardized laws for computer networks. It is mainly composed by a stack of layers: each layer represent a sub-protocol that reduces the implementation complexity for a network communication system. There are 7 total layers, from physical layer (i.e. the cable, optic fiber or any other transmission medium) to application layer (high level communuication) as shown in Fig.2.1.

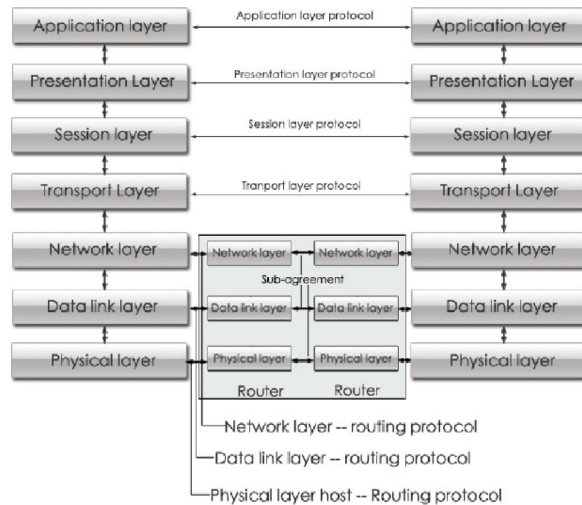


Figure 2.1: OSI model layers representation

According to IEEE 802.3 standard, the ethernet frame has the structure here reported:

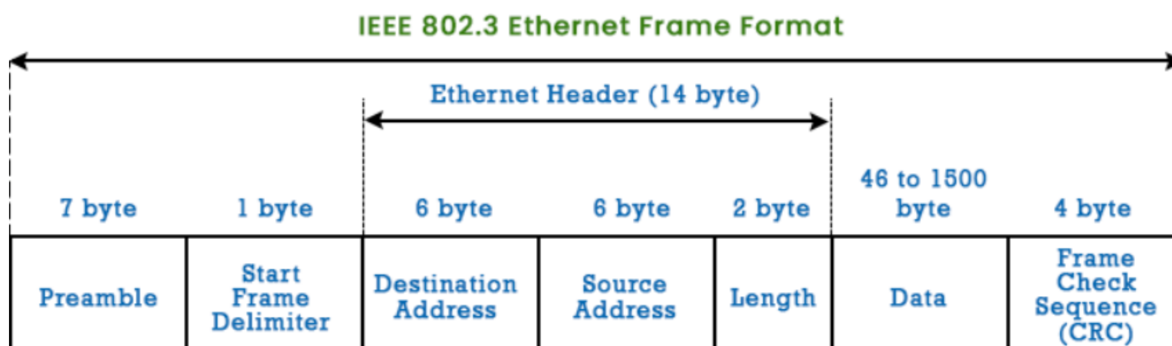


Figure 2.2: Data-link frame format

Fig.2.2 shows thee different fields present in the data-link frame structure:

- **Preamble**: 7 bytes to synchronize source and destination nodes
- **Start** of Frame Delimiter: 10101011 byte to identify the start of the frame
- **Destination** address: identifies the address of the recipient, it is composed by 3 bytes identifying the vendor ID and 3 bytes identifying the host ID
- **Source address**: identifies the address of the sender, and it is composed as the destination address
- **Length/Type**: 2 bytes field: if its value is below 1500, it identifies the length of the data payload; on the other hand, if it is greater than 1536 it identifies the network protocol used in the communication

- **Data** payload: from 0 to 1500 bytes of the message to be sent
- **Pad**: bunch of bytes (usually zeroes) to guarantee the minimum frame length
- **Frame Check Sequence**: contains Cyclic Redundancy Check (CRC) to verify if there have been errors in the frame during the transmission of the message

As said, many layers can be inserted in the ethernet frame to reduce the complexity of the communication.

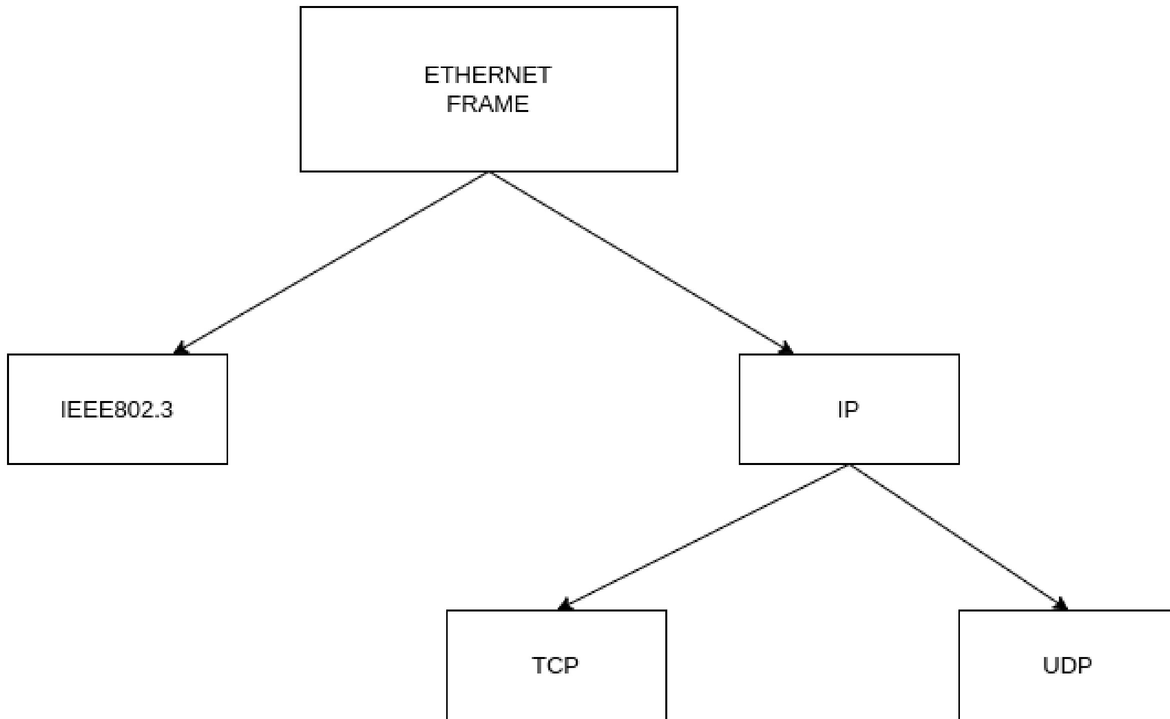


Figure 2.3: Ethernet types relationship

In Fig.2.3 it is highlighted the relationship among the supported ethernet types. The frame structure for each type is instead reported in the following pictures:

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset												
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
⋮	⋮																																
56	448																																

Figure 2.4: IP frame format

Fields of IP header:

- **Version**: IP version of the current frame (4 for IPv4 or 6 for IPv6)
- **Internet header length (IHL)**: header length of the IP header (expressed in number of 32-bit words)
- **Explicit Congestion Notification (ECN)**: notifies the network congestion without losing packets
- **Total length**: packet total length (expressed in bytes)
- **Identification (ID)**: primarily used for uniquely identifying the group of fragments of a single IP datagram
- **Flags**: 3 bits that indicate:
  - bit0: reserved (must be 0)
  - bit1: don't fragment (DF)
  - bit2: more fragments (MF)
- **Fragmented offset**: offset of a particular fragment relative to the beginning of the original unfragmented IP datagram
- **Time to Live (TTL)**: seconds for specifying the time of the packet life to prevent network failure
- **Protocol**: IP protocol for transmission layer
- **Header checksum**: value for error-checking of the header
- **Source address**: IPvX address of the sender
- **Destionation address**: IPvX address of the receiver
- **Options**: additional information for routers

TCP segment header

Offsets		0										1										2										3	
Octet	Bit	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0 0				C W R	E C E	U R G	A C K	P S H	R S T	S Y N	F I N	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20	160	Options (if *data offset* > 5. Padded at the end with "0" bits if necessary.)																															
⋮	⋮																																
56	448																																

Figure 2.5: TCP frame format

Fields of TCP header:

- **Source port**: port number associated to the source host
- **Destination port**: port number associated to the destination host
- **Sequence number**: number of the TCP segment beginning inside the entire communication flow
- **Acknowledgment number**: notifies the correct reception of the frame by the receiver
- **Data offset**: TCP header length (expressed in 32-bit words)
- **Reserved**: not used, must be set to 0
- **Flags**: 8 bits of control flags:
  - URG (1 bit): indicates that the Urgent pointer field is significant
  - ACK (1 bit): indicates that the Acknowledgment field is significant
  - PSH (1 bit): push function. Asks to push the buffered data to the receiving application
  - RST (1 bit): reset the connection
  - SYN (1 bit): synchronize sequence numbers. Only the first packet sent from each end should have this flag set
  - FIN (1 bit): last packet from sender
- **Window size**: specifies the amount of data that a sender can send before receiving an acknowledgment from the receiver

- **Checksum**: value for error-checking of the header
- **Urgent pointer**: indicates the last urgent data byte
- **Options**: payload options

UDP datagram header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Length																Checksum															

Figure 2.6: UDP frame format

Fields of UDP header:

- **Source port**: port number associated to the source host
- **Destination port**: port number associated to the destination host
- **Length**: length of header and data (expressed in bytes)

2.3.2 Automotive Ethernet

Standard ethernet is a standard that is currently dominating the computer network communications, because it can offer very high speed in terms of baud rate, but it cannot guarantee high performances in terms of latency, jitter time, message integrity and bandwidth: it can be suitable for non-real time applications, such as web applications where the user can reasonably wait a certain amount of second for a website load in the browser, but it can't be used for a context where a crashing detection must be recognized in few microseconds to activate the airbag of the vehicle. Furthermore, conventional Ethernet exhibits excessive noise and interference, rendering it unsuitable for automotive applications.
For these reasons, automotive industry has taken some features from the standard ethernet (such as the protocol structures) and adapted them for its needs.

Physical layer changes

Starting from the physical layer, standard ethernet uses coaxial cables, Twisted Pair cables and fiber optic cables: in the automotive industry, it is used a Unshielded Twisted Pair (UTP) cable for up to 1000BASE-T1 communication to reduce cost and weight[19], but for higher speeds it is used a single Shielded Twisted Pair (STP) for better Radio Frequency (RF)/ElectroMagnetic Interference (EMI) protection and higher data transmission speeds.

Ethernet speeds

Regarding the baud rates, standard LAN ethernet can use the following speeds:

- 10BASE-T
- 100BASE-T
- 1000BASE-T
- 10GBASE-T
- 40GBASE-T
- 100GBASE-T

In the automotive field, only 100BASE-T and 1000BASE-T are used because they can guarantee:

- Compatibility: these two baud rates are a widely adopted standard
- Cost-effectiveness: higher speeds would require more sophisticated hardware and cabling, which can increase costs
- Scalability: higher speeds are not necessary to increase performances
- Signal integrity: higher speeds would be highly affected by electromagnetic interference, temperature variations and mechanical vibrations

2.4 100BASE-T1 technology

In this section, it is described how the 100BASE-T1 technology is implemented in an automotive environment and how the conditioning circuitry is composed.

2.4.1 Cabling and wiring

For LAN networks, the most used ethernet standard is 100BASE-TX and it uses 2 pairs of UTP cable (1 for transmission and 1 for receiving); on the other hand, 100BASE-T1 meets all the requirements just as 100BASE-TX but it uses just 1 pair of UTP cable: this leads to significant improvements for reducing weight (30%) and costs (80%) according to OPEN Alliance and Broadcom[1].
The difference between the two standars is reported in Fig.2.7 and Fig.2.8.

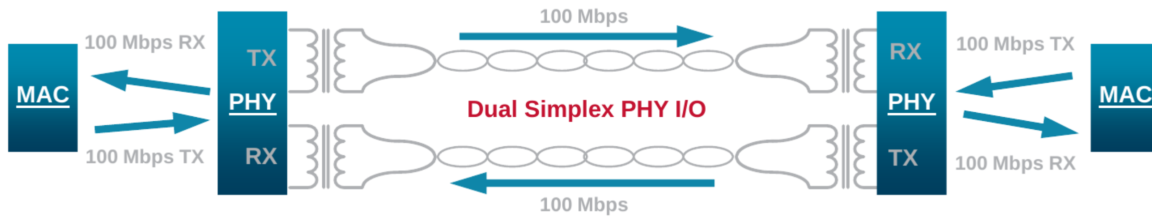


Figure 2.7: 100BASE-TX connection

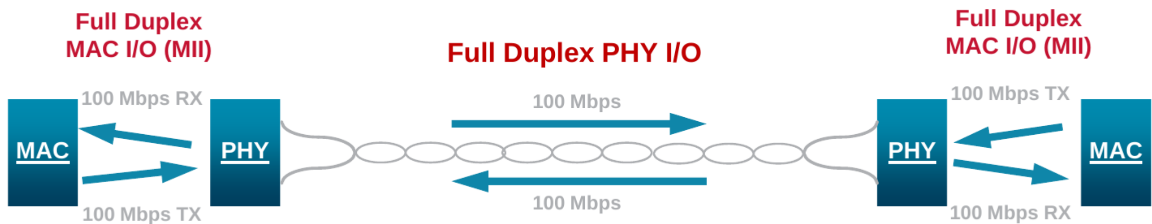


Figure 2.8: 100BASE-T1 connection

Using 100BASE-T1 standard and an UTP cable the maximum cable length is around 15m

2.4.2 Noise reduction

As a 100BASE-T1 compliant Ethernet PHY, the TJA1101B provides 100 Mbit/s transmit and receive capability over a single UTP cable[22], supporting a cable length of at least 15 m with a bit error rate less than or equal to 1×10^{-10}: to be compliant with this specification and with automotive ElectroMagnetic Compatibility (EMC) requirement, a Common Mode Choke (CMC) is typically inserted into the signal path.

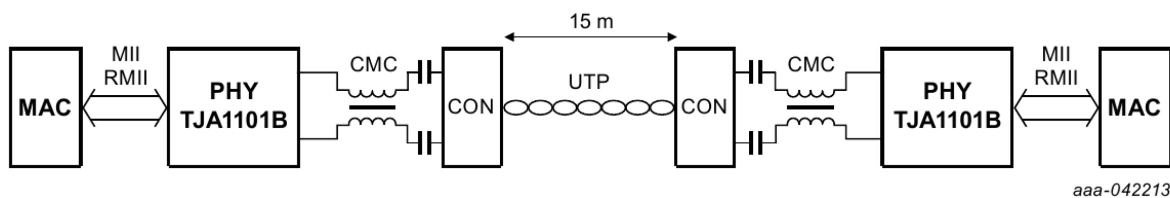


Figure 2.9: TJA1101B PHY application circuit for noise reduction

The PHY basically exchanges messages with a Media Access Control (MAC) using its MII, and over the UTP cable usign the MDI connected to the CMC.
To build a specific interface circuit, OPEN Alliance defines a generic interface circuit diagram for Electric Static Discharge (ESD) protection and filtering besides the CMC[1]:

[1]Source [8], pp11

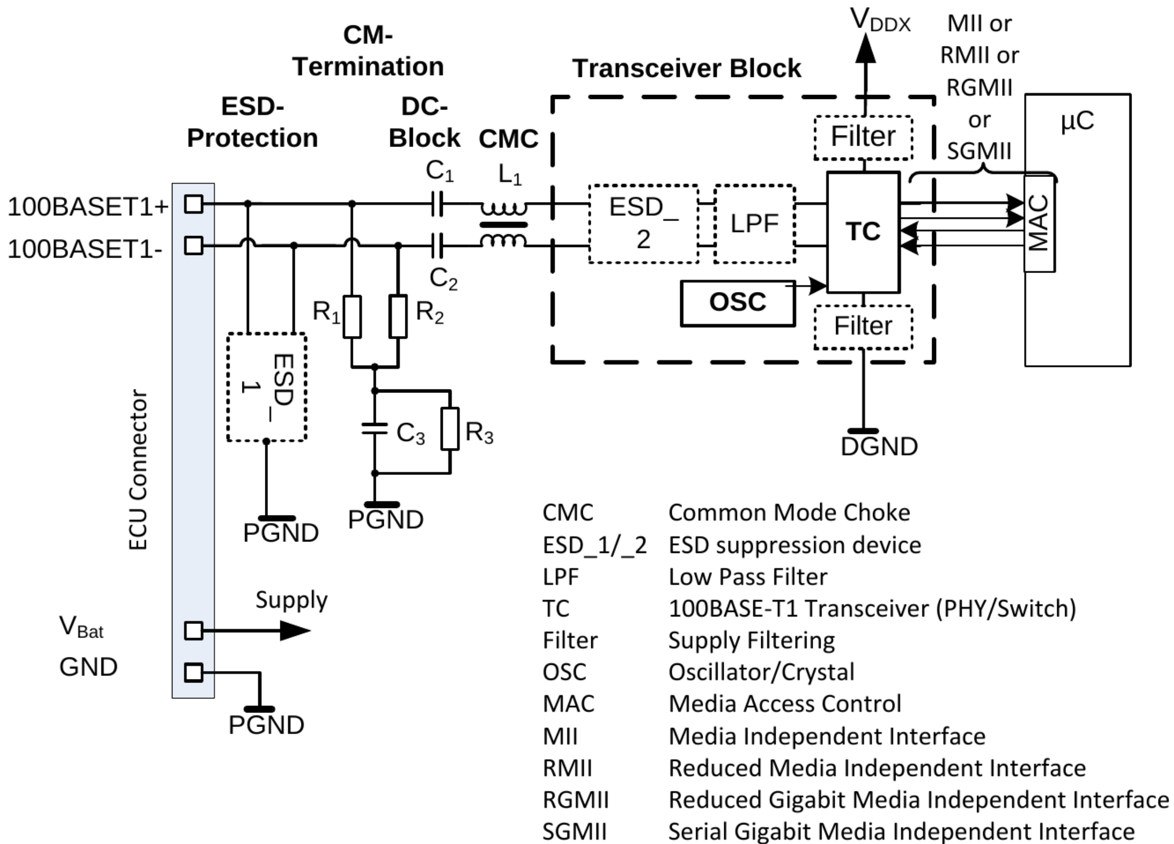


Figure 2.10: Block diagram of interface circuitry 100BASE-T1

Fig.2.15 shows the CMC, DC block capacitors C_1 and C_2, the ESD protection circuit and a Low Pass Filter (LPF) composed by R_1, R_2, R_3, C_3 to avoid static charge of cable harness.
All the required components must be selected properly according to the datasheet of the device provided by the manufacturer.

2.4.3 Echo reduction

When dealing with full-duplex mode (such as ethernet PHYs), it may occur that a portion of the transmitted signal is reflected back and received again at the source, creating an unwanted duplicate of the original signal: this phenomena is known as echo.
100BASE-T1 PHYs come out with intergrate hybrids circuits to eliminate the echo that can modify the signal transmission in a noisy environment[27]. A single PHY is composed by a transmitter, a receiver and a hardware block that acts as echo canceler; what the echo canceler does is, qualitatively, removing the transmitted signal at the receive block by detecting its transfer function and subtracting it to the received signal transfer function, such that the link partner information is extracted correctly.

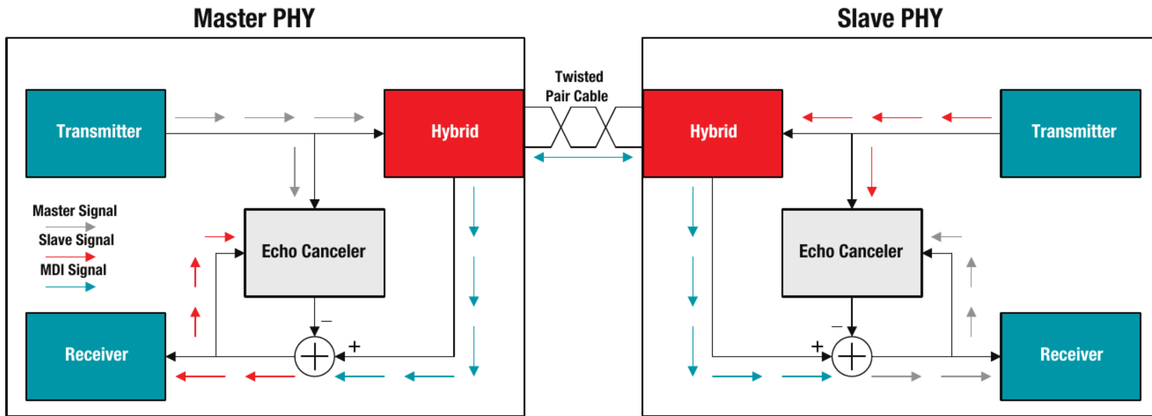


Figure 2.11: Echo cancellation block diagram

2.4.4 Bit encoding for EMI constraints

100BASE-T1 PHY can operate with different encoding maps for transmitting messages from an external MCU or MAC over the UTP cable to reduce signal degradation: this requires an appropriate bit encoding among the links MCU-PHY and PHY-UTP[27]. The following standards are:

- 4 bit to 3 bit (4B3B)
- 3 bit to 2 ternary pair (3B2T)
- Three level pulse modulation (PAM3)

The MAC can send a frame to the PHY: if the bit number of the frame is not divisible by 3, the PHY adds dummy bits to make it divisible by 3; then the PHY uses 4B3B conversion and 3B2T conversion to scale the message down to a ternary pair vector of values (-1, 0, 1) and modulates it using PAM3.

3-bit data	TA	TB
0 0 0	-1	0
0 0 1	0	1
0 1 0	-1	1
0 1 1	0	1
1 0 0	1	0
1 0 1	0	-1
1 1 0	1	-1
1 1 1	0	-1

Figure 2.12: Generation of the 2T ternary pair vector

In Fig.2.12 it is shown the bit mapping of 3 bits group to the ternary pairs to modulate using PAM3.

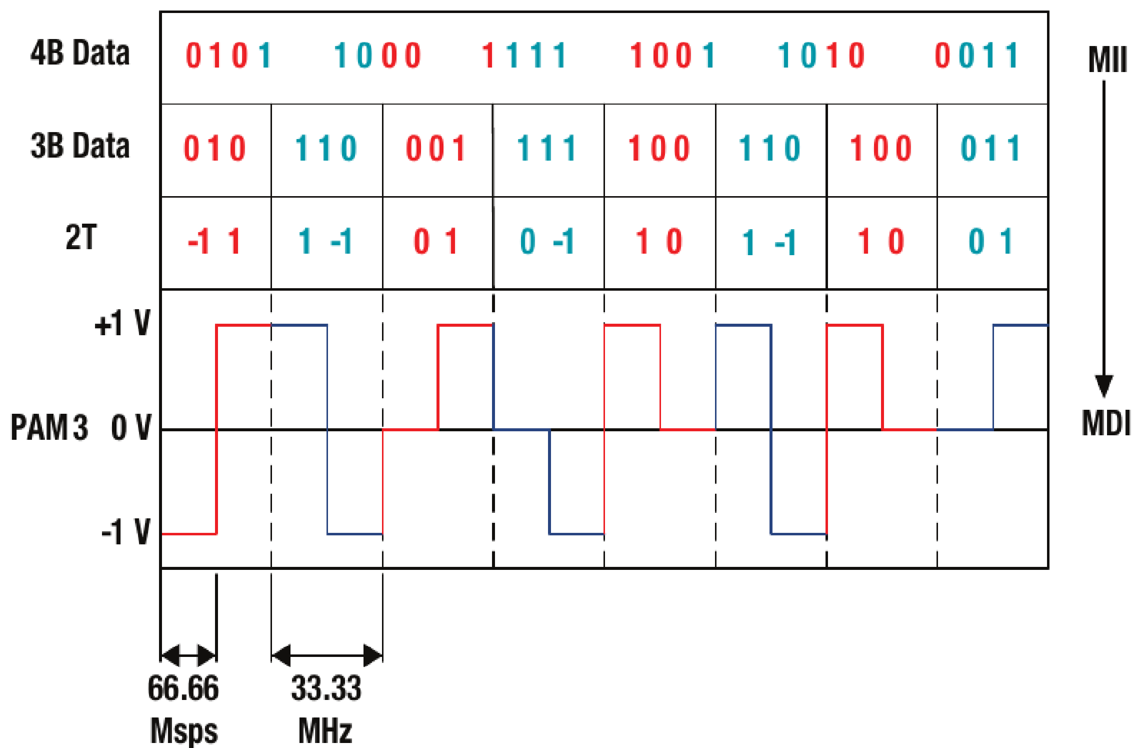


Figure 2.13: From MII to MDI conversion process

In Fig.2.13 it is illustrated the process that maps the MAC message from 4 bits

groups down to 2 bits to be sent on the UTP cable.
Starting from a base frequency of 25MHz (standard frequency), when the PHY uses 4B3B conversion it must increase the frequency up to $25MHz \times 4 \div 3 = 33.3MHz$ to keep constant the bit rate; same concept is used when using 3B2T conversion, $66.6MHz \times 2 \div 3$ [2]

2.4.5 Link startup

Link startup[23] between two network nodes is illustrated in Fig.2.14:

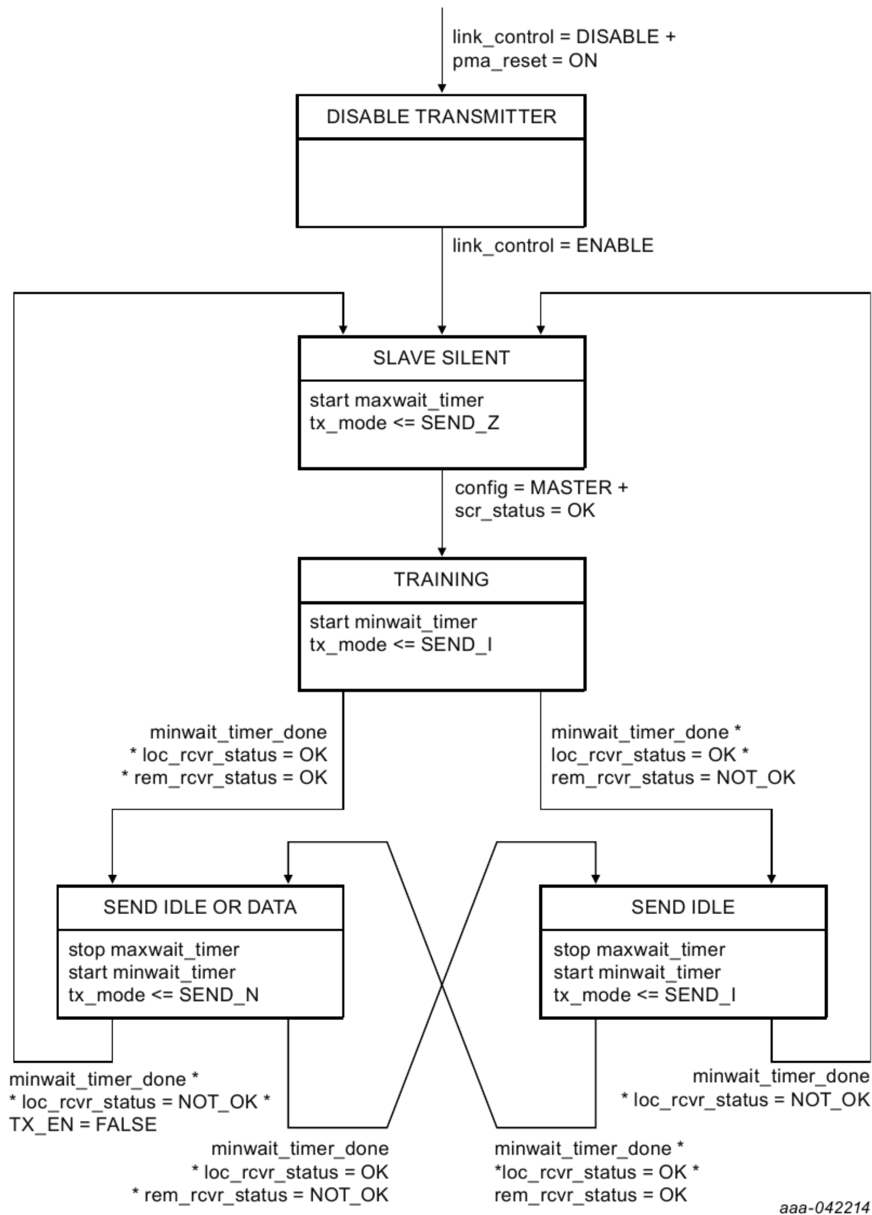


Figure 2.14: 100BASE-T1 link startup process

[2]Source [27], pp7

It highlights some of the aspects of a link startup, where:

- SEND_Z: transmission of all zeroes
- SEND_I: transmission of PAM3 idle signals
- SEND_N: transmission of idle/data signals

Master PHY initiates the training phase by transmitting an idle pattern (SEND_I); after the slave PHY receiver has been synchronized with the IDLE pattern, it enters training state too and sends IDLE pattern to the Master PHY; as soon as *min-wait_timer* expires, the Slave PHY switches to SEND_IDLE state. Now the Master PHY receives an idle pattern from the Slave PHY and synchronizes to the idle pattern to enter SEND_IDLE or SEND_DATA state (SEND_N); finally, when the slave recognizes that the master is synchronized, it enters in turn SEND_IDLE or SEND_DATA state: from now on, the bidirectional link is enstablished and the two nodes can exchange data normally.

2.5 1000BASE-T1 technology

When a higher bit rate is required, using 1000BASE-T1 technology is a suitable solution. However, to support a higher speed, a different cabling and bit modulation are required: 1000BASE-T1 uses 4 UTPs as cable and a PAM5 modulation for bit encoding [9].

2.5.1 Interface circuitry

Due to the higher bit rate, a dedicated circuitry is needed for 1000BASE-T1. According to OPEN Alliance requirements [8] the interface consists of the transceiver block with transceiver supply decoupling and filtering:

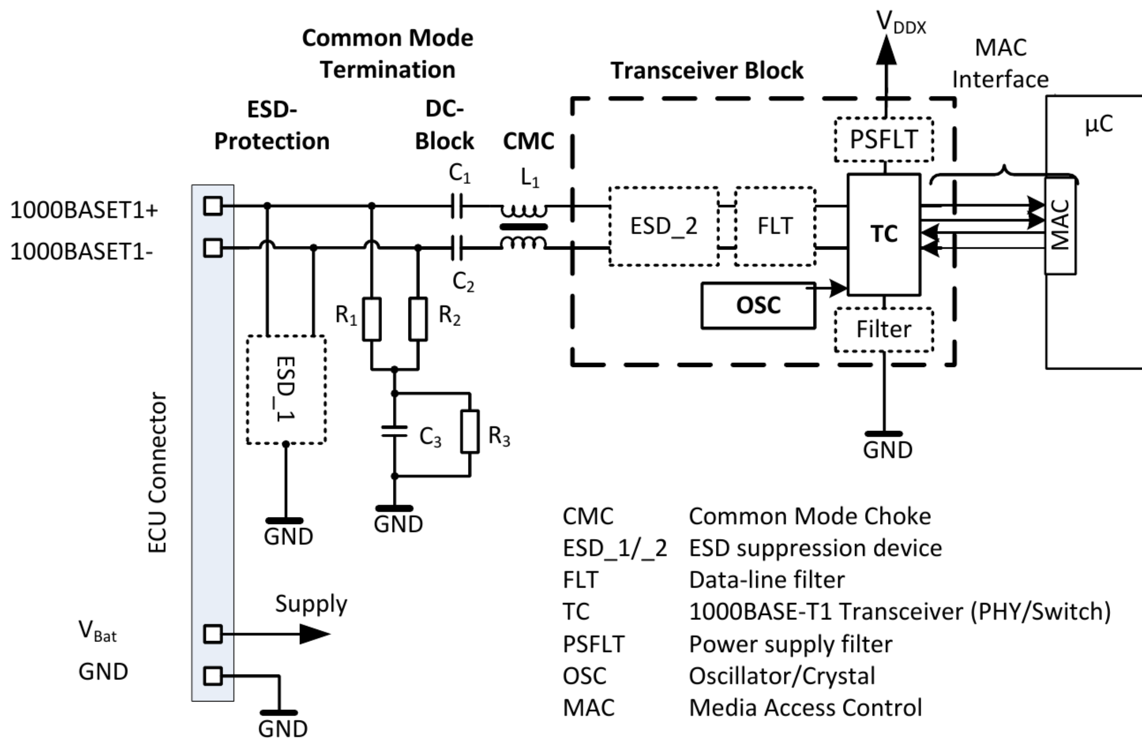


Figure 2.15: Block diagram of interface circuitry 1000BASE-T1

Optional protection can be included in the design, such as:

- Common Node Chocke
- DC block capacitors
- Common mode termination network
- ESD protection

All the required components (mandatory and optional) must be selected properly according to the datasheet of the device provided by the device vendor company, as Fig.2.15 is just a generic block diagram that represents how to interface the components.

2.5.2 PAM5 modulation and wiring

PAM5 bit modulation includes 5 voltage levels to encode data: it uses (-2, -1, 0, +1, +2)V to be sent over 4 wires simultaneously.

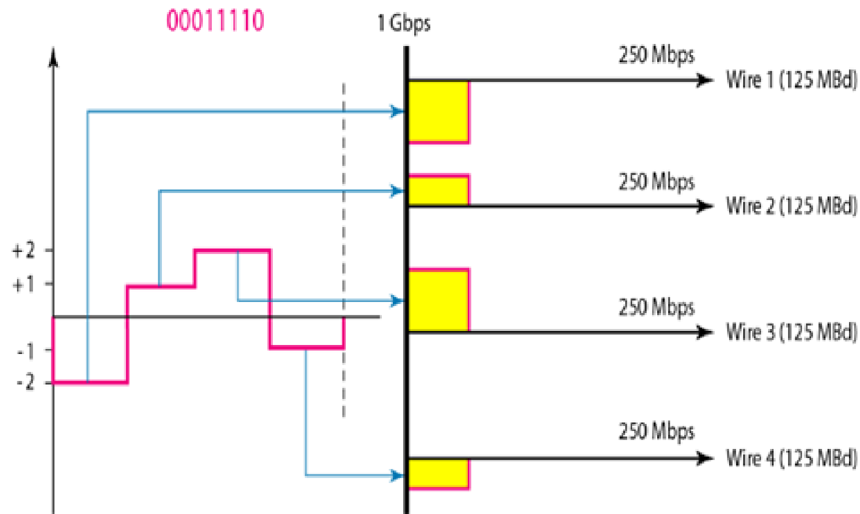


Figure 2.16: PAM5 modulation. Source [2]

The advantages of PAM5 are:

- signal rate is reduced by N/8 as data is sent over 4 wires
- redundancy is used for error detection (value 0 is used for error detection)
- self synchronization is available
- no DC component is used to transmit signal (this saves a lot of static power)

On the other hand, PAM5 uses lot of redundancy because it maps 2^8 input patterns to 4^4 output patterns[3].

2.5.3 Link startup

Link startup process of a 1000BASE-T1 PHY is very similar to the 100BASE-T1 one and it is deeply described in [14], where:

- SEND_Z: transmission of all zeroes
- SEND_I: transmission of PAM3 idle signals
- SEND_T: transmission of training signals
- SEND_N: transmission of data signals

[3]Source [2]

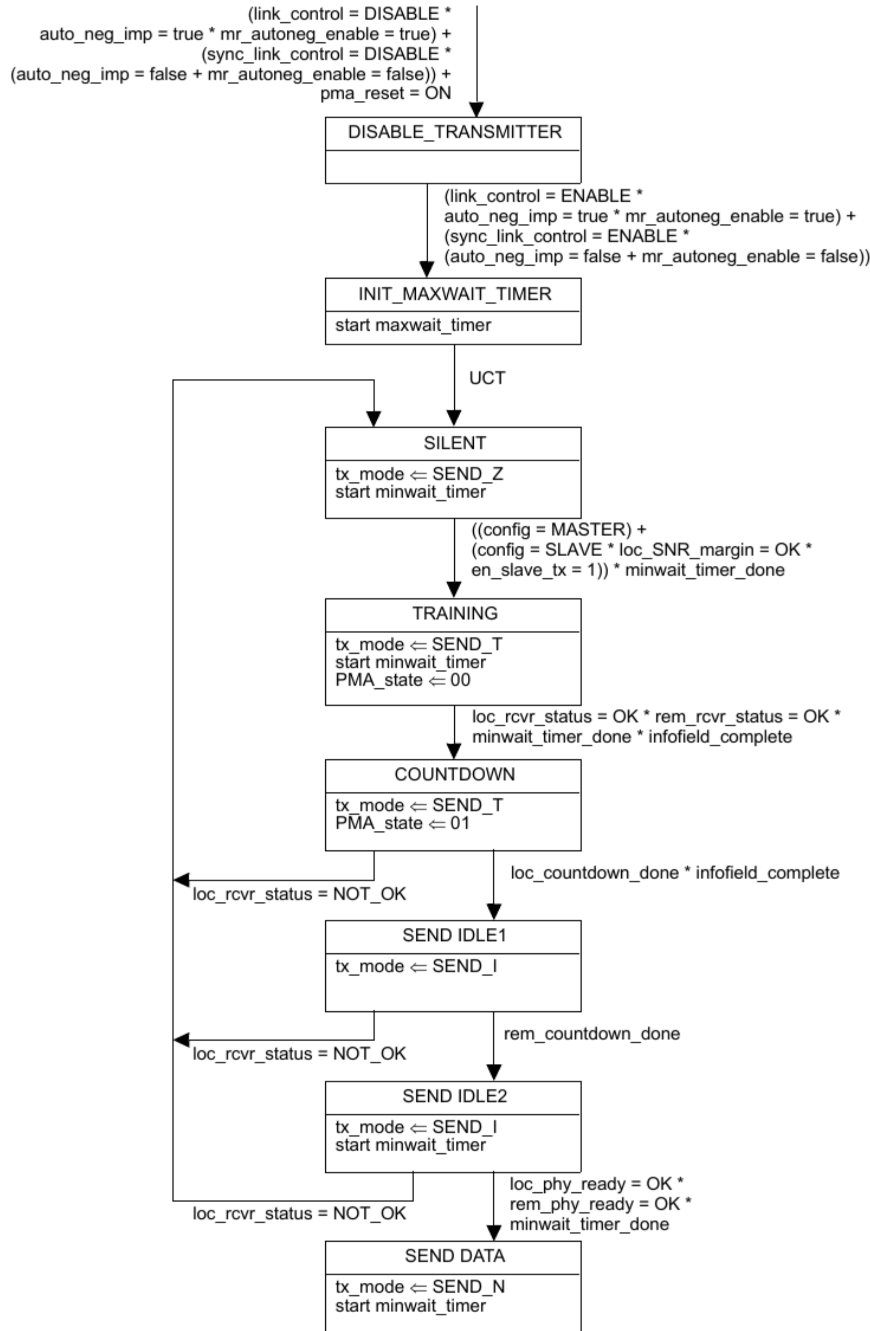


Figure 2.17: 1000BASE-T1 link startup process

Master PHY initiates a training phase by sending a training pattern (SEND_T) for a time determined by *minwait_timer*; after the Slave PHY has been synchronized with the training sequence, Master PHY enters COUNTDOWN state where it sends SEND_T pattern to the Slave for a time determined by *loc_countdown* and the Slave PHY sends an acknowledge SEND_T sequence for a time determined by *rem_countdown*; after that, an IDLE sequence is instanciated the same way as 100BASE-T1 process where the two PHYs synchronize themselves on an idle pattern

(SEND_I) and a stable link is enstablished, and SEND_N data can be exchanged over the network[4].

2.6 TC10 specification for Sleep/Wakeup

Both 100BASE-T1 and 1000BASE-T1 are compliant to TC10 standard: it is a standard defined by OPEN Alliance in document [4] to allow Sleep transition and Wakeup transition for PHYs involved in an automotive environment, because it is not present in IEEE 802.3bw protocol (computer networks do not have the need of sleeping and waking up).
New primitives can be found, such as:

- Low Power Sleep (LPS)
- WakeUp Request (WUR)
- WakeUp Pulse (WUP)

They identify signal patterns exchanged by two link partners, that are a PHY and an external MAC or microcontroller.

[4]Source [14], pp127

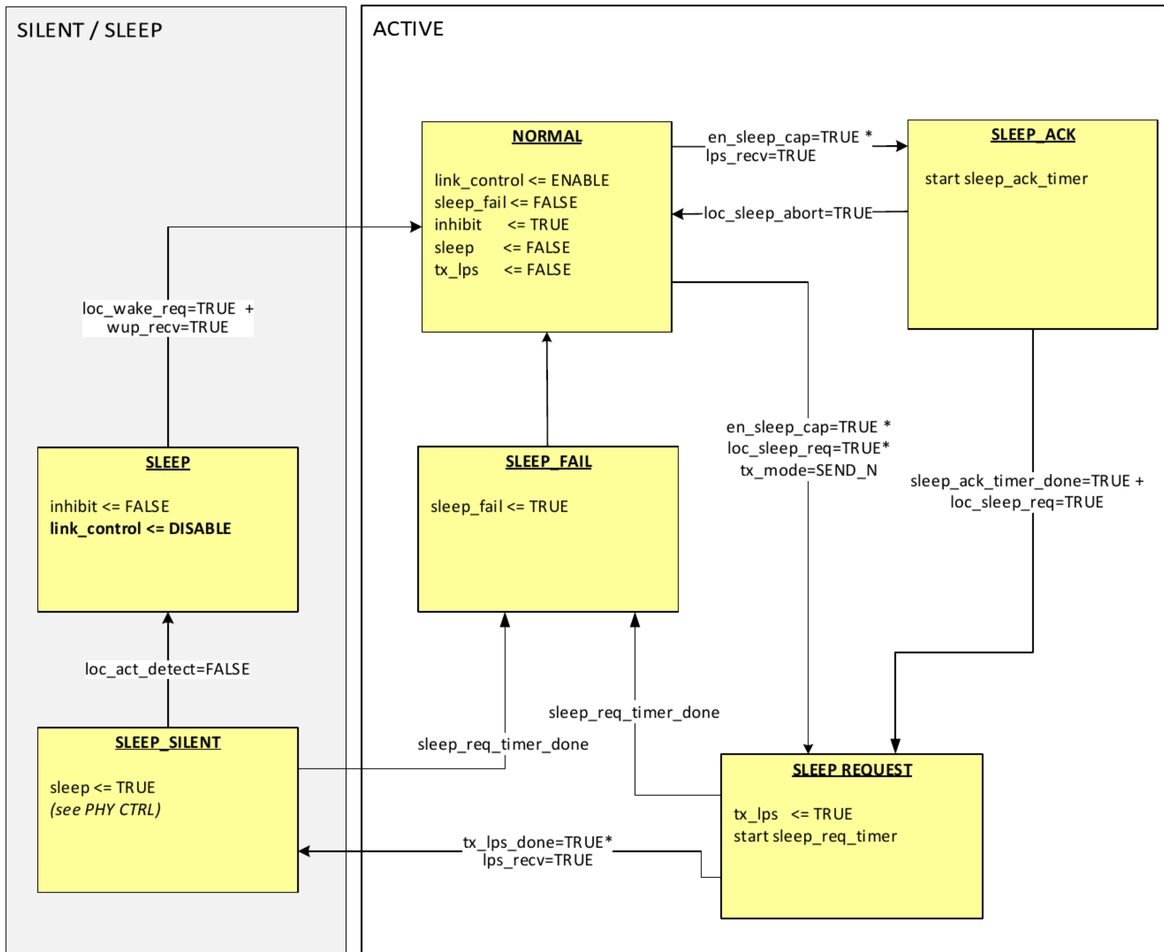


Figure 2.18: TC10 Sleep/Wakeup mechanism

From active to Silent/Sleep mode

If the PHY is in NORMAL (active) state and it receives an LPS code group from its link partner, it enters SLEEP_ACK state: *sleep_ack_timer* is started and if data is detected on MII/MDI peripheral, the PHY returns in NORMAL state; otherwise, if *sleep_ack_timer* expires before data detected, a handshake protocol between MAC and PHY through LPS takes place and *sleep_req_timer* starts; if the handshake process ends before *sleep_req_timer* expires, then the device enters SLEEP state, otherwise it generates *sleep_fail* interrupt and goes back to NORMAL state.

From Silent/Sleep to Active mode

When the PHY is in SLEEP state, it is still able to detect WUP from its link partner: if it receives a WUP or a local wakeup request (i.e. logical value on a dedicated pin), it enters NORMAL state without generating the *sleep_fail* interrupt.

CHAPTER 3

Punch Electronic Platform and ASIC specifications

In order to start developing the virtual model of the TJA1101B, it is required to deeply analyze the Application Specific Integrated Circuit (ASIC) specifications, datasheet and application note.
The device is going to be a piece of a puzzle inside the Punch Electronic Platform (PEP), which is a multi-purpose electronic device that manages the propulsion control and energy generation of the entire vehicle system[28]. PEP design is made in order to be compliant with AUTomotive Open System ARchitecture (AUTOSAR) rules and standards[3], automotive safety and cybersecurity requirements.
This chapter aims to investigate the PHY ethernet block, the configuration memory peripheral, the control blocks and the frame structures.

3.1 Punch Electronic Platform specifications

PEP is the ECU product developed by Punch: it is a propulsion controller able to run a V8 engine propelled with hydrogen. It is optimized for integrating H2 components specific control algorithms and for offering flexibility to test Direct Injection and Ported Fuel Injection (PFI), where gasoline is sprayed into the intake manifold, where it mixes with air, and then is sucked down into the cylinders.
PEP supports Solenoid Control Valves for fuel injection, Direct Current (DC) Motor controllers, analog and digital sensor for data acquisition systems, CAN, LIN and Ethernet peripherals for communication among vehicle peripherals and microcontroller.
PEP exploits Infineon AURIX™ TC399X microcontroller provided by Infineon Technologies[16]: it belongs to the hexa-core high performance architecture family, which allows advanced features for connectivity, security and functional safety for automotive applications; in particular, the ASIC involved provides up to 6 cores with a nominal

frequency of 300MHz, 16MB of Flash memory and 2048 kB of RAM memory.
Among the involved peripherals, it is possible to find:

- 4x Buffered Sensor Supply Outputs for 300mA and 50mA current load
- 12x Boosted Flexible Solenoid Valve Control
- 6x Solenoid Flexible Valve Control Outputs
- Low Side Outputs
- High Side Outputs
- 5x Full H bridges
- PFI injectors
- 2x UHEGO Sensor Heater Drivers
- 4x CAN peripherals
- 3x LIN master nodes
- 1x Ethernet network
- 8x Single Edge Nibble Transmission (SENT) inputs for point-to-point communication between sensor and microcontroller

3.2 TJA1101B overview and pinout

PHY (Physical Layer) is the first subcategory of the OSI model. It's main purpouse is to manage the data transmission on the physical layer.
The overall structure can be divided in 2 macroblocks: one related to the MII/RMII and MDI interface (the PHY itself), and one related to the registers and control blocks driven by Serial Management Interface (SMI)[22].

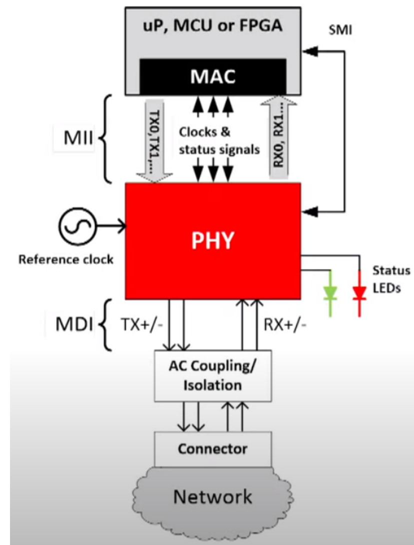


Figure 3.1: TJA1101B overview

3.3 TJA1101B PHY

As said, TJA1101B is compliant with 100BASE-T1, and it is responsible for the transmitting and receiving data from the external MAC through MII and from the external network through the MDI. The PHY is composed by 2 physical sub-blocks, which serialize/deserialize symbols/external bit streams and to map them in symbol maps: these two are Physical Coding Sublayer (PCS) and Physical Medium Attacher (PMA). Their functionalities are here summarized:

- PCS: encodes and decodes payload data. This means that it can serialize/deserialize outcoming/incoming bits for both the transmission/receiving
- PMA: translates bit stream from the PCS into symbol map to be transmitted and received over the PHY network

Others control pins are provided to keep track of the current transmission status:

- TXEN
- TXER
- RXDV
- RXER

Their meaning will be explained in section 3.9 and 3.10.

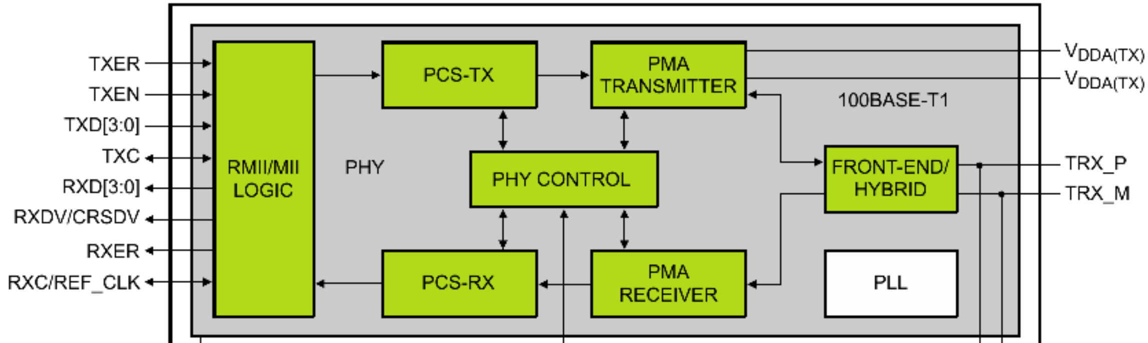


Figure 3.2: Ethernet MII-MDI peripheral

3.4 SMI interface and registers

The MAC can control the PHY behaviour by accessing its internal registers. This is possible due to the SMI interface which receives MAC frames to configure internal registers (i.e. the MAC can perform a read/write operation); it is possible to hardwire some pins of the ASIC on the board for configuration features, and to write on the dedicate registers through a configuration control interface (i.e. select master/slave configuration, select MII/RMII mode, etc...); finally, the ASIC is able to send interrupts and reset signals to the MAC through interrupt and reset control blocks.
The SMI registers are read by the PHY in order to behave according to the configurations; additional blocks are instantiated, such as Under Voltage (UV) and overtemperature detector.

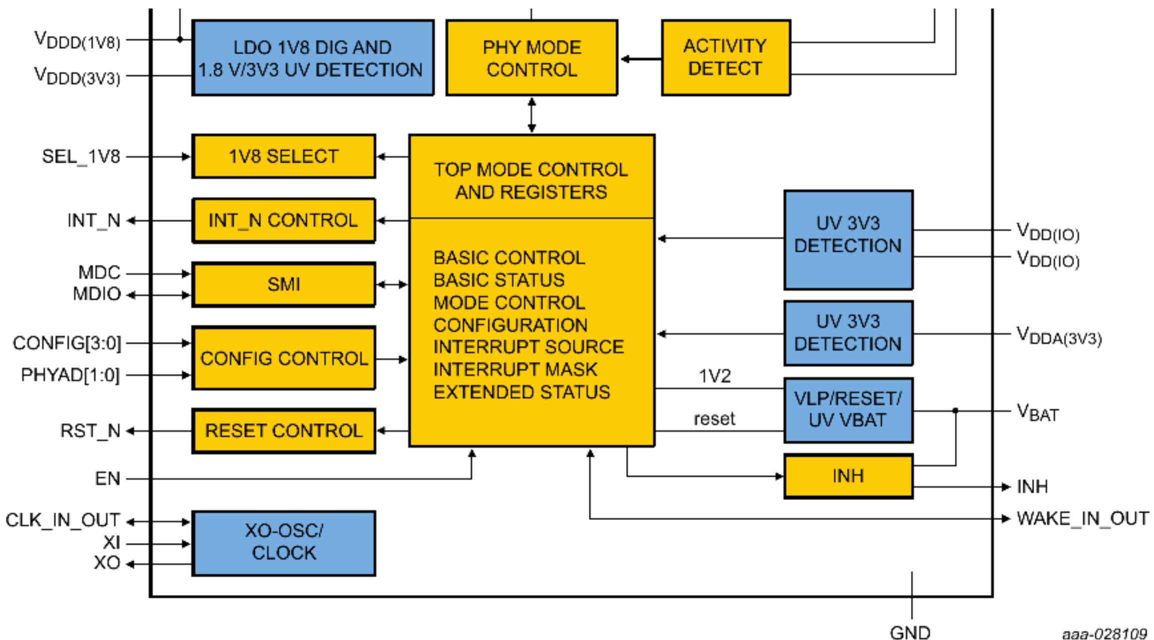


Figure 3.3: SMI interface and control blocks

3.5 Pinout

Here it is reported the PHY pinout, according to the datasheet specifications: for each pin it is reported its number, name and functionality[1].

Pin number	Pin name	Pin functionality
1	MDC	SMI clock input
2	INT_N	Interrupt output (active-LOW)
3	RST_N	Reset input (active-LOW)
4	SEL_1V8	1.8V LDO mode selection (internal or external)
5	XO	Crystal feedback
6	XI	Crystal input
7	VDDA3V3	3.3V supply voltage
8	WAKE_IN_OUT	Local/forwarding wake-up in/out
9	Vbat	Battery supply voltage
10	INH	Inhibit output for voltage regulator control (active-HIGH)
11	VDDAtx	3.3V analog supply voltage for transmitter
12	TRX_P	+ terminal for tx/rx signal
13	TRX_N	- terminal for tx/rx signal
14	VDDAtx	3.3V analog supply voltage for transmitter
15	VDDD3V3	3.3V digital supply voltage
16	VDDD1V8	1.8V digital supply voltage
17	RXER CONFIG3 TXCLK	RXER: MII/RMII receive error output CONFIG3: pin strapping configuration input 3 TXCLK: clock output in test mode and slave jitter test
18	RXDV CONFIG2 CRSDV	RXDV: receive data valid output for MII CONFIG2: pin strapping configuration input 2 CRSDV: carrier sense/receive data valid output (during RMII mode)

[1]Source [22], pp5

Pin number	Pin name	Pin functionality
19	VDD(IO)	3.3V I/O supply voltage
20	CLK_IN_OUT	25MHz reference clock in/out
21	RXD3 CONFIG1	RXD3: receive data output (bit 3 of RXD[3:0] in MII mode) CONFIG1: pin strapping configuration input 1
22	RXD2 CONFIG0	RXD2: receive data output (bit 2 of RXD[3:0] in MII mode) CONFIG0: pin strapping configuration input 0
23	RXD1 PHYAD2	RXD1: receive data output (bit 1 of RXD[3:0] in MII mode or bit 1 of RXD[1:0] in RMII mode) PHYAD2: pin strapping configuration input for bit 2 of PHY address used for SMI address/Cipher scrambler
24	RXD0 PHYAD1	RXD0: receive data output (bit 0 of RXD[3:0] in MII mode or bit 0 of RXD[1:0] in RMII mode) PHYAD1: pin strapping configuration input for bit 1 of PHY address used for SMI address/Cipher scrambler
25	RXC REF_CLK	RXC(MII mode): external 25MHz receive clock output RXC(MII reverse mode): external 25MHz clock input REF_CLK(RMII mode): interface reference clk input (external 50MHz) REF_CLK(RMII mode): interface reference clock output
26	GND	Ground reference
27	VDD(IO)	3.3V I/O supply voltage
28	TXC	MII mode: 25MHz transmit clock output Reverse MII mode: external 25MHz transmit clock input

Pin number	Pin name	Pin functionality
29	TXEN	Transmit enable input (active-HIGH) for MII/RMII mode
30	TXD3	Transmit data input (bit 3 of TXD[3:0] in MII mode)
31	TXD2	Transmit data input (bit 2 of TXD[3:0] in MII mode)
32	TXD1	Transmit data input (bit 1 of TXD[3:0] in MII mode or bit 1 of TXD[1:0] in RMII mode)
33	TXD0	Transmit data input (bit 0 of TXD[3:0] in MII mode or bit 0 of TXD in RMII mode)
34	TXER	Transmit error input (for both MII and RMII mode)
35	EN	PHY enable, active-HIGH
36	MDIO	SMI data I/O

Table 3.1: Pins functionality

Here it is also reported the datasheet pinout to be referenced to:

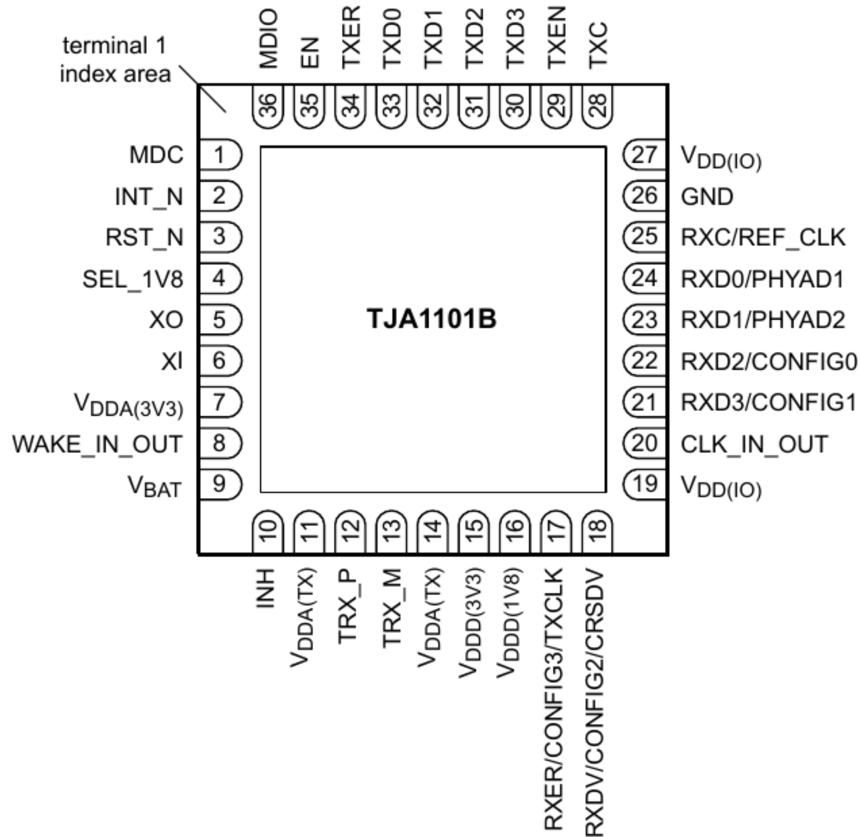


Figure 3.4: TJA1101B pinout

3.6 Functional block and diagram

A detailed implementation of the TJA1101B is shown here: the upper part (highlighted in grey) contains MII/RMII logic, PCS and PMA layers, Front-end/Hybrid block for MDI interface and Phase Locked Loop (PLL) to generate clock signals derived from external crystal or input oscillator signal[2].

[2]Source [22], pp3

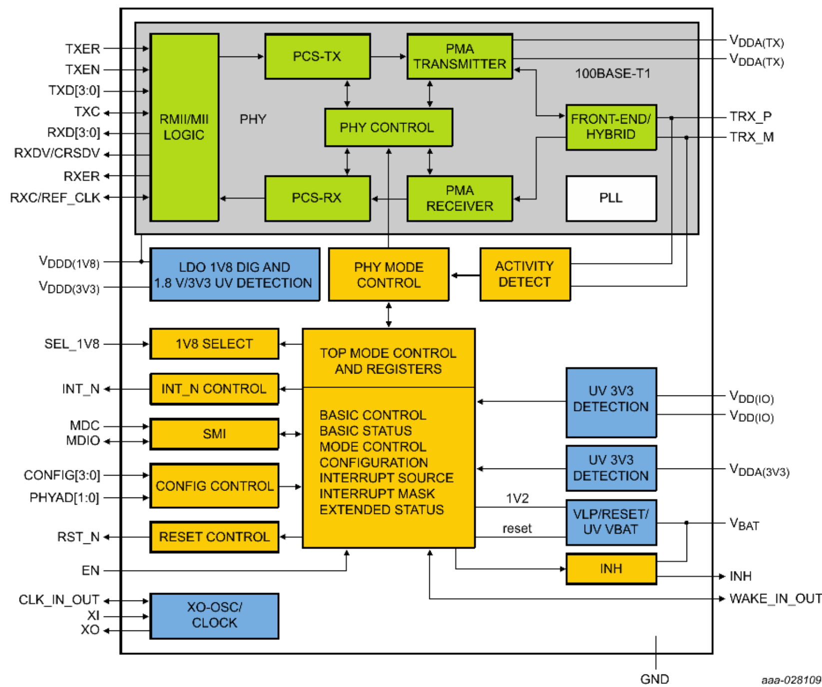


Figure 3.5: TJA1101B block diagram

The lower part (not highlighted) contains logic blocks for SMI, Configuration, Interrupt and Reset signals. On top of that, it is possible to notice the internal registers that are accessed by the interface logic together with undervoltage and overtemperature detection blocks.

3.7 SMI registers

In this section it is reported the SMI register mapping: it is composed of 18 registers of 16 bits width, each one dedicated to a specific purpose. The following table shows their usage[3]:

Register index	Register name
0	Basic control register
1	Basic status register
2	PHY identification register 1
3	PHY identification register 2
15	Extended status register
16	PHYidentification register 3
17	Extended control register
18	Configuration register 1

[3]Source [22], pp24

Register index	Register name
19	Configuration register 2
20	Symbol error counter register
21	Interrupt source register
22	Interrupt enable register
23	Communication status register
24	General status register
25	External status register
26	Link-fail counter register
27	Common configuration register
28	Configuration register 3

Table 3.2: SMI registers numbering and functionality

3.7.1 SMI frame

A single MAC configured as Master can have up to 32 PHYs configured as slave. Communication among MAC and PHYs on SMI interface is based on frame transmission, which should have the following structure[4]:

Preamble [31:0]	Start	OP code	PHY addr[4:0]	Register addr[4:0]	Turnaround	Data [15:0]	Idle
1...1	01	10(read)	AAAAA	RRRRR	Z0	D...D	Z
1...1	01	01(write)	AAAAA	RRRRR	10	D...D	Z

Table 3.3: SMI frame structure

It is important to specify that bit sampling is performed on the rising edge of the clock (MDC because it is referred to SMI interface).

3.8 Hardware configuration

The PHY controller enables several pins dedicated to hardware configuration, such that there is no need of microcontroller interaction. Values corresponding to the configuration options are stored in SMI registers.
The following table summarizes the configurable functions:

Symbol	Pin	Value	Function
MASTER_SLAVE	22(CONFIG0)	0/1	Slave/Master configuration

[4]Source [23], pp9

Symbol	Pin	Value	Function
AUTO_OP	21(CONFIG1)	0/1	Managed/Autonomous operation
MII_MODE	17(CONFIG3) 18(CONFIG2)	00 01 10 11	Normal MII mode RMII mode (50MHz input) RMII mode (50MHz output) Reverse MII mode
PHYAD[2:1]	23(PHYAD2) 24(PHYAD1)	- -	Address bit 2 for SMI Address bit 1 for SMI
LDO mode	4(SEL_1V8)	0 1	Internal 1.8V LDO enabled External 1.8V supply

Table 3.4: Pins for hardware configuration

3.9 MII signal encoding

Using MII mode, data is exchanged between PHY and MAC via 4-bit wide payload. Transmit and receive data is synchronized with transmit and receive clocks (TXC and RXC) which are derived from external clock source or 25MHz crystal oscillator. Data is encoded as follows:

- Transmission

TXEN	TXER	TXD[3:0]	Indication
0	0	0000-1111	Normal interframe
0	1	0000-1111	Reserved
1	0	0000-1111	Normal data transmission
1	1	0000-1111	Transmit error propagation

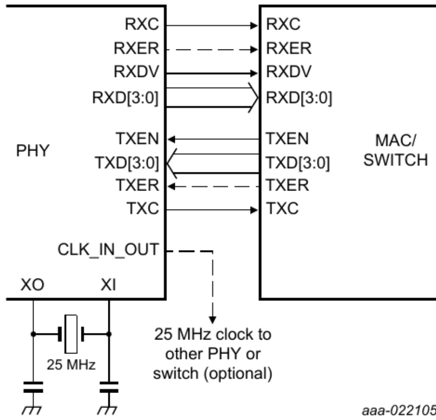
Table 3.5: MII encoding for transmission

- Receiving

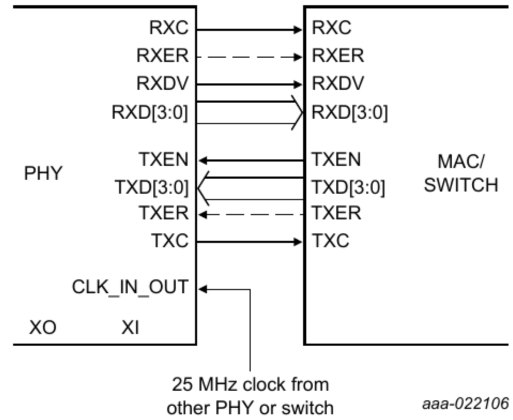
RXDV	RXER	RXD[3:0]	Indication
0	0	0000-1111	Normal interframe
0	1	0000	Normal interframe
0	1	0001-1101	Reserved
0	1	1110	False carrier indication
0	1	1111	Reserved
1	0	0000-1111	Normal data transmission
1	1	0000-1111	Data reception with errors

Table 3.6: MII encoding for reception

TJA1101B MII peripheral can use either the external crystal oscillator (XTAL) clock or an external reference clock, on *XI-XO* pins or on the *clk_in_out* pin (from another PHY or switch) respectively, both running at 25MHz to meet the 100Mbit/s requirement as specified in 2.4.4:



(a) MII XTAL clock



(b) MII external reference clock

3.10 RMII signal encoding

Using RMII mode, data is exchanged between PHY and MAC via 2-bit wide payload. It is used only one single clock signal, REF_CLK, that runs at 50MHz to achieve the same data rate of MII mode: it can be an external 25MHz crystal, 50MHz clock signal generated by an external oscillator connected to REF_CLK or a 25MHz clock signal coming from another PHY connected to CLK_IN_OUT.
Data is encoded as follows:

- Transmission

TXEN	TXD[1:0]	Indication
0	00-11	Normal interframe
1	00-11	Normal data transmission

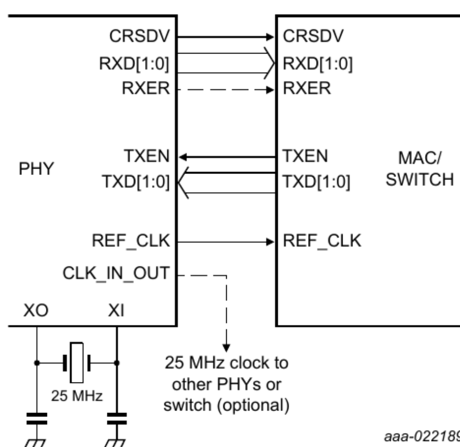
Table 3.7: RMII encoding for transmission

- Receiving

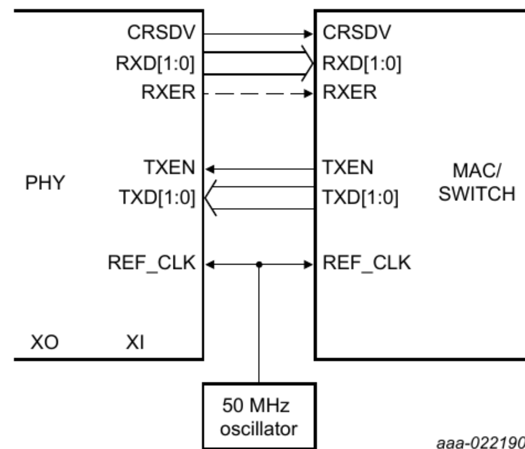
CRSDV	RXER	RXD[1:0]	Indication
0	0	00-11	Normal interframe
0	1	00	Normal interframe
0	1	01-11	Reserved
1	0	00-11	Normal data transmission
1	0	00-11	Data reception with errors

Table 3.8: RMII encoding for reception

Just as MII mode, the RMII can use either an external crystal oscillator clock or an external reference clock: in order to meet the 100Mbit/s requirement, the external reference clock must have a nominal frequency of 50MHz, while the XTAL clock must run at 25MHz and the PHY will derive its internal clock from it, and it will have a nominal frequency of 50MHz.



(a) RMII XTAL clock



(b) RMII external reference clock

3.11 Reverse MII

To realize a repeater function, two PHYs can be connected in series using the MII interface: this is called Reverse MII mode, and it is selected setting MII_MODE = 11 (pin 17 and 18, as specified above).

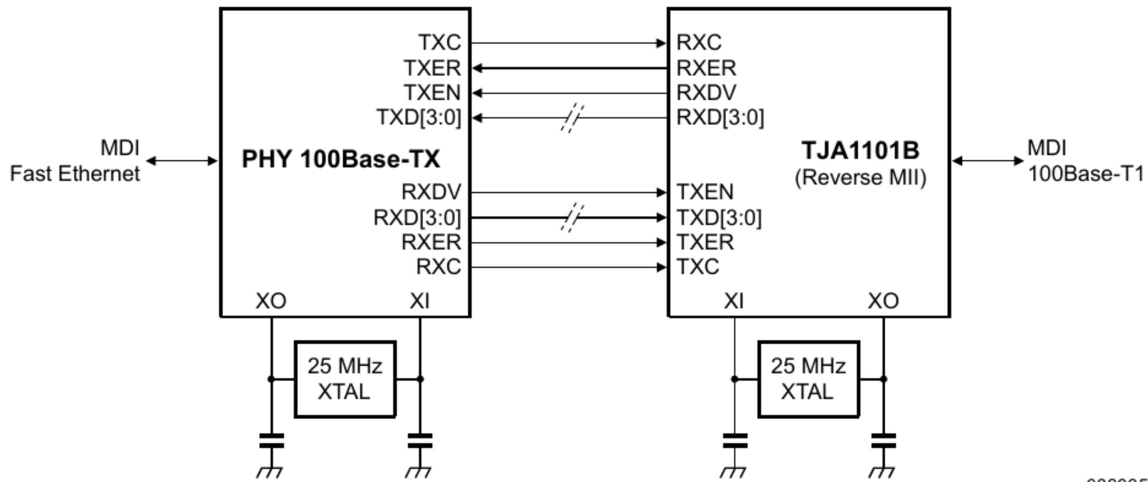


Figure 3.8: Reverse MII configuration

3.12 Loopback

TJA1101 implements the loopback functionality to test the correctness of the sent/received packets through the MII/MDI interfaces by sending back those packets and compare them to the previous ones. Three types of loopbacks are available:

- Internal loopback: packets sent through MII are sent back to verifiy PCS correct behaviour

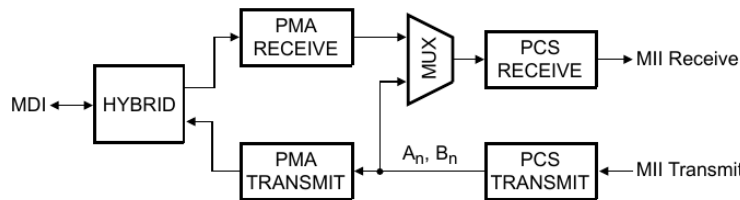


Figure 3.9: internal loopback

- External loopback: packets sent through MII are sent back to verifiy PMA correct behaviour

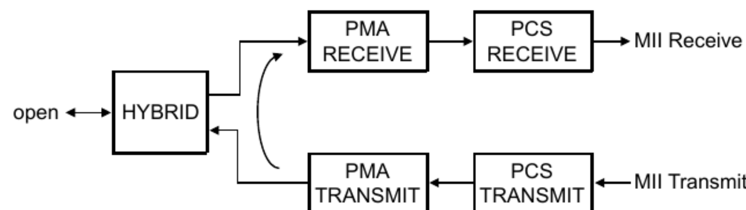


Figure 3.10: External loopback

- Remote loopback: packets received from the MDI are sent back from the PCS receiver to verify the overall correctness of the PHY transmit and receive blocks

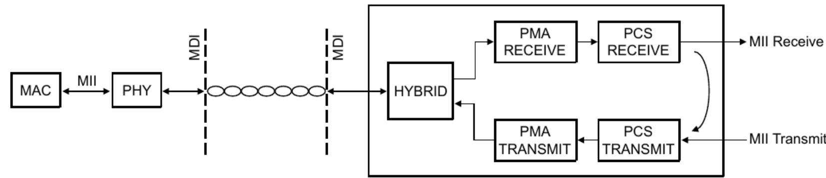


Figure 3.11: Remote loopback

3.13 MDI interface circuit

According to the application note document[5], it is possible to compare the interface circuit diagram of the MDI peripheral with the one provided by the OPEN Alliance document[6]:

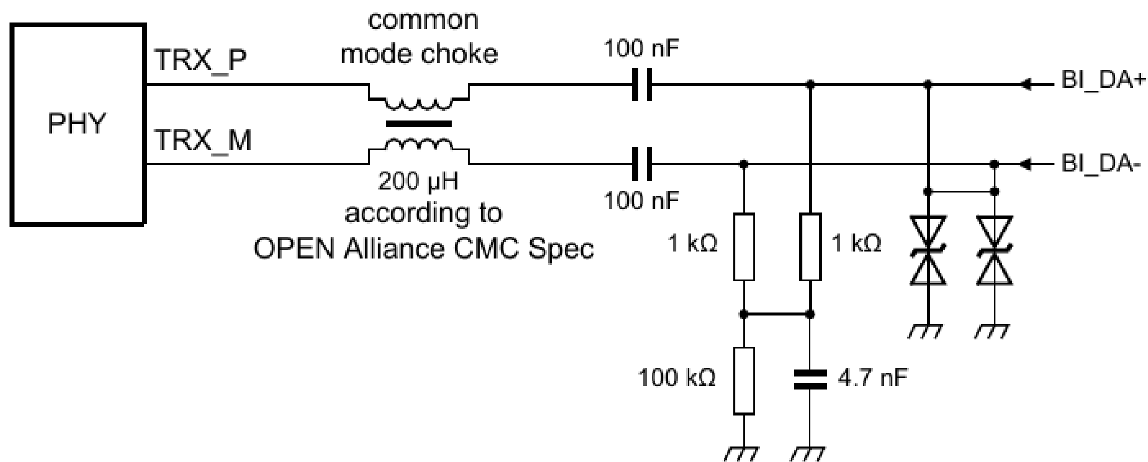
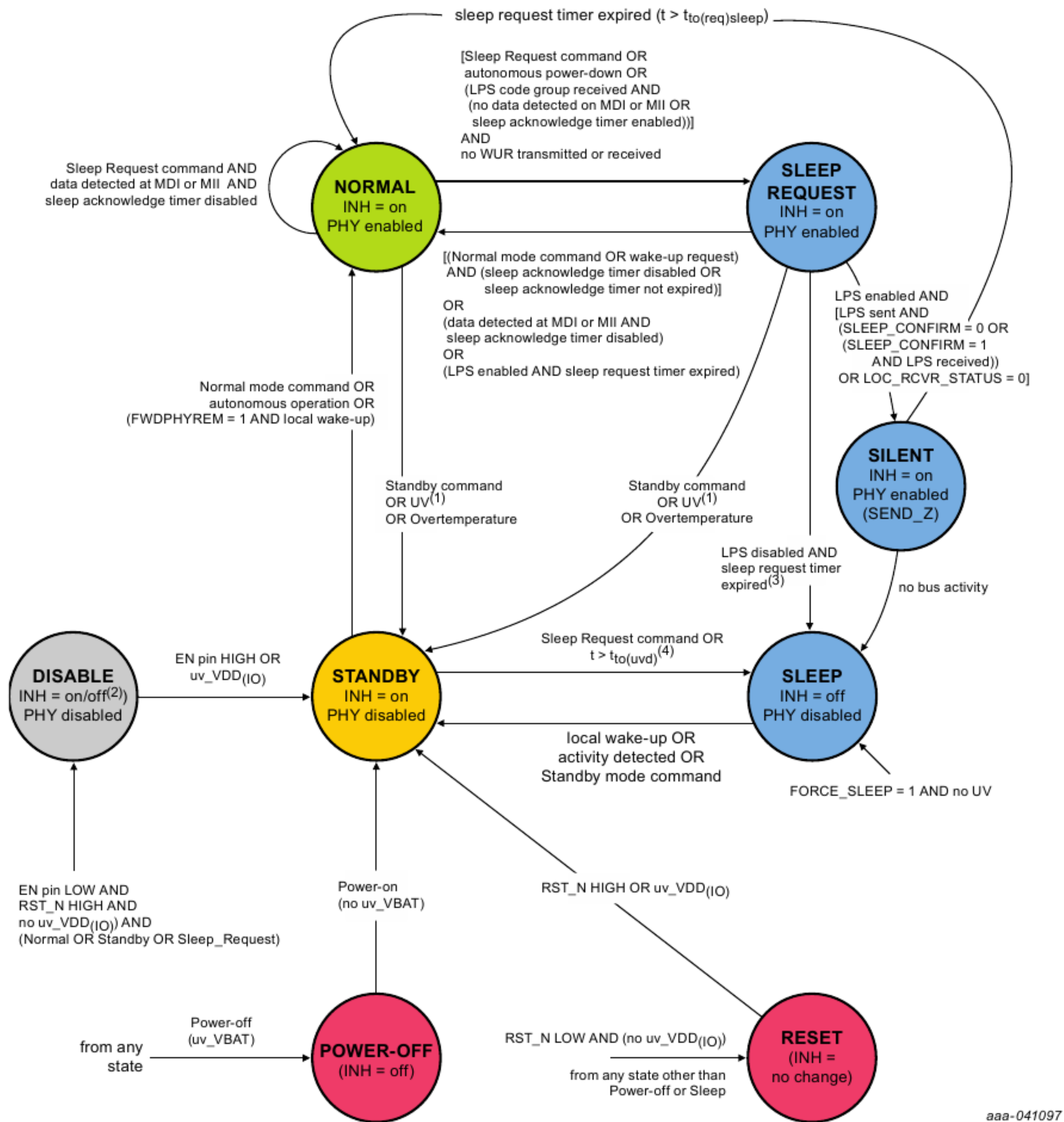


Figure 3.12: MDI interface circuit

It is possible to identify the DC block capacitors C_1 and C_2, the CMC, the LPF and the ESD protection circuit which is composed by protection diodes: each value is reported in Fig.3.12.

[5]Source [23], pp10
[6]Source [10], pp11

3.14 Finite state machine



1. UV means undervoltage on one of the power supply pins $V_{DD(IO)}$, $V_{DDA(3V3)}$, $V_{DDD(1V8)}$, $V_{DDD(3V3)}$.
2. INH can be configured to be on or off.
3. The PHY will not be in Sleep mode, and cannot be woken up, until the timeout associated with the transition has expired (after $t_{to(req)sleep}$).
4. At power-on, after a transition from Power-off to Standby mode, undervoltage detection timeout is enabled once all supply voltages are available. When an undervoltage is detected, the TJA1101B switches to Sleep mode after $t_{to(uvd)}$.

Figure 3.13: Finite state machine diagram

Fig.3.13 is taken from the datasheet and shows signals responsible for state changing. Depending on the current state of the ASIC, only a limited number of operations are allowed.

3.14.1 DUT operations allowed

- POWER OFF
  - Communication on SMI interface is disabled
  - Communication on MII/MDI is disabled
  - Reset SMI registers to default value
  - Read power mode is disabled
- DISABLE
  - Communication on SMI interface is disabled
  - Communication on MII/MDI is disabled
  - Do not reset SMI registers to default value
  - Read power mode is disabled
- RESET
  - Reset SMI registers to default value
  - Communication on SMI is disabled
  - Communication on MII/MDI is disabled
  - Capture pin strapping configuration
  - Read power mode is disabled
- STANDBY
  - Set INH output pin to HIGH
  - Set PHY configuration mode according to pin strapping
  - Communication on SMI interface is enabled after ts_{Pon} from the time STANDBY state is entered
  - Read power mode is enabled
  - Reset *req_ timer* and *ack_ timer*
- NORMAL
  - The ASIC transmit and receive functions are enabled and can setup a link
  - MII/MDI communications enabled
  - SMI communication enabled
  - Read power mode is enabled

- Reset *req_timer* and *ack_timer*
- Set INH pin to HIGH

- SLEEP
  - Communication on MII/MDI is disabled
  - Set INH output pin to LOW
  - Do not reset SMI registers to default value
  - Only POWER_MODE bitfield reading is allowed on the SMI interface
- SLEEP REQUEST
  - Start $t_{to(req)sleep}$;
  - If *sleep_ack_timer* is disabled, it switches to NORMAL mode if data detected at MII and MDI interface, set DATA_DET_WU and generate WAKEUP interrupt if REMWUPPHY = 1;
  - If *sleep_ack_timer* is enabled, start $t_{to(ack)sleep}$ and data at MII and MDI interfaces is ignored;
- SILENT
  - Communication on MII, MDI and SMI is allowed

3.14.2 Relevant timing

During state transition, some relevant timings must be considered:

- POWER-ON → STANDBY: ts_{Pon} = 2ms (no SMI access during ts_{Pon})
- From any state (except from NORMAL) → NORMAL: $t_{init(PHY)}$ = 2ms (no link can be established before $t_{init(PHY)}$)
- NORMAL → SLEEP REQUEST: $t_{to(pd)autn}$ = 1-2sec
- Sleep request time-out time starts when TJA1101 enters SLEEP REQUEST MODE: $t_{to(req)sleep}$ = 400μs when SLEEP_REQUEST_TO = 0, 1ms when SLEEP_REQUEST_TO = 1, 4ms when SLEEP_REQUEST_TO = 2, 16ms when SLEEP_REQUEST_TO = 3
- Sleep acknowledge time-out time starts in SLEEP REQUEST MODE if SLEEP_ACK is set: $t_{to(ack)sleep}$ = 200μs when SLEEP_REQUEST_TO = 0, 500μs when SLEEP_REQUEST_TO = 1, 2ms when SLEEP_REQUEST_TO = 2, 8ms when SLEEP_REQUEST_TO = 3

- Under-voltage detection time-out time = to_{uvd} = 670ms
- DISABLE → STANDBY: $t_{det(EN)} = 20\mu$s
- RESET → STANDBY: $t_{det(rst)} = 20\mu$s

3.15 Undervoltage detection methods

The ASIC behaviour strongly depends on the voltages upon power pins. Voltages must be modelled using *sc_in<double>* ports, and each time one or more pins changes its value, the state transition method checks whether and undervoltage has occurred to properly switch the current state. Here there are listed the voltage thresholds for the power pins:

- Vdd_io: 3.3V
- Vddd3V3: 3.3V
- Vdda3V3: 3.3V
- Vddd1V8: 1.8V
- V_bat: 3.3V

3.16 Overtemperature and Temperature warning methods

Temperature, just as voltages, has to be constantly monitored to change the state if an overtemperature has occurred. Proposal: it can be modelled as an *sc_signal<double>*, and the state change method is sensitive to it such that it can check the temperature value.
Overtemperature values: (180 – 200) °C
Temperature warning values: (155 – 175) °C

3.17 Interrupt handling

Register 21 is responsible for keeping track of all the interrupts events that occur between 2 read operations of the register itself. Here it is possible to identify all the events that can generate an interrupt:

- Power-on
- Wake-up (local or remote)

- LPS code group received
- UV (undervoltage on VDD_IO or VDDD3V3 or VDDA3V3 or VDDD1V8)
- Overtemperature
- SMI error (any value written in POWER_MODE different from normal_cmd, stand-by_cmd, sleep_mode, silent_mode)
- UV recovery
- Sleep abort

If the interrupt is not masked (corresponding bitfield in register 22 set to 1) then the TJA1101 sets the INT_N pin to LOW. If instead a read operation is performed on register 21, the pin INT_N is set to HIGH.

3.18 PHY wakeup concept

Looking at the TC10 mechanism described in the document provided by OPEN Alliance, several things can be deduced regarding the wake-up protocol: the PHY can receive a wake-up command on the *wake_in_out* pin or through the MDI interface on the reception of a WUR from the link partner. The PHY can react to the wake-up call if the corresponding bit in register 18 is enabled (LOCWUPHY for local wake-up on *wake_in_out* pin, REMWUPHY for remote wake-up on the MDI interface); if the bit is not set, the PHY won't recognize these commands and does not wake-up from a sleep state. PHY local wake-up time is determined by the value present in bitfield LOC_WU_TIM in register 27:

- 00: 10-20ms
- 01: 500μs
- 10: 200μs
- 11: 40μs

Regarding the forwarding concept, it is a method to propagate a wake-up request over an entire network of multiple PHYs connected: if the local forwarding bitfield is set and a remote wake-up request is detected, the PHY will propagate the request by pulling the wake_in_out pin high; on the other hand, if the remote forwarding bit is set and a local wake-up request is detected, the PHY will propagate the request by sending WUP on the MDI peripheral.

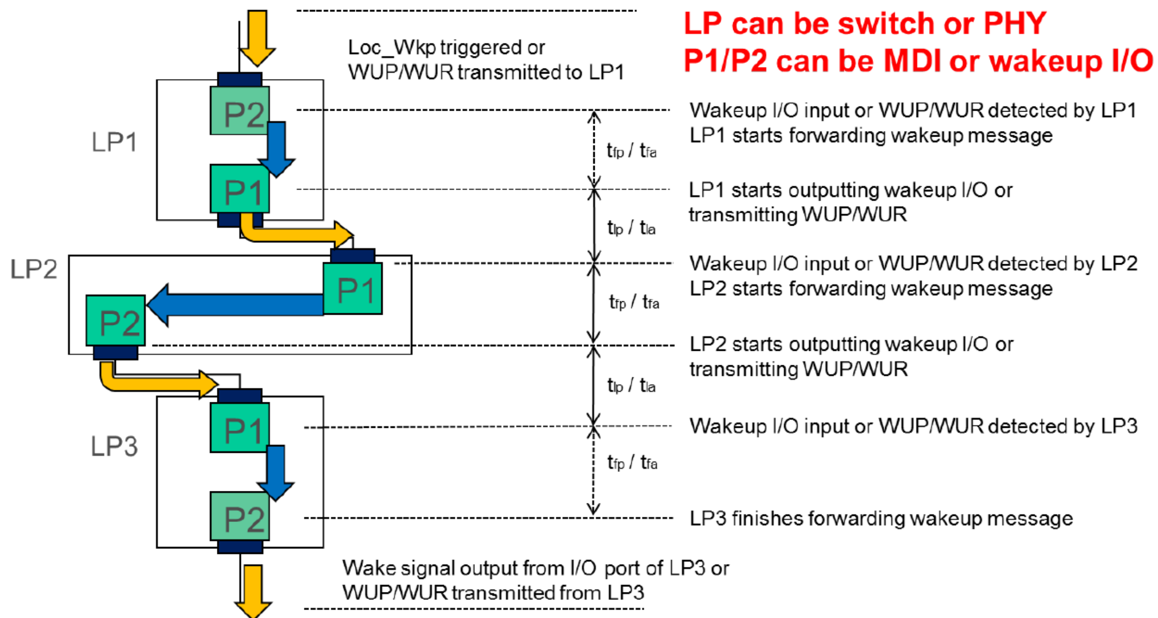


Figure 3.14: TC10 wakeup implementation

3.19 PHY sleep concept (LPS)

The PHY can enter a sleep request state by receiving a LPS code group: looking at the TC10 specification, the encoding of a LPS code group is is performed by the bit scrambler of the link partner, just as the encoding of a remote wakeup frame. In the virtual model, it is reasonable to use the length/type field of the ethernet frame as a LPS code field (it can be set to a random value, greater than 1536(dec)). This mechanism implies that a handshake will be required to confirm the state transition. When the PHY enter SLEEP_REQUEST state, the timer $t_{to(req)sleep}$ timer will start and the PHY sends LPS code group to the link partner to inform it that it is switching to a SLEEP state: if the timer expires before the link partner has sent a LPS code group back as acknowledgement for the ASIC, the PHY will switch back to NORMAL state; on the other hand, if the handshake protocol is satisfied, the PHY will enter SLEEP state (through SILENT state). Moreover, the TJA1101 offers the possibility to wait for a wake-up host command thanks to the SLEEP_ACK bitfield in register 28. If the bit is set, the $t_{to(ack)sleep}$ timer is started when the SLEEP REQUEST state is entered, and it determines the maximum time amount in which the PHY can receive a wake-up command: if the timer expires before receiving a WUP or WUR, the PHY switches to SLEEP state, otherwise to NORMAL state; if the bit is not set, it switches back to NORMAL state if data is detected at MII or MDI interface and generate a SLEEP_ABORT interrupt.

CHAPTER 4

Model requirements

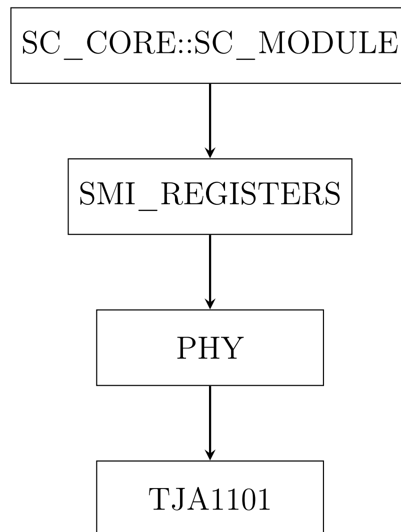
The aim of this chapter is to describe how the virtual model requirements have been implemented using the SystemC[5] library and the SCML2 2.8.0 library[13].
In particular, the followings items are described:

- TJA1101B hierarchy
- Pinout
- MII interface
- MDI interface
- SMI interface
- SMI registers
- Pin-strapping hardware configuration
- Internal loopback
- External loopback
- Remote loopback
- Ethernet frame class description
- Finite State Machine
- Undervoltage detection methods
- Overtemperature and temperature warning detection methods
- Interrupt handling
- Local/Remote wake-up

- LPS
- Ethernet sniffer

4.1 TJA1101B hierarchy

To keep the overall design well organized and clean, the TJA1101 virtual model is divided into 3 classes: SMI_registers, PHY and TJA1101.



First, the entire design must inherit all the methods coming from the *sc_ core::sc_ module* class, such that it is identified as an hardware model by the Virtualizer tool. After that, the SMI_register module is defined as the parent class of the PHY module, which is in turn the parent class of the top module TJA1101.

4.2 Pinout

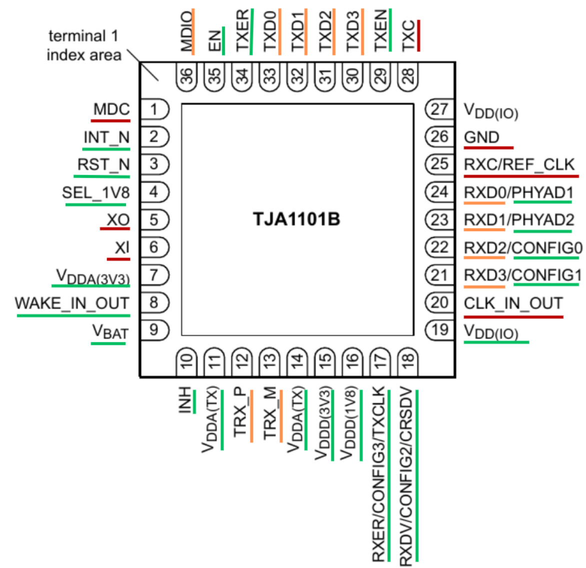


Figure 4.1: Pinout implementation
Green:implemented
Orange: abstracted
Red: omitted

In Fig.4.1 it is highlighted how the device pins have been modelled in the virtual model: they can be implemented, abstracted or omitted.
Only pins that are responsible for a behaviour change of the Device Under Test (DUT) have been implemented or abstracted, clock pins have been omitted in order to speed up the simmulation because they don't act on the model behaviour.

4.3 MII and MDI interface

The need of trans receiving an ethernet frame and move it on the other side of the PHY (from MII to MDI and vice versa) leads to an implementation with 2 target sockets, to handle transactions instantiated by the external MAC/network, and 2 initiator sockets to transfer the generic payload to the external network/MAC.
In Fig.4.2 it is shown how it has been implemented:

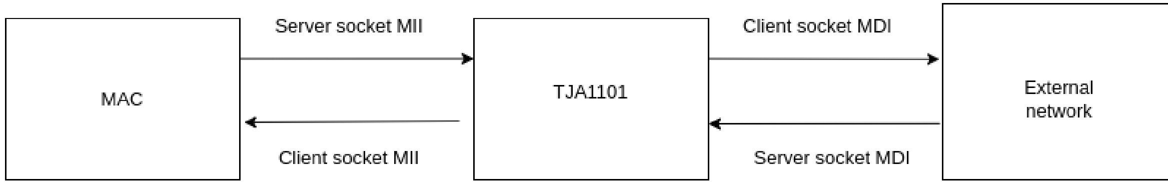


Figure 4.2: TLM representation of MII and MDI interface

Server sockets are defined as tlm_utils::tlm_simple_target_socket<PHY>, while client sockets as tlm_utils::simple_initiator_socket<PHY>. Each time a server socket detects a transaction, the PHY forwards the transaction itself to the other link partner if the operation is allowed (i.e. the current state is NORMAL and the PHY is enabled). As specified by the datasheet, a normal data transmission is identified by the pin configuration: txen = 1 AND txer = 0, otherwise it identifies an interframe or an error propagation (both cases are considered as errors in the virtual model because they are not carrying useful information).

4.4 SMI interface

The SMI interface communication is managed by a tlm_utils::simple_target_socket<SMI_registers>, which is a socket that calls the memory *b_transport* function to write/read to/from the registers of the ASIC, if the operation is allowed.
The TLM version of the SMI frame does not involve the preamble bytes and the start of frame byte: they do not carry any useful information, and also it is assumed that the entire system is always synchronized; information like OP_CODE and ADDRESS are set in the generic payload exchanged among sockets (using methods *set_command()* for the OP_CODE and *set_address()* for the ADDRESS).

4.5 SMI registers

SMI register file is be implemented using SCML2 library: it is modelled as a class containing the memory map and registers. Memory map is a *scml2::memory* object of unsigned short elements, because the register parallelism is on 16 bits; it is the parent class of objects of type *scml2::reg*, which are all the registers listed above. SCML2.8.0 librar allows to set read/write callbacks and to set read/write restrictions on memory objects that will facilitate the debugging phase.
List of remarkable register callbacks:

- Register 0: if bitfield RESET = 1, then generate a software reset event

- Register 17: POWER_MODE bitfield contains the information for software commands which are useful for state transitions, respectively NORMAL_MODE, STANDBY_MODE and SLEEP_MODE
- Register 18: whenever an access takes place on register 18, the callback checks if the bitfield CONFIG_EN in register 17 is set to 1 in order to grant the required access, otherwise it is ignored
- Register 21: the interrupt source register must be cleared after a read operation
- Register 23: RECEIVE_ERR and TRANSMIT_ERR bitfields must be set to 1 after a read operation
- Register 24: LOCAL_WU, REMOTE_WU, EN_STATUS must be set to 0 after a read operation;
- Register 26: must be reset to 0 after a read operation

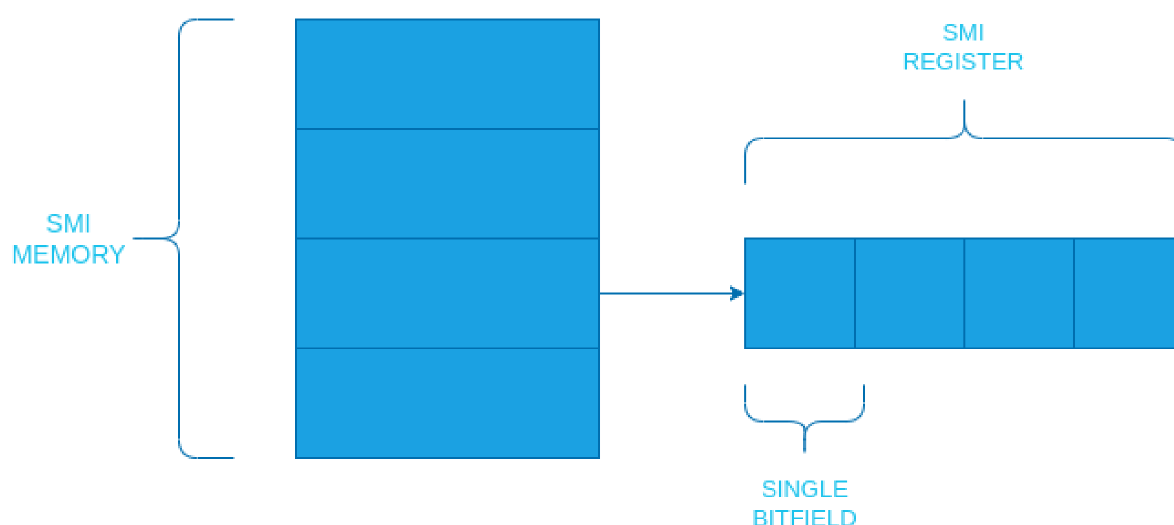


Figure 4.3: SMI TLM memory structure

4.6 Internal and external loopback

Loopback is a method that is useful to test the correct functionality of the PCS and PMA blocks.
In the virtual model they are abstracted, but since this function can be configured via SMI it has been virtually implemented.
For what it concerns the internal and external loopback, the implementation is the same for both: if the LOOPBACK is enabled and an ethernet frame is received at the

MII target socket, the MII initiator socket will initiate a transition back with the same received payload, otherwise not.

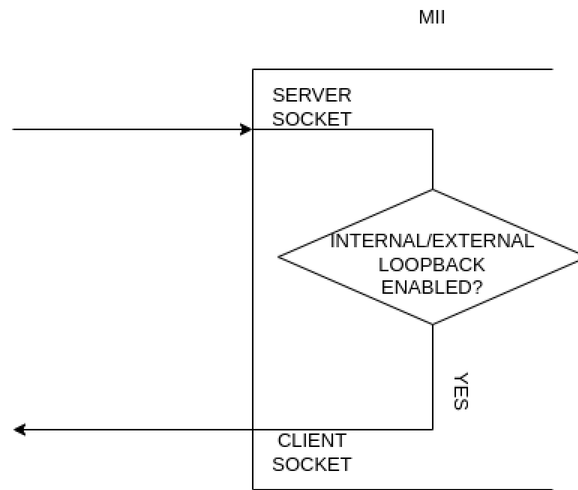


Figure 4.4: Block diagram of the virtual internal and external loopback

4.7 Remote loopback

The remote loopback virtual implementation is the specular with respect to the internal/external one, the difference is that the loopback is on the MDI interface.

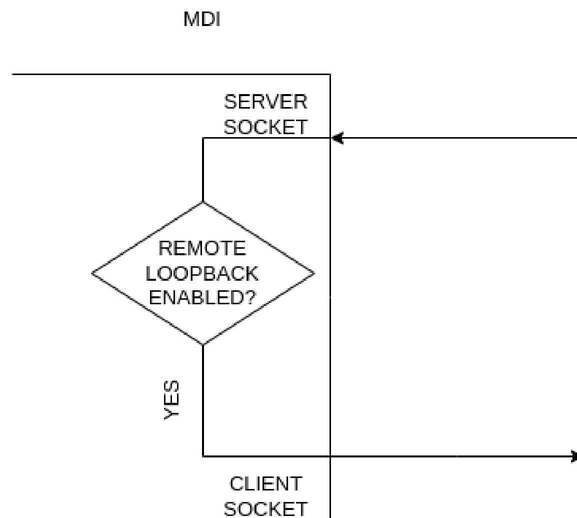


Figure 4.5: Block diagram of the virtual remote loopback

4.8 Ethernet frame class description

The ETHERNET_FRAME class contains all the methods to set the frame fields and to get the field values from the frame. The frame has been modelled as an unsigned char array and set/get functions act on this array. It is important to specify that array bytes are stored in Big Endian mode (MSB is stored in the lower memory region).
Every time the sockets exchange ethernet frames, the payload array is copied in the memory region of the destination payload array.
In the TLM implementation, the preamble and Start of Frame Delimeter bytes are not implemented because it is assumed that the entire system is already synchronized.
Data payload size has to support different sizes of ethernet frames: standard ethernet frame size is usually 1518 bytes, while longer frames are classified as jumbo frames; according to the configuration of JUMBO_ENABLE bitfield in register 19, the PHY is able to support up to 4kB of payload when JUMBO_ENABLE = 0 or up to 16kB when JUMBO_ENABLE = 1.

4.9 Finite state machine

The FSM has been implemented as a Moore state machine, using two different *sc_ methods*. The first one, *change_state()*, implements the state transitions according to the following list of events and signals:

- Battery voltage
- Vdd_io
- Vddd3V3
- Vdda3V3
- Vdda1V8
- Hardware reset
- Software reset
- Hardware enable
- Temperature
- Ethernet frames on MII and MDI
- SMI mode commands (normal, standby, sleep)
- Local wakeup

- Remote wakeup
- Sleep request timer expiration
- Acknowledge timer expiration
- LPS code group

After a state transition, this method notifies the start of the virtual control unit, *FSM()*, after the proper time as specified by the datasheet.
The role of the control unit is to configure the PHY control signals according to the current state: for example it enables/disables the Ethernet peripherals, starts the timers for sleep request transition, resets the registers, so on and so forth.

4.10 Undervoltage detection methods

Voltage upon power pins has been modelled with sc_in<double>ports, and they are constantly read by the methods in charge of changing the DUT behaviour:

- State machine: to go POWER-OFF or STANDBY if undervoltage detected on V_bat or digital and analogue pins
- External status register
- Interrupt source register

4.11 Overtemperature and temperature warning detection methods

The temperature has been modelled as an sc_signal<double>, and it is set from the testbench to test correct state transitions and registers update, and it is monitored by:

- State machine
- External status register
- Interrupt source register

If the temperature value is between 180 and 200 it means that an overtemperature event has occurred.
If the temperature value is between 155 and 175 it means that a temperature warning event has occurred.

4.12 Interrupt handling

For what it concerns the interrupt handling, the TJA1101 register 21 is the interrupt source register, while register 22 is the register that contains the bitfields associated with one interrupt and, if set, pulls the INT_N pin LOW if that corresponding interrupt occurs.
The interrupt generation is modelled with an *SC_METHOD*, which is sensitive to all the events listed in the datasheet: it checks what is the interrupt source and updates the corresponding bit; after that, it checks whether the mask bit in register 22 is enabled, and it sets the INT_N pin LOW if it is 1. If a read operation event is triggered on register 21 via read callback, it pulls INT_N at a HIGH value, and the register is automatically reset to its default value.

4.13 Local/Remote wakeup

The local wakeup has been modelled through the *wake_in_out* pin that is a sc_inout<bool>object, and an *SC_METHOD* that notifies a local wakeup event according to the configuration written in register 18, after the time specified in the LOC_WU_TIM bitfield.
On the other hand, the remote wakeup has been modelled through an ethernet frame whose length/type field contains a *WAKE_ON_LAN* code 0x842, and it is sent on the MDI interface.

4.14 Low Power Sleep (LPS)

The LPS code group has been encoded with 0x900 value in the length/type, which is a random value greater than 1536(dec), because, according to the TC10 standard, the encoding of the LPS is left to the implementer. If the device is in NORMAL state and it receives a LPS command, it will go in SLEEP_REQUEST state and start $t_{to(req)sleep}$ and $t_{to(ack)sleep}$ according to SMI registers values.

4.15 Ethernet sniffers

The virtual model of the TJA1101B comes out with a further feature which is useful for monitoring the ethernet packets exchanged over the network and to visualize them on the Virtualizer tool: ethernet sniffers have been developed to capture packets on the MII interface and to save data in *scml2::memory* objects every time a frame is detected on the peripheral. For each of the frame types, the model traces the followings:

- DATA LINK

  - VENDOR ID SRC
  - VENDOR ID DST
  - HOST IS SRC
  - HOST IS DST
  - LENGTH/TYPE

- IP
  - VERSION
  - HEADER LENGTH
  - TOS
  - TOTAL LENGTH
  - IDENTIFICATION
  - FLAGS
  - FRAGMENTED OFFSET
  - TTL
  - PROTOCOL
  - CHECKSUM
  - SOURCE ADDRESS
  - DESTINATION ADDRESS
  - OPTIONS

- TCP
  - SOURCE PORT
  - DESTINATION PORT
  - SEQUENCE NUMBER
  - ACK NUMBER
  - HEADER LENGTH
  - RESERVED
  - CONTROL FLAGS
  - WINDOW SIZE
  - CHECKSUM
  - URGENT POINTER
  - OPTIONS

- UDP
  - SOURCE PORT
  - DESTINATION PORT
  - CHECKSUM
  - PAYLOAD LENGTH

Each of the listed field is modelled through a *scml2::reg* memory object properly mapped in the parent memory object.
In addition to this information, the sniffers trace the frame number captured on the interface and the elapsed time from the beginning of the simulation, expressed as unsigned integer number to represent the elapsed nanoseconds. Payload data is omitted because it does not contain useful information about the ethernet packet communication fields.

4.16 Live demostration

To provide a live demostration of the Ethernet sniffers, the testbench comes with an ethernet client socket that receives all packets from the network when the PC is connected to a LAN. Each time a packet is captured on the network, it is copied in the memory location of the ethernet payload of the testbench and sent to the MII interface: in this way, it is possible to compare in real time packets sniffed using C Application Programming Interface (API) and with Wireshark.

CHAPTER 5

Test environment

After the virtual model developing, it is necessary to build a proper test environment to verify all the required functionalities of the device. In this chapter it is described how the testbench is implemented, the list of the tests that are performed, the waveforms to analyze the behaviour of each feature and the dump of the ethernet sniffers.
Finally, it will be shown how the virtual model will be visualized on the Synopsys Virtualizer tool and how simulations are performed in this environment.

5.1 Testbench structure

The test environment has been implemented as a *sc_module* connected to the DUT. All of the DUT pins are connected in the *sc_main* process to the testbench pins, and they are driven by the testbench itself.

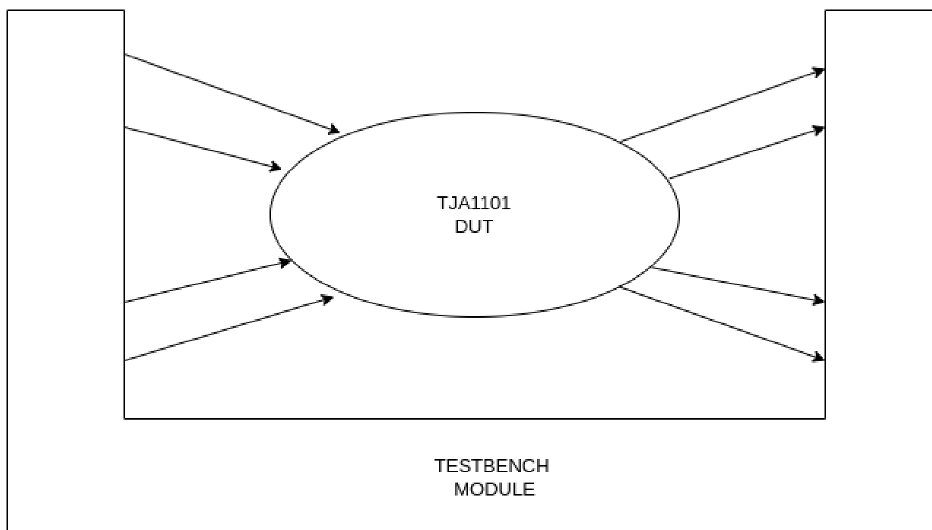


Figure 5.1: Test environment

In order to monitor the DUT internal variables, such as the Finite State Machine (FSM) state and the temperature, two further sockets are provided: each time the state has to be read and the temperature to be set, *b_ transport* socket functions are called to exchange data between the DUT and the testbench. This is how the tests are performed:

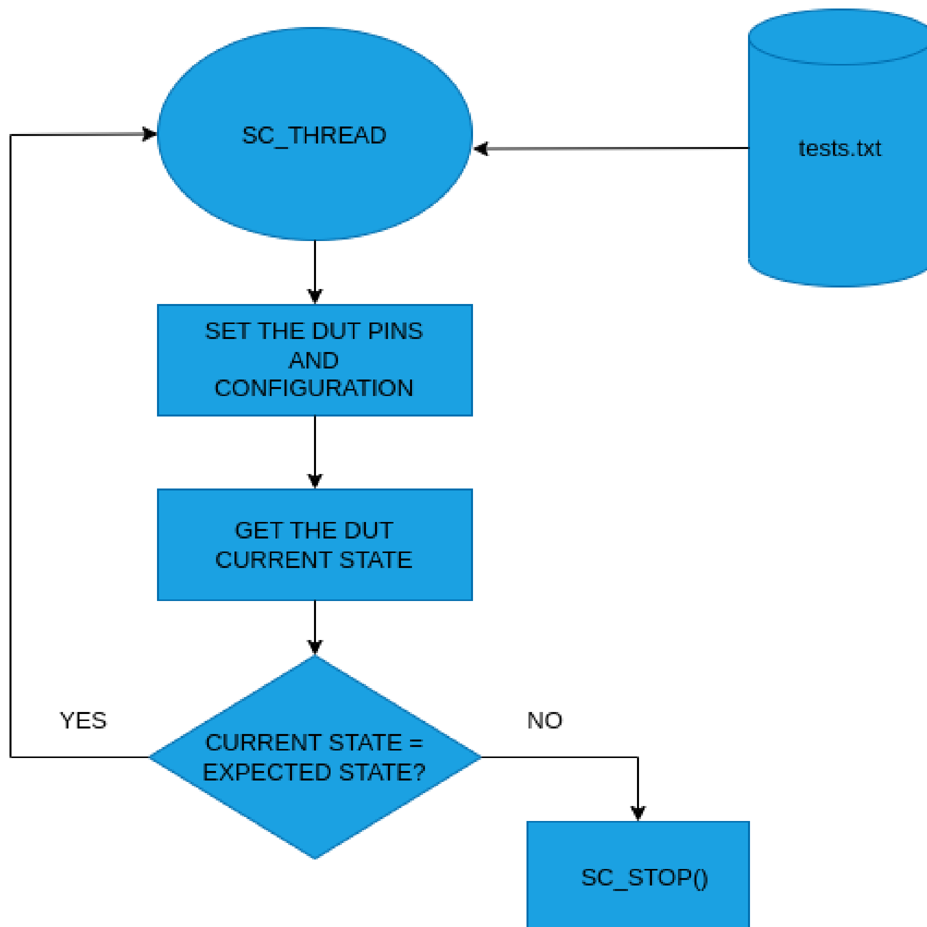


Figure 5.2: Test process flowchart

The SC_THREAD *main_ thread()* reads the test to be performed from a text file, and it sets all the variables to test whether the DUT behaves according to the requirements: it checks if the observed variable is equal to the expected one and if it is different an error message is printed on the terminal, and the simulation stops. To test different conditions, an sc_signal<unsigned int>called *n_ test* is incremented each time a test ends: if it is even some conditions are tested, if not other conditions are; *n_ test* is initialized in the *main_ thread()* to a certain value and it can be used as a seed to control the test type at each simulation. To trace the current state of the DUT on the waveform viewer, it is used a *sc_ bv* object: each of its element is the binary representation of the ASCII code correspondent to the current state, and it will be displayed as an ASCII character in the waveform viewer.

5.2 Test list

An exhaustive list of the test to be performed is here reported:

Test number	Transition to state
1	POWER OFF
2	STANDBY
3	RESET
4	STANDBY
5	DISABLE
6	STANDBY
7	NORMAL
8	STANDBY
9	SLEEP
10	STANDBY
11	NORMAL
12	SLEEP REQUEST
13	SILENT
14	SLEEP
15	STANDBY
16	NORMAL

Table 5.1: State transitions list

As mentioned before, to test different conditions for a state transition, the variable *n_test* can be initialized to a different value each time a new simulation is started. In the following analysis, it is shown all the required features and the state transition associated; finally, a general view of the tests listed above is reported together with an Ethernet Sniffer dump of the exchanged data.

5.3 Undervoltage detection

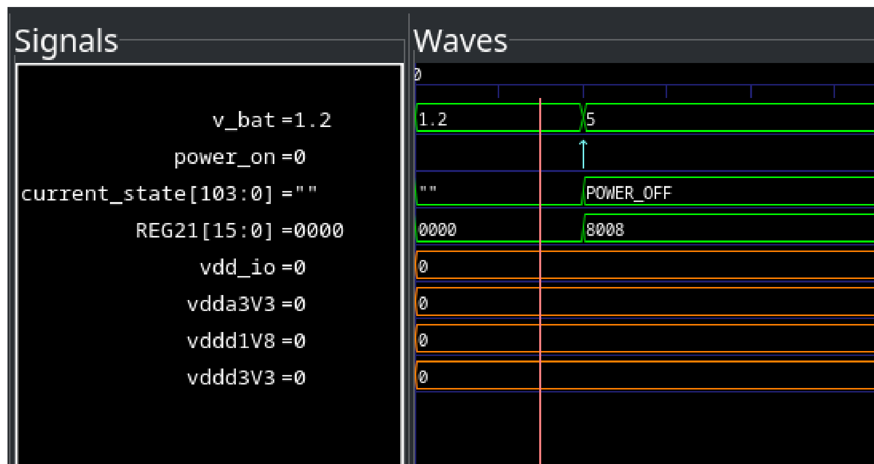


Figure 5.3: Undervoltage on V_bat

As it is possible to see, the value of V_bat is set to 1.2 such that the DUT enters POWER-OFF state, and when it is set to 5 it enters STANDBY state: the interrupt source register (REG21) PWON bit is set to 1; also UV_ERR bit is set to 1 because an undervoltage has been detected upon VDD_IO, VDDA3V3, VDDD3V3, VDDD1V8 pins.

5.4 Overtemperature and temperature warning detection

Figure 5.4: Overtemperature event

In this example, the value for the temperature is set to 200 °C and it causes a state transition from NORMAL to STANDBY, updates the interrupt source register (REG21) TEMP_ERR bitfield to 1 and the external status register (REG25) TEMP_HIGH bitfield to 1.

5.5 Interrupt event

Figure 5.5: Interrupt handling

In this example it is shown an interrupt handling: when POWER_ON event is triggered, REG21 is updated with the corresponding value and INT_N pin is pulled down; on the other hand, when a read operation is performed on REG21, INT_N is pulled high and the register is reset to its default value. As it is possible to see, all the events that could trigger an interrupt are reported and register 21 is update correctly.

5.6 Local/Remote wakeup handling

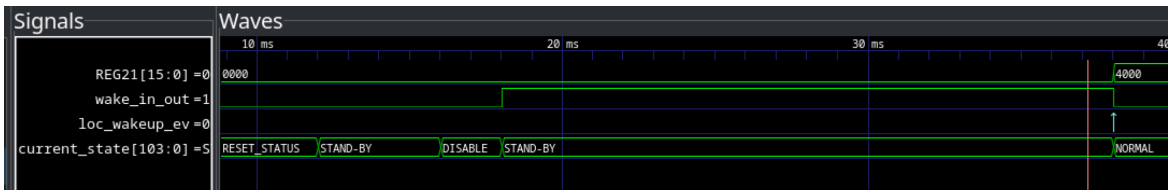


Figure 5.6: Local wakeup

Regarding the local wakeup, *wake_in_out* pin is pulled high to locally wake up the device: the event LOC_WAKEUP_EV is triggered 20ms after according to LOC_WU_TIM value (0) and it will cause a state transition from STANDBY to NORMAL.

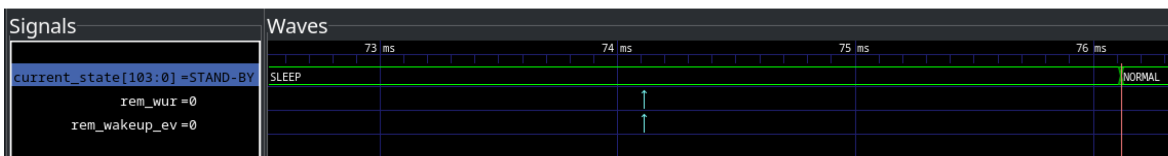


Figure 5.7: Remote wakeup

In this example, a remote wake-up request is detected at the MDI interface: the DUT is configured to react to a remote wake-up request, and it notifies the REM_WAKEUP_EV to allow the state transition from SLEEP to STANDBY.

5.7 LPS detection

This has been the most critical and challenging feature to test, and it is no exaggeration to assert that it is the most important aspect of the ASIC. This is because the LPS is an extremely valuable feature in automotive applications for power saving and is not implemented in standard Ethernet.
The hardware component required precise timing and specific command sequences to transition between sleep and wake states. Initially, understanding and correctly programming these sequences proved problematic.
Fortunately, the documentation is sufficiently comprehensive to facilitate the implementation of this key feature.
In order to address this issue, several steps have been strictly followed:

- In depth analysis into technical manuals to clarify every possible doubt
- Use of pen-and-paper approach to better describe the problem using flowcharts and to reorganize information concerning the correct timings
- Incremental testing and debugging for testing smaller parts of the mechanism step-by-step; this methodical process helped identify the specific points of failure
- Technical meeting with the project tutor for the consultation with an expertise

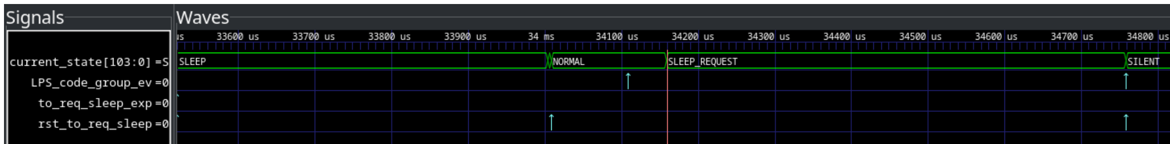


Figure 5.8: LPS code group detection

In Fig.5.8, a LPS code group is received while the DUT is in NORMAL state, and this will cause a state transition from NORMAL to SLEEP_REQUEST; after that, the TC10 handshake protocol is started: the DUT sends LPS code group in SILENT state to the testbench and receives another LPS code group back before the $t_{to(req)sleep}$ timer expires and it enters SLEEP state.

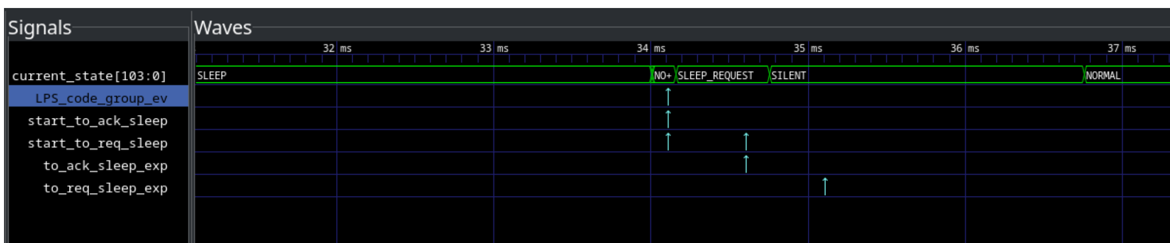


Figure 5.9: Timeout on $t_{to(req)sleep}$

In Fig.5.9, no LPS code confirmation is sent to the DUT and the timer $t_{to(req)sleep}$ expires: the DUT is in SILENT state when the timeout event occurs and it switches back to NORMAL state, as specified in the model requirement document.

Figure 5.10: Acknowledge timer behaviour

Fig.5.10 highlights the case when the DUT enters SLEEP_REQUEST state when it receives a LPS code from the link partner, and SLEEP_ACK timer is enabled: when it expires, the DUT completes the LPS handshake protocol and enters SLEEP state.

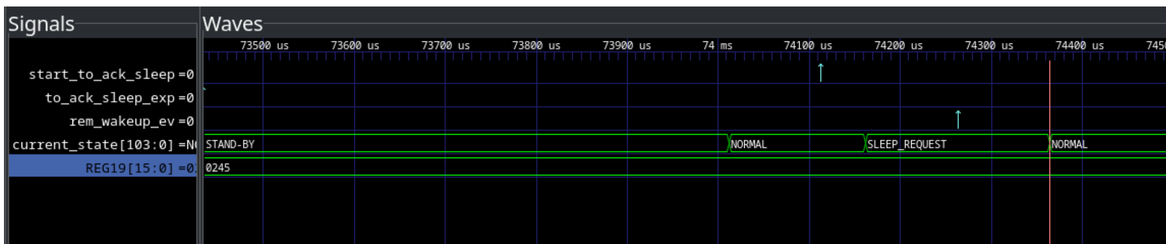


Figure 5.11: Transition to NORMAL with no $t_{to(ack)sleep}$ timeout

Finally, in Fig.5.11 it is shown that the DUT receives a remote wakeup request before the SLEEP_ACK timer expires: the timer is reset, and the device enters NORMAL state.
It is important to specify that in order to test these specific conditions ($t_{to(req)sleep}$ expiration and $t_{to(ack)sleep}$ no expiration) the test list was modified for sake of semplicity.

5.8 Results of listed tests

Finally, after the verification of the required features listed in the model requirements document, the final overview of the state transition table is here reported:

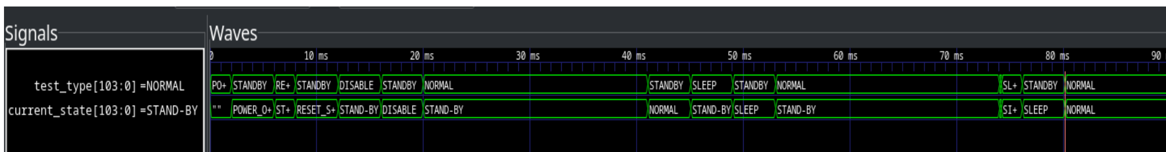


Figure 5.12: State tracing using waveform viewer

It is also shown the curson at a specific timestamp because, by just looking at the picture, the last transition wouldn't be right: the values in the *Signals* box show that there is an intermidiate transition (not visible because of the short amount of time).

5.9 Ethernet sniffer dumps

In this final section, dumps of the ethernet sniffers are reported to show the correctness of the behaviour by comparing the model frames with sniffed packets from Wireshark. The following dump is obtained with the model simulation by sending a TCP packet:

```
DATA_LINK_SNIFFER
FRAME_NUMBER  : 3
TIME_STAMP    : 6a72000
VENDOR_ID_SRC : 74366d
HOST_ID_SRC   : 94460
VENDOR_ID_DST : 7cc2c6
HOST_ID_DST   : 48103d
LT            : 800
CRC           : 35000000

IP_SNIFFER
FRAME_NUMBER   : 3
TIME_STAMP     : 6a72000
VERSION        : 4
HEADER_LENGTH  : 5
TOS            : 0
TOTAL_LENGTH   : 34
IDENTIFICATION : c69d
FLAGS          : 2
FRAGM_OFFSET   : 0
TTL            : f0
PROTOCOL       : 6
CHECKSUM       : 47bb
SRC_ADR        : 82c037f0
DST_ADR        : c0a80112
OPT            : 0

TCP_SNIFFER
FRAME_NUMBER     : 3
TIME_STAMP       : 6a72000
SOURCE_PORT      : 1bb
```

```
DESTINATION_PORT : 9a26
SEQUENCE_NUMBER  : 7cb038a3
ACK_NUMBER       : 44ae86bc
HEADER_LENGTH    : 8
RESERVED         : 0
CONTROL_FLAGS    : 10
WINDOW_SIZE      : 1000
CHECKSUM         : 1ecd
URGENT_POINTER   : 0
OPTIONS          : 1
                   1
                   1
                   8
                   a
                  ed
                  4b
                  3e
                  40
                  fc
                  61
                  86
                  f7

UDP_SNIFFER
FRAME_NUMBER     : 0
TIME_STAMP       : 0
SOURCE_PORT      : 0
DESTINATION_PORT : 0
DATAGRAM_LENGTH  : 0
CHECKSUM         : 0
```

The packet has been sniffed using Wireshark (only the most relevant fields have been shown for sake of readability):

```
Ethernet II, Src: Vodafone_09:44:60 (74:36:6d:09:44:60), Dst: TP-Link_48:10:3d (7c:c2:c6:48:10:3d)
  Destination: TP-Link_48:10:3d (7c:c2:c6:48:10:3d)
  Source: Vodafone_09:44:60 (74:36:6d:09:44:60)
  Type: IPv4 (0x0800)
Internet Protocol Version 4, Src: 130.192.55.240, Dst: 192.168.1.18
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
  Total Length: 52
  Identification: 0xc69d (50845)
  010. .... = Flags: 0x2, Don't fragment
  ...0 0000 0000 0000 = Fragment Offset: 0
  Time to Live: 240
  Protocol: TCP (6)
  Header Checksum: 0x47bb [correct]
  [Header checksum status: Good]
  [Calculated Checksum: 0x47bb]
  Source Address: 130.192.55.240
  Destination Address: 192.168.1.18
Transmission Control Protocol, Src Port: 443, Dst Port: 39462, Seq: 1, Ack: 1, Len: 0
  Source Port: 443
  Destination Port: 39462
  [Stream index: 1]
  [Conversation completeness: Incomplete, DATA (15)]
  [TCP Segment Len: 0]
  Sequence Number: 1    (relative sequence number)
  Sequence Number (raw): 2091923619
  [Next Sequence Number: 1    (relative sequence number)]
  Acknowledgment Number: 1    (relative ack number)
  Acknowledgment number (raw): 1152288444
  1000 .... = Header Length: 32 bytes (8)
  Flags: 0x010 (ACK)
  Window: 4096
  [Calculated window size: 262144]
  [Window size scaling factor: 64]
  Checksum: 0x1ecd [correct]
  [Checksum Status: Good]
  [Calculated Checksum: 0x1ecd]
  Urgent Pointer: 0
  Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
```

Figure 5.13: TCP packet sniffing using wireshark

UDP packet fields have been verified using the following frame:

```
DATA_LINK_SNIFFER
FRAME_NUMBER  : 5
TIME_STAMP    : 5eaa600
VENDOR_ID_SRC : 74366d
HOST_ID_SRC   : 94460
VENDOR_ID_DST : 7cc2c6
HOST_ID_DST   : 48103d
LT            : 800
CRC           : 86f73500
IP_SNIFFER
FRAME_NUMBER   : 5
TIME_STAMP     : 5eaa600
VERSION        : 4
HEADER_LENGTH  : 5
TOS            : 0
TOTAL_LENGTH   : 32
```

```
IDENTIFICATION : 0
FLAGS          : 2
FRAGM_OFFSET   : 0
TTL            : 38
PROTOCOL       : 11
CHECKSUM       : 3d5c
SRC_ADR        : 8efab4aa
DST_ADR        : c0a80112
OPT            : 0
TCP_SNIFFER
FRAME_NUMBER     : 0
TIME_STAMP       : 0
SOURCE_PORT      : 0
DESTINATION_PORT : 0
SEQUENCE_NUMBER  : 0
ACK_NUMBER       : 0
HEADER_LENGTH    : 0
RESERVED         : 0
CONTROL_FLAGS    : 0
WINDOW_SIZE      : 0
CHECKSUM         : 0
URGENT_POINTER   : 0
OPTIONS          : 0
                   0
                   0
                   0
                   0
                   0
                   0
                   0
                   0
                   0
                   0
                   0
                   0
UDP_SNIFFER
FRAME_NUMBER     : 5
TIME_STAMP       : 5eaa600
SOURCE_PORT      : 1bb
DESTINATION_PORT : a97e
DATAGRAM_LENGTH  : 1e
```

```
CHECKSUM            : aeb7
```

This is the sniffed packet using Wireshark:

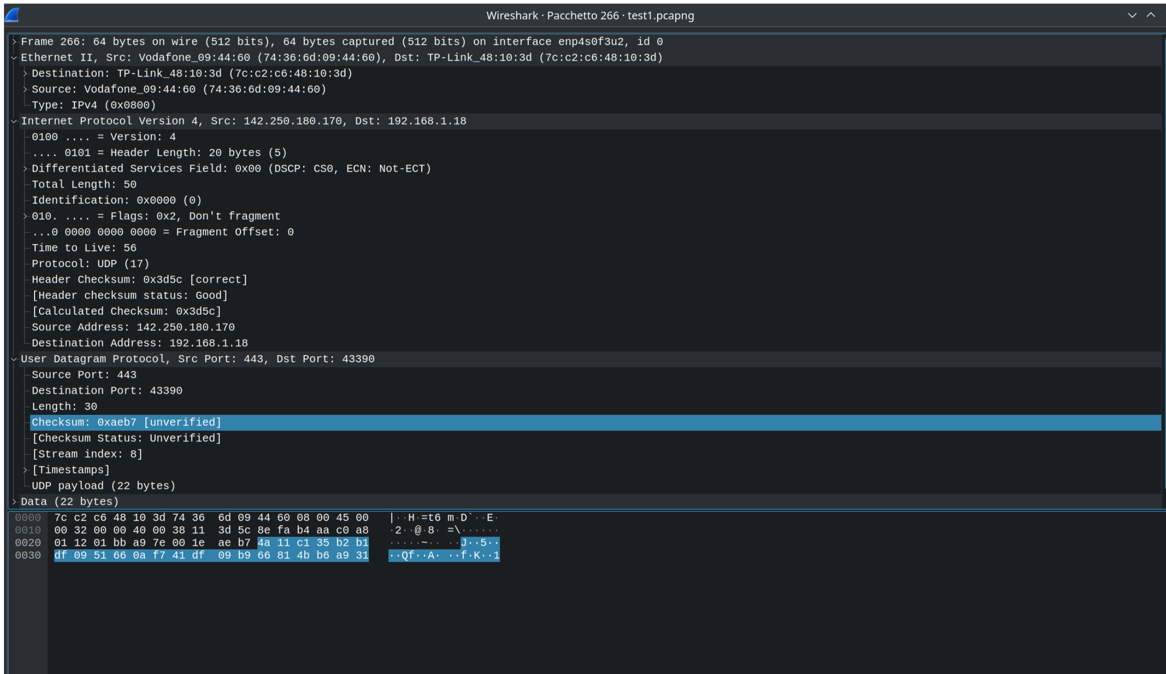


Figure 5.14: UDP packet sniffing using wireshark

5.10 Code coverage

In this section it is reported the code coverage percentage of the model: this is an important test phase in which the designer can see how many code lines are hit during the execution of the simulation, and so the testbench can be improved in order to verify all the requirements. Code coverage has been obtained using the *coverage* flag in the compilation and converted in *.html* files using *genhtml* command, in order to have a visual representation.

LCOV - code coverage report

Current view: top level - /home/LorisPanaro/Documenti/thesis/TJA1101B/src
Test: coverage.info
Test Date: 2023-09-01 12:34:23

	Coverage	Total	Hit
Lines:	82.1 %	1657	1360
Functions:	78.8 %	156	123

Filename	Line Coverage			Function Coverage		
	Rate	Total	Hit	Rate	Total	Hit
ETHERNET_FRAME.cpp	70.1 %	552	387	75.3 %	89	67
PHY.cpp	89.0 %	145	129	88.9 %	9	8
SMI_registers.cpp	95.2 %	331	315	73.3 %	15	11
SNIFFER_DATA_LINK.cpp	90.3 %	31	28	80.0 %	5	4
SNIFFER_IP.cpp	93.3 %	45	42	80.0 %	5	4
SNIFFER_TCP.cpp	93.2 %	44	41	80.0 %	5	4
SNIFFER_UDP.cpp	70.4 %	27	19	80.0 %	5	4
TJA1101.cpp	82.8 %	482	399	91.3 %	23	21

Figure 5.15: Code coverage representation

In Fig.5.15 it is shown how the visualization of the code coverage appears only for the DUT (and not the testbench): it has been used a value for *n_seed* equal to 0, so the total percentage cannot be 100% because not all the conditions are tested, but for what regards *SMI_registers*, *PHY* and *TJA1101* classes it has been reached a good coverage; on the other hand, *ETHERNET_FRAME* class has a low coverage because the testbech does not use UDP packets (and so all the related functions are not hit).

5.11 Functional verification

During the developing of a real hardware ASIC, functional verification plays a crucial role. Detecting a hardware bug after production can result in significant financial losses due to the necessity of redeveloping silicon masks. To mitigate this issue, functional verification is employed. The primary goal of functional verification is to identify and correct design errors by applying different methods:

- Logic simulation: it aims to simulate separate logical structures that make up a functional unit by performing logic simulation before the functional unit is built
- Emulation: by applying emulation, the ASIC is built on top of a Field Programmable Gate Array (FPGA) or a programmable logic device; this method is expensive and slow, but it is much faster than simulation and makes possible to boot up certain software programs
- Formal verification: it uses mathematical expressions to check the logic design; through mathematics, it is possible to prove that specific requirements are met in the design; in addition to that, this method checks that deadlocks do not happen in the design

During the developing of the project, it has been decided with the thesis tutor to not perform functional verification for several reasons:

1. The effort that could have been required for a complete functional verification was too high for the model requirement; ensuring that every bit and logic switching combination behaves correctly would have been excessively time-consuming due to the absence of any commercial tools capable of facilitating comprehensive functional verification
2. Even if a bug will be found in the design, the effort to fix it relies on a simple bug fixing in the source code and a recompilation of the model

In this perspective, only code coverage can be used as a metric to measure the quality of both the model and the testbench.
Unfortunately, the PEP firmware does not include Ethernet drivers. Consequently, the

TJA1101 model cannot be tested within a real software environment. To date, testing has been conducted without dedicated test software, relying solely on the custom testbench outlined in this chapter.

5.12 Live sniffing demostration

When the constructor of the testbench object is called, aN ethernet socket is opened; it is of type *SOCK_RAW* because it is necessary to sniff all the packets detected in the network, even those without data payload (i.e. TCP packets used for synchronization), the domain has been set to *AF_PACKET* such that it is possible to extract information regarding physical connection (MAC addresses and length/type field) and the protocol type has been set to *ETH_P_IP* to capture IP frames.

```
while(1) {
    eth_frame_MII_tb.reset_payload();
    _data_size = recvfrom(_sockfd_raw, eth_frame_MII_tb.get_payload(),
  ETH_PAYLOAD_MAX_SIZE, 0, &_server_addr, &_server_addr_len);
    if(_data_size < 0) {
        perror("recvfrom() failed\n");
        exit(EXIT_FAILURE);
    }
    _net_protocol = eth_frame_MII_tb.IP_get_protocol();
    if(_net_protocol == UDP_CODE || _net_protocol == TCP_CODE) {
        trans.set_data_length(eth_frame_MII_tb.get_payload_length());
        trans.set_streaming_width(eth_frame_MII_tb.get_payload_length());
        client_socket_MII_tb->b_transport(trans, b_trans_del);
        wait(t_wait);
    }
    std::cout << std::endl;
}
```

Packets are copied in the *ETHERNET_FRAME* object using the blocking *recvfrom()* API and, if the protocol type is either TCP or UDP, it is sent to the virtual device from the testbench using the *b_transport()* function.
These frames are real time compared to packets sniffed using Wireshark by launching the simulation together with the software without opening web pages or other programs (otherwise there will be too much network traffic, making a manual comparison difficult).

5.13 Virtual Development Kit implementation

After having completed the SystemC model, the very next step of the project is to build the model in the Synopsys VDK tool[30], which is the Punch tool used to create virtual ECUs. The Virtualizer allows the model to be wrapped into a python environment

in which the user can change the parameters, the values of registers, and add faults even at run time. Virtual prototypes offer a distinct advantage over physical hardware by granting comprehensive visibility and control throughout the entire system. This encompasses cores, interconnects, and peripherals, ending in an expedited and highly effective edit-compile-debug process. Moreover, these prototypes ensure deterministic system execution and facilitate non-intrusive debugging[12]. The python environment is created using Corba tool. When the user imports the C++ code into the Virtualizer, the tool generates specific description files (xml and json) to be correctly interpreted, and recompiles the code using Visual Studio Professional. Before starting the simulation on the Virtualizer, the tool offers several analysis settings that can be useful for debugging purposes, such as:

- functional coverage port monitor
- process trace
- register trace
- socket trace
- TLM port trace
- transaction contention statistics
- transaction count
- transaction field statistics
- transaction latency statistics
- value trace

Each of these possibilities can be selected before running the simulation, as shown in Fig.5.16:

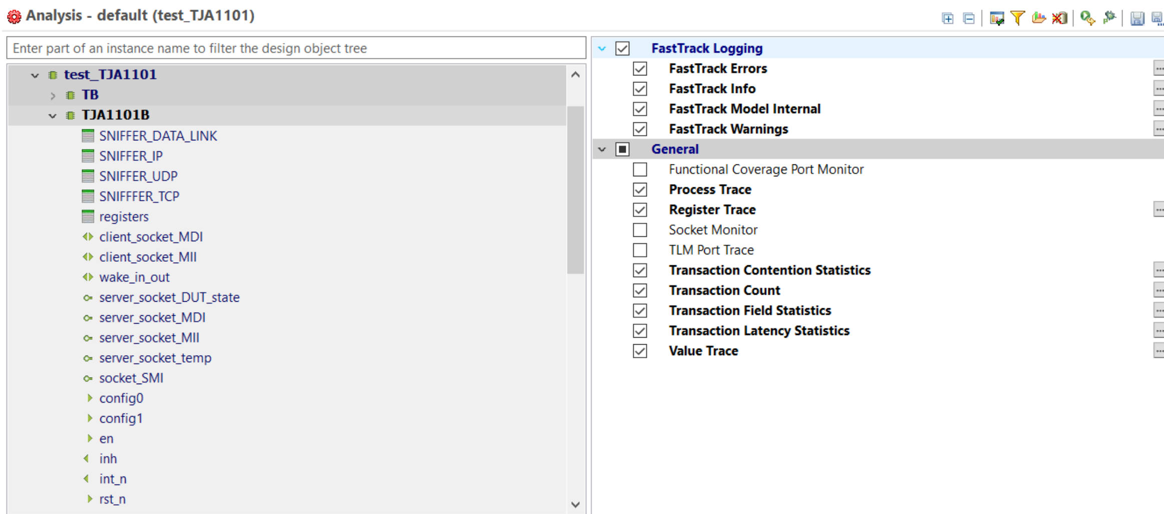


Figure 5.16: Virtualizer analysis settings

After having configured analysis settings, the tool has the capability of using breakpoints at a specific simulation time and to analyze register contents, signal values and payload data.

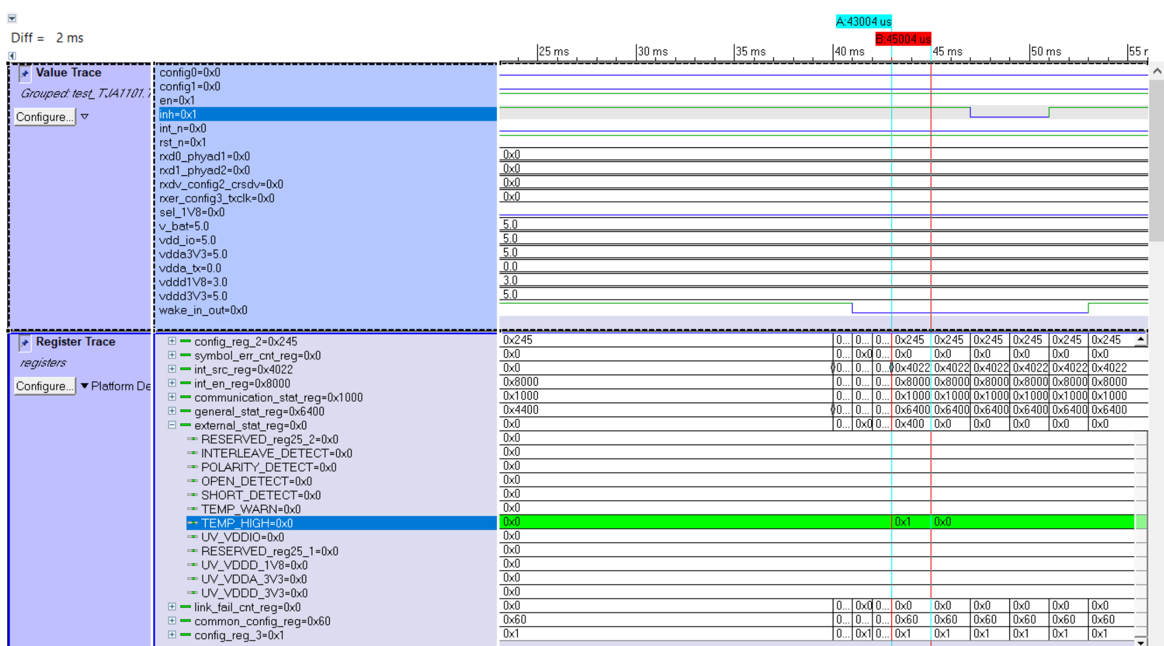


Figure 5.17: Pin and SMI registers tracing

In Fig.5.17 a segment of simulation is being presented. In this example, an overtemperature event has occurred (temperature value is not shown in the figure): the value of the external status register has changed properly (TEMP_HIGH bitfield set to 1) and when the temperature goes down the threshold value, the bitfield value is reset to 0. As it possible to see, all pins are traced by the Virtualizer and so the SMI registers.

An additional capability of the tool is its ability to display the pinout of the device model together with the sockets involved in the simulation:

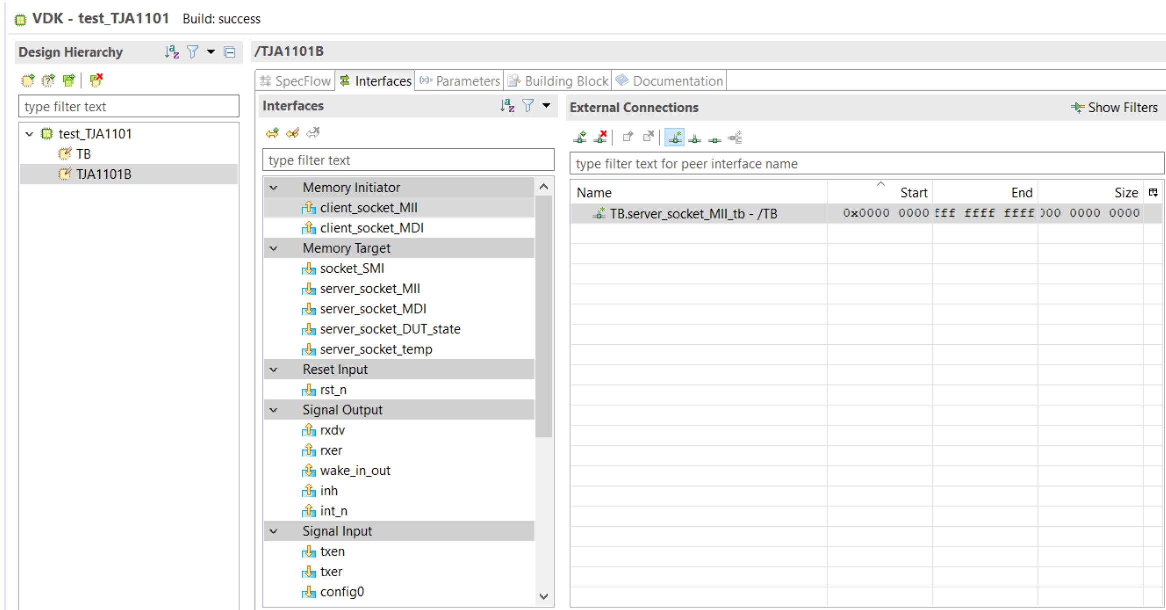


Figure 5.18: Pins and sockets interface

In Fig.5.18 it is possible to see all the sockets of the model:

- client and server sockets for the MII interface
- client and server sockets for the MDI interface
- socket for SMI interface
- socket for the current state variable
- socket for the temperature value

CHAPTER 6

Final conclusions

In conclusion, the adoption of automotive Ethernet marks a significant paradigm shift in the design, development, and functionality of modern vehicles. This research has given a general overview about Automotive Ethernet, highlighting its benefits and challenges. In addition to that, this thesis aimed to provide a comprehensive understanding of the role of Automotive Ethernet in shaping the future of mobility and how DT technology is able to provide a significant value to its development and improvement. The natural prosecution of this work is therefore the verification and validation of the virtual model in an integrated system composed by a microcontroller and the TJA1101B device together with the official software developed by Punch Softronix, because, as well explained in chapter 5, the SystemC model of TJA1101B device has been tested by means of a TLM testbench and not with a real software: it has not been possible verifying the correct behaviour and interaction between the DUT and the rest of the PEP environment. In conclusion, the final results will have to be validated with the physical counterpart of the DT, which is the physical ECU.

Bibliography

[1] Ali Abaye. "BroadR-Reach® Technology: Enabling one pair ethernet". In: *Broadcom Corp* (2012).

[2] *Advantages of 4D-PAM5 line coding and disadvantages of 4D-PAM5 line coding*. URL: https://www.rfwireless-world.com/Terminology/Advantages-and-Disadvantages-of-4D-PAM5-line-coding.html.

[3] *AUTOSAR*. URL: https://www.autosar.org/.

[4] Philip Axer, Charles Hong, and Antony Liu. "OPEN Sleep/Wake-up Specification". Version 2.0. In: *OPEN Alliance* (2017).

[5] John Aynsley. "New Features of IEEE Std 1666-2011 SystemC". In: *Accellera Systems Initiative* (2012).

[6] John Aynsley and Doulos. "The Transaction Level Modeling standard of the Open SystemC Initiative (OSCI)". In: *OSCI TLM-2.0* (2009).

[7] Martin Barnasconi. "SystemC and Digital Twin: good match or not?" In: *SystemC Evolution day* (2019).

[8] Davide Bollati, Bernd Körber, and Michael Kaindl. "1000BASE-T1 System Implementation Specification". Version 1.6. In: *1000BASE-T1* (2022).

[9] *BroadR Reach vs 100Base-Tx vs 1000BASE-T; Difference between BroadR Reach, 100Base-Tx, 1000Base-T*. URL: https://www.rfwireless-world.com/Terminology/BroadR-Reach-vs-1000Base-T-vs-100Base-Tx.html.

[10] Stefan Buntz, Bernd Körber, and David Bollati. "100BASE-T1 System Implementation Specification". In: *100BASE-T1* (2017).

[11] Steven B. Carlson et al. "IEEE 802 Ethernet Networks for Automotive". In: *IEEE 802 Plenary Tutorial* (2017).

[12] Synopsys Group. "Platform Architect and Virtualizer Introduction". In: *Verification continuum* (2022).

[13] Synopsys Group. "SystemC Modelling Library manual". In: *Synopsys verification* (2022).

[14] IEEE. "IEEE Standard for Ethernet Amendment 4: Physical Layer Specifications and Management Parameters for 1 Gb/s Operation over a Single Twisted-Pair Copper Cable". In: *IEEE Std 802.3bp-2016 (Amendment to IEEE Std 802.3-2015 as amended by IEEE Std 802.3bw-2015, IEEE Std 802.3by-2016, and IEEE Std 802.3bq-2016)* (2016), pp. 1–211. DOI: {10.1109/IEEESTD.2016.7564011}.

[15] IEEE. "IEEE Standard for Standard SystemC Language Reference Manual - Redline". In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) - Redline* (2012), pp. 1–1163.

[16] *Infineon Aurix TC399X microcontroller*. URL: https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/.

[17] Open SystemC Initiative. "Draft standard SystemC language reference manual". In: *April 25th* (2005).

[18] Michael Kaindl. "Advanced diagnostic features for automotive Ethernet PHYs". Version 2.2. In: *1000BASE-T1* (2022).

[19] Bernd Körber. "Definitions for Communication Channel". In: *OPEN Alliance* (2017).

[20] Yadong Li et al. "Research based on OSI model". In: *2011 IEEE 3rd International Conference on Communication Software and Networks*. 2011, pp. 554–557. DOI: 10.1109/ICCSN.2011.6014631.

[21] Megha Nanda. *9 Amazing Examples of Digital Twin Technologies for Industries*. 2013. URL: https://www.toobler.com/blog/digital-twin-examples.

[22] NXP. "TJA1101B 100BASE-T1 PHY for automotive Ethernet". In: *Product datasheet* (2021).

[23] NXP. "TJA1101B 100BASE-T1 PHY for automotive Ethernet". In: *Product Application Note* (2021).

[24] Open SystemC Initiative. *Concurrency*. URL: https://learnsystemc.com/basic/concurrency.

[25] Open SystemC Initiative. *SystemC Version 2.0 User's Guide*. 2001. URL: http://www.systemc.org.

[26] OSCI language working group. "SystemC 2.1 overview". In: *SystemC language* (2004).

[27] Donovan Porter. "100BASE-T1 Ethernet: the evolution of automotive networking". In: *Texas Instruments, Techn. Ber* (2018), p. 2.

[28] *Punch Electronic Platform*. URL: https://punchsoftronix.com/punch-electronic-platform/.

[29] Stuart Swan. "An Introduction to System Level Modeling in SystemC 2.0". In: *Cadence Design Systems* (2001).

[30] *Virtualizer Development Kits (VDKs)*. URL: https://www.synopsys.com/verification/virtual-prototyping/vdk.html.

[31] Lane Warshaw and Aaron Parrott. *Industry 4.0 and the digital twin*. URL: https://www.google.com/search?q=Industry+4.0+and+the+digital+twin+deloitte&oq=Industry+4.0+and+the+digital+twin+deloitte&aqs=chrome..69i57j69i60l2.1918932j0j15&sourceid=chrome&ie=UTF-8.

[32] Meng Zhang et al. "Digital twin data: methods and key technologies". In: *Digital Twin* 1 (Sept. 2021), p. 2. DOI: 10.12688/digitaltwin.17467.1.