Alma Mater Studiorum · University of Bologna

School of Science
Department of Physics and Astronomy
Master Degree in Physics

Development of a PicoTDC-based card for the ALICE TOF detector

Supervisor:
Dr. Davide Falchieri

Co-supervisor:
Dr. Pietro Antonioli

Submitted by:
Sandro Geminiani

Academic Year 2023/2024

To my beloved family.

Abstract

The readout electronic system related to the ALICE TOF (Time-Of-Flight) uses the TRM (TDC readout module) card, based on the HPTDC (High-Performance TDC) ASICs, as the main element to provide the time measurements of a particle crossing the TOF detector.

This master thesis is linked to the wider project of a new TRM2 board, begun since the TRM board components are now out of production and the TOF readout system must be reliable during the current and the next LHC Runs. The thesis focuses on the development of a test board hosting a PolarFire FPGA and two PicoTDC ASICs, which are the main components selected for the TRM2 development. The work done for the thesis regards the control and test of the PicoTDCs configuration and readout, through the design of the FPGA firmware architecture and the related software routines.

As the board supports Ethernet connection with the FPGA, the firmware and software were built considering the IPbus protocol, which communicates over the Ethernet protocol exploiting the UDP/IP model. The IPbus is implemented on the FPGA as a master-slaves structure, in which each slave is identified using a 32-bit address and communicates with the master through a 32-bit data bus. The IPbus slaves implement different hardware features that can be controlled through IPbus data transfers.

The designed firmware relies on a structure supporting a 1Gb/s Ethernet communication able to control the on-chip IPbus, which includes up to 12 IPbus slaves to manage the PicoTDCs and board features. Then, dedicated software front-end libraries were built to manage the operations of each slave, using the μHAL IPbus back-end library.

The whole system was used to test the setup for a PicoTDC resolution measurement, employing a programmable delay line with a 0.5 ps resolution. The developed firmware and software proved reliable during all 36 hours of measurement, providing 18 datasets of 10^7 events each. Furthermore, the analysis showed excellent compatibility between the measured time differences and the configured time delays. Finally, a single channel resolution of (2.95 ± 0.43) ps was estimated for both the PicoTDC ASICs.

Contents

Introduction

The Run 3 phase of LHC (Large Hadron Collider) at CERN started in July 2022, reaching new records center of mass energy for proton-proton, proton-ion, and ion-ion collisions. This new data-taking phase concludes three years of updating and maintenance work for the collider and the 4 experiments, located along the accelerator. The ALICE (A Large Ion Collider Experiment) experiment is located at point n.2 of the LHC and was built to study particles' strong interaction and the QGP (Quark-Gluon Plasma). To cope with the higher luminosity and interaction rate, the ALICE upgrade considered some sub-detectors overhaul and the restyling of the readout system to supply a continuous readout.

The ALICE Time-Of-Flight (TOF) detector was built to perform particle identification within an intermediate momentum range. During LHC Run 3, only the TOF detector electronic readout system was modified to follow the continuous readout direction of the whole experiment. Such an upgrade was completed while keeping the main readout board used in previous LHC Runs: the TRM (TDC Readout Module).

In detail, the TRM is a VME slave card and it hosts 30 HPTDC (High-Performance TDC) ASICs able to perform time digitization of the front-end signals, related to a particle crossing an MRPC (Multi-gap Resistive Plate Chamber). The board uses an FPGA to manage the readout workflow and the VME interface. Since the TRM components described are out of production and their maintenance is increasingly problematic, a project for a new TRM2 card started.

This thesis work focuses on the development of a test board used to characterize the PicoTDC ASIC and the PolarFire FPGA, selected as the new TDC and FPGA that will be implemented on the TRM2 board. The hardware layout was designed by the INFN electronics laboratory and the ALICE group of Bologna, while my work consisted of developing the firmware architecture and the related software suites used for the configuration and readout of both the integrated TDCs. Since the PicoTDC board supports Ethernet connection, the software and firmware projects were designed considering the IPbus communication protocol, built over Ethernet protocol.

The thesis's first chapter includes an introduction to QGP (Quark-Gluon Plasma) physics and a brief description of the ALICE layout, considering the last upgrades for Run 3. This chapter focuses on the TOF detector, emphasizing its readout system. In the end, a description of the TRM board and some considerations about the new TRM2 board are provided.

The following four chapters describe the work done for the development of the PicoTDC test board. Specifically, the second chapter shows the test board layout, focusing on the Ethernet subsystem. Then an introduction to the IPbus protocol is also provided at firmware and software levels.

Chapters 3 and 4 describe the implemented firmware architecture and the related software suites. Software and firmware are designed to control the board features and

the PicoTDC operations.

Finally, chapter 5 shows the setup used to perform a time resolution measurement, employing 2 input channels for each PicoTDC. This was done to test the readout and configuration system reliability and to estimate the effective time resolution of the PicoTDCs.

Chapter 1

The ALICE experiment and its TOF detector

ALICE is a general-purpose detector focused on the study of QCD (Quantum Chromodynamics), which is the theory of strong interactions in the Standard Model. ALICE was designed to support analysis for heavy-ion collision physics, which generates the requested conditions to study the Quark-Gluon Plasma.

This detector provides a precise tracking system and a sophisticated Particle Identification (PID) setup covering a wide momentum range (0.1 to 100 GeV/c), to cope with the high particle multiplicities produced in heavy-ion collision. During LHC Run 2 (2015-2018) ALICE accumulated an integrated luminosity of $\sim$0.4 nb^{-1} for Pb-Pb interactions at the center of mass energy $\sqrt{s} = 5.02$. Furthermore, p-p and p-Pb collisions were also investigated considering respectively energy up to $\sqrt{s} = 13$ TeV and $\sqrt{s} = 8.16$ TeV [1]. Up to now during LHC Run 3, ALICE collected data considering energies $\sqrt{s} = 13.6$ TeV and $\sqrt{s} = 5.36$ TeV respectively for p-p and Pb-Pb collisions [2].

The TOF system is one of the ALICE PID sub-detectors, located at 3.7 m from the beam axis. It is used for particle identification in the intermediate momentum range and covers all the ALICE barrel for time-of-flight measurements.

The chapter introduces the QCD and QGP physics plus provides a brief overview of the QGP probes studied at the ALICE experiment. The following sections explain the ALICE current layout focusing on the TOF detector specifications and performance. Then a complete explanation of the TOF readout chain is provided, aiming to introduce the main element of the readout electronic system: the TRM card (TDC Readout Module). In the end, an upgrade of the TRM board is discussed, considering new generation components, explained in the next chapters.

1.1 Heavy-ion collision physics

Ordinary nuclear matter is made up of protons and neutrons that are composed in their internal structure by partons: quarks and gluons. Quantum Chromodynamics is the theory within the framework of the Standard Model built to describe the strong interactions between partons and used to explain the formation of hadrons, such as neutrons and protons. Therefore, QCD mainly enters into the description of the Quark Gluon Plasma which is a deconfined state of quarks and gluons that is thought to expand in the early stage of the Universe at $\sim 10^{-6}$ s after the Big Bang.

The QGP has been studied since the 70s [3] using heavy-ion collision, as they ensure, at relativistic energies, the best conditions of particle density and temperature. The first indirect result of a new state of matter was announced by CERN in 2000, providing at SPS[1] heavy-ion collisions with beam energy up to 40 GeV per nucleon [4]. Centers of mass energy $\sqrt{s} \approx 200$ GeV were reached in the following facilities at RICH[2], finding new results at higher temperatures for QGP studies. New record collision energy is reached at the LHC in Geneva, in which the ALICE experiment is located providing a specific layout able to perform QGP analysis for heavy-ion collision at energy up to $\sqrt{s} = 5.5$ TeV.

1.1.1 Introduction to Quantum Chromodynamics (QCD)

The first step to the historical introduction of QCD was the proposal of a *color* quantum number that explains the internal structure of hadrons, defined in the quark model as:

- mesons for quark anti-quark states ($\bar{q}q$),
- baryons for three-quark states (qqq).

This quantum number is related to a conserved color charge that can assume only three values by experimental observations: red, blue and green. Therefore quarks and anti-quarks carry respectively color charge and anti-color charge. This new quantum number was proposed to explain the presence in nature of 3/2-spin baryons composed of the same flavor quarks as:

$$\Omega^- = |s_\uparrow s_\uparrow s_\uparrow\rangle \otimes |\psi_{color}\rangle \tag{1.1}$$

The $|\psi_{color}\rangle$ can be seen as an antisymmetric superposition of three quarks wavefunctions, including all three possible color charges. $|\psi_{color}\rangle$ state is added to preserve the Pauli exclusion principle[3], as the $|s_\uparrow s_\uparrow s_\uparrow\rangle$ is symmetric under the position exchange of two quarks. As the color charges contribute equally, the hadron state seen in Equation 1.1 is defined as colorless [5]. In the end, the theory asserts color-singlet wavefunction states for both baryons and mesons explaining why no colored states have ever been detected as free particles.

This whole mechanism is explained by QCD, which is a gauge field theory based on the symmetry group $SU(3)_c$[4] and used in the Standard Model to describe strong interactions between quarks. Specifically, each quark is associated with a matter field spinor defined by three different color fields: q_{red}, q_{blue} and q_{green}. The spinors interact with each other using a massless vector gauge field called gluon, which provides eight different colored states considering the $SU(3)_c$ adjoint representation [5]. This interaction mechanism is built to preserve locally the $SU(3)_c$ non-abelian symmetry and to conserve the color charge. The Lagrangian density of QCD is:

$$L^{QCD} = -\frac{1}{4} F^a_{\mu\nu} F_a^{\mu\nu} + \sum_f \bar{q}^f_\alpha (i\gamma_\mu D^\mu)^{\alpha\beta} q^f_\beta - \sum_f m_f \bar{q}^f_\alpha q^{\alpha f}, \tag{1.2}$$

[1] Super Proton Synchrotron.

[2] Realtivistic Heavy Ion Collider at BNL.

[3] It asserts that anti-symmetric wavefunctions must define particles with half-integer spin. Therefore in this case the baryon wavefunction must be anti-symmetric under the exchange of any two quarks.

[4] The c subscript states for the conserved *color* charge.

in which $F_a^{\mu\nu}$ is the strength field vector defined as:

$$F^a_{\mu\nu} = (\partial_u G^a_\nu - \partial_\nu G^a_\mu - g_s f^a_{bc} G^b_\mu G^c_\nu) \tag{1.3}$$

and D^μ is the covariant gauge derivative:

$$D^\mu = \partial^\mu + i g_s \frac{\lambda_a}{2} G^\mu_a. \tag{1.4}$$

Here the G^a represents the gluon field with state a (a = 1, ..., 8), while the spinor q^f_α is the quark with a color α (α = red, blue, green) and a flavor f. The λ_a denotes the Gell-Mann matrices which are the generators of the $SU(3)_c$ group[5], while g_s is the dimensionless coupling constant of the strong interactions. Considering the quark masses m_f, a Lagrangian quark mass term is also defined.

In the QGP description, two important concepts of QCD were considered: the *asymptotic freedom* and the *confinement*. Since the QCD works as a perturbative theory for weak values of g_s coupling, $\alpha_s = \frac{g_s^2}{4\pi}$ [5] must be investigated by taking an analogy with the fine structure constant of QED[6]. The α_s value must be estimated using amplitude computation of Feynman diagrams. At higher transferred momentum Q, loops like the

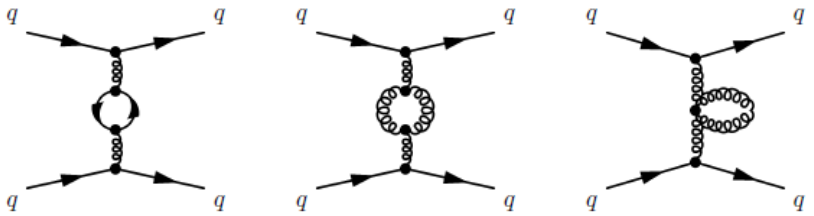


Figure 1.1: One loop diagrams for qq interactions in QCD [5].

ones in Figure 1.1 must be considered among all the diagrams. Therefore, divergent terms appear in the amplitude computations and the renormalization method must be applied to absorb these terms into the estimation of α_s, causing the running of the coupling [6]. As QCD is a not-abelian theory, gluon mediators are color-charged allowing gluon self-interaction Feynman diagrams. As shown in Figure 1.1, QCD theory allows two loop types that imply a $\bar{q}q$ couple or gluons. These two provide different contributions for the estimation of α_s, generating the following equation for the coupling as a function of Q:

$$\alpha_s(Q) = \frac{1}{B\ ln(Q^2/\Lambda^2_{QCD})}, \tag{1.5}$$

in which B is a positive quantity and the Λ ($\simeq$ 200 MeV) is an intrinsic energy scale of the strong interaction.

The plot of Figure 1.2 mostly reproduces Equation 1.5 and at large Q values the $\alpha_s \to 0$. This effect is called *asymptotic freedom* and shows that quarks behave as free at short distances (high energy), allowing for perturbative QCD calculations. At large distances or low transferred Q momentum, the α_s becomes larger and below $Q \sim \Lambda_{QCD}$ the QCD gauge theory is no more consistent, as it works as a perturbative theory. This α_s behavior gives strong hints for *confinement* effect explanation, implying that only color-singlet states, as hadrons, can propagate as free particles. In fact, at large distances, quarks show the largest coupling and are more inclined to interact generating hadrons [6].

[5]For a $SU(N)$ group the total generators are $N^2 - 1$, so 8 generators are considered.

[6]$\alpha = \frac{e^2}{4\pi}$, in which e is the electric charge unit.

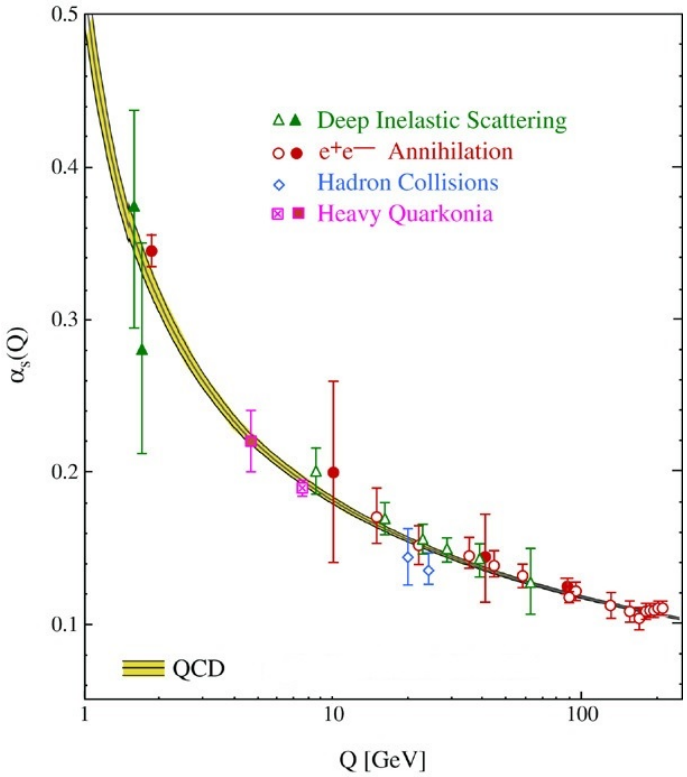


Figure 1.2: Running coupling constant α_s as a function of the exchanged momentum Q [7].

1.1.2 Quark-Gluon Plasma (QGP) overwiew

As mentioned, the QCD asserts that quarks and gluons behave as deconfined states at high energy levels, and at lower energy regimes they can not propagate as free particles forming the hadrons. The quark-gluon plasma is a colored matter state consisting of deconfined quarks and gluons, provided at extremely high temperature and/or density conditions. A QCD thermodynamic approach can be used to explain the QGP formation as a hadronic matter phase transition. Figure 1.3 shows the QCD phase diagram, defined by the temperature T and the baryon chemical potential μ_B. The μ_B variable is related to the net baryon density and is the system energy U deviation caused by increasing the N_B baryon quantum number by one unit:

$$\mu_B = \frac{\partial U}{\partial N_B}. \tag{1.6}$$

in which $dU = TdS - PdV + \mu_B dN_B$ considers all the classical thermodynamic variables.

In equilibrium thermodynamics, the phase transitions of a system are classified by exploiting the free energy function $F = U - TS$, defined using the internal energy U and the entropy S. As shown in Figure 1.3, the transition between the hadrons gas and QGP behaves mainly as a first-order transition, indicating a discontinuity of $\frac{\partial F}{\partial T}$ that identifies an abrupt change of matter phase. A critical point[7] is defined by specific values $[T_{crit}, \mu_B^{crit}]$, in which the QGP and hadrons gas phases coexist. Above this point on the boundary path, the transition becomes a crossover [9], indicating no discontinuities in all F derivative orders, and provides a smooth phase change.

[7]It is a second-order critical point [9].

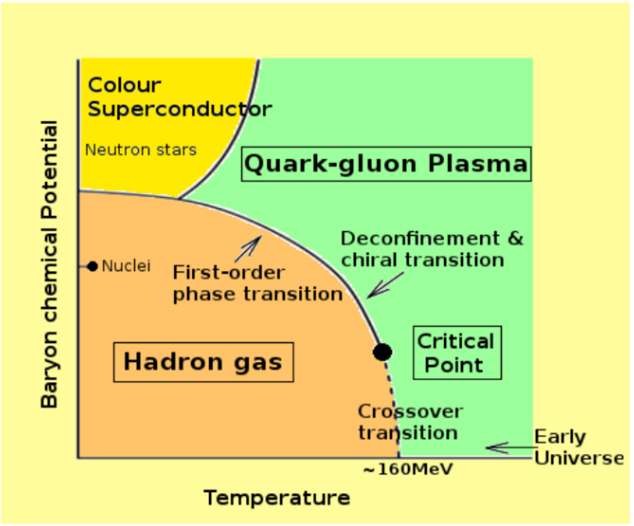


Figure 1.3: QCD phase diagram [8].

At high baryonic potential and temperatures close to zero, the matter behaves as a color superconductor in which the deconfined quarks tend to form couples with specific color combinations. This behavior is thought to describe the internal structure of neutron stars. Instead, the crossover region, for $\mu_B \to 0$ and $T > T_{crit}$, represents the early universe conditions for the QGP phase transition.

In the statistical mechanics framework, the F free energy of a particles system is defined by the Z partition function as follows[8]:

$$F(V,T) = -T\ ln(Z(V,T)), \tag{1.7}$$

in which Z definition involves directly the Lagrangian describing the dynamic of the system [10]. As mentioned in subsection 1.1.1, the QCD gauge field theory is consistent in the asymptotic freedom regime, so an analytic approach using L^{QCD} is allowed only at high energies. To investigate hadronic matter phase transition, energy regimes among $\Lambda_{QCD} \sim 200$ MeV are considered, therefore non-perturbative approach as Lattice-QCD has been provided. The L-QCD is a computational method that mainly infers the thermodynamics behavior of a lattice structure of quarks linked by gluons, for different T and μ_B values.

As a brief introduction to L-QCD methods, an important variable used is the Polyakov loop, which provides a measure of the quark deconfinement along the Lattice [10]. This is defined to inspect the characteristics of QCD chiral symmetry breaking, which is exploited in theoretical models to explain light mesons appearance [11] and also indicate the behavior of the quark effective mass [10]. Past models, with $\mu_b = 0$ and using the Polyakov loop as a variable, show that deconfinement and chiral symmetry restoration happen at the same temperature T_C[9].

Figure 1.4 shows an L-QCD plot for the scaled energy density ϵ/T^4 as a function of T/T_C, within limit condition $\mu_B = 0$. Each curve is found considering a different number of quark flavors and T_C represents the phase transition temperature at $\mu_B = 0$. Therefore, in a small values range around $T/T_C = 1$ an abrupt rise in energy is found

[8]Considering k = 1 for Plank units.

[9]This proves that deconfinement coincides with the shift of quark constituent mass to its real one, at vanishing baryon potential.

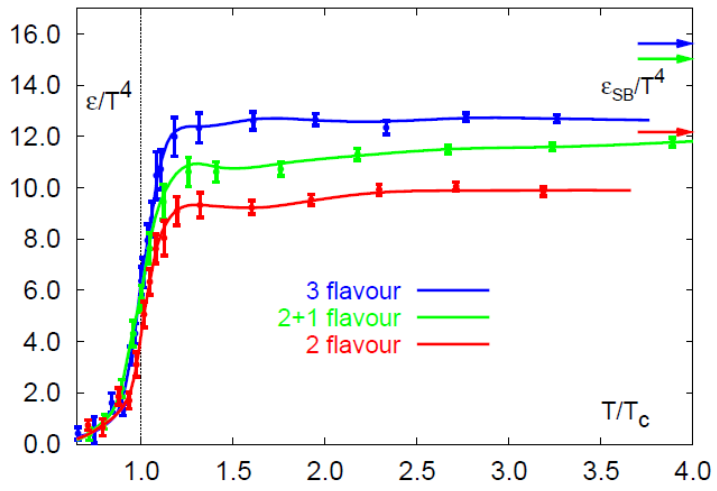


Figure 1.4: Lattice QCD results for ϵ/T^4 as a function of T/T_c, considering different n_f cases and showing the Stefan-Boltzmann limit ϵ_{SB}/T^4 [12].

from a low value in the hadron regime to a higher plateau, slightly below the limit for a non-interacting quark and gluons gas defined by the following Stefan Boltzmann's law:

$$\frac{\epsilon_{SB}}{T^4} = \left(16 + \frac{21}{2} n_f\right) \frac{\pi^2}{30}, \tag{1.8}$$

in which n_f is the considered number of flavors. As the transition is a crossover at T_C value, strong interactions are still ongoing in the energy range around T_C providing a slower function rise. In this region, the matter behaves as a strongly interacting QGP (sQGP) and the quark-gluon gas limit is believed to be reached slowly at higher temperatures [12].

The value of T_C has been investigated over the years and the last results for L-QCD provide a crossover temperature estimation $T_C \in [149, 163]$ MeV [13].

1.1.3 QGP probes in heavy-ions collisions

At LHC heavy-ions collisions are employed to study the QGP crossover phase transition region at temperatures characteristic of the $\mu_B \to 0$ limit. The formation and evolution over time of QGP matter, inside ALICE, is schematized in Figure 1.5 considering mainly 6 steps [14]:

- **Collision:** two beams of heavy ions are accelerated at relativistic energy providing a high Lorentz contraction. As shown in Figure 1.5, the distance between the centers of two colliding nuclei is defined by the impact parameter b, which is used to know the number of nucleon-nucleon collisions performed.

- **Initial state:** within nucleon inelastic collision partons interact considering a large Q^2 transfer momentum range. Softer scattering partons mainly enter an initial pre-equilibrium phase of a weakly interacting partons gas, while hard scattering produces high-momentum gluons and quarks.

- **QGP formation:** creation of even softer partons in the initial state leads to the QGP formation at $\sim$ 1 fm/c after the ions collision. At this point the matter

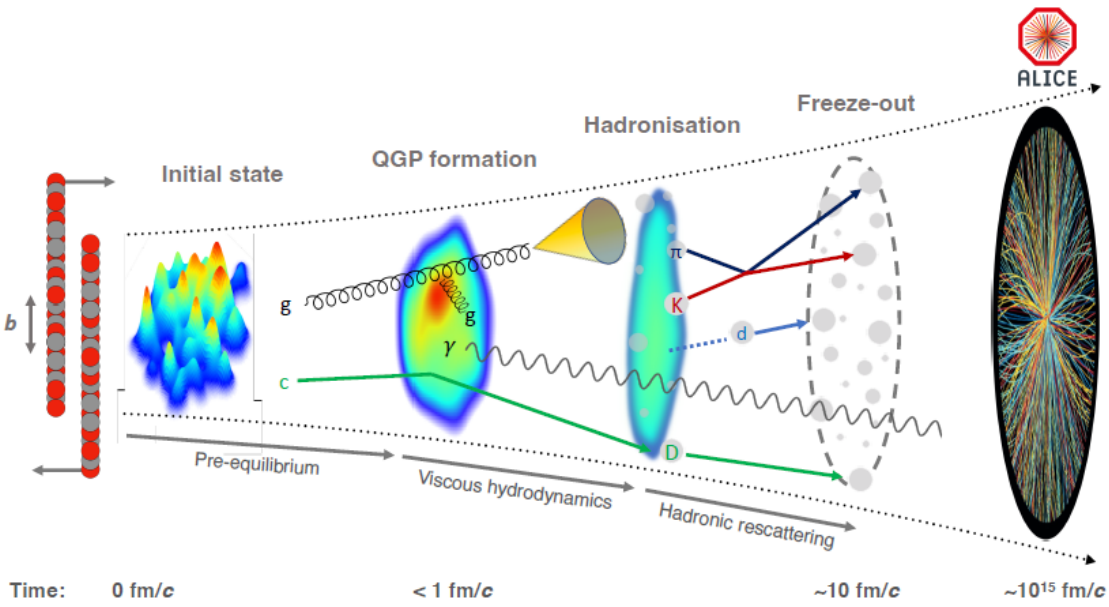


Figure 1.5: Heavy-ion collision time evolution inside ALICE [14].

expands driven by multiple partons interactions and its behavior can be described through hydrodynamics, considering an analysis of the bulk viscosity.

- **Hadronization:** temperature and energy density decrease through the expansion started from the collision point, then, close to the transition temperature, partons begin to confine into hadrons considering different times and phase-space conditions since the transition is a crossover.

- **Freeze-out:** after hadrons materialize out of partons, the energy density could be high enough to allow further inelastic collisions causing a new hadrons "chemical" composition. Reaching T_{chem} (it mainly coincides with T_C for $\mu_B \to 0$) the particle composition freeze-out while elastic collision, as for π and K in the figure, continues until T_{kin} is achieved at ~ 10 fm/c [14] after the initial collision.

- **Detection:** ALICE detect final particles from initial collision after $\sim 10^{15}$ fm/c.

At LHC collision energies the QGP state is generated with record lifetime and volumes, while its features are inspected in ALICE detecting specific decaying products. In heavy ion collisions, the QGP lifetime (few fm/c) does not allow a QGP direct study, so the analysis relies on some probes divided into 3 categories considering the system evolution phases in which they were generated [1] [14]:

- **Soft probes**: they include particles produced within the hadronization phase resulting from the soft scattering products of the collision. These are sensitive to the whole medium evolution, so they are used to analyze the final QGP hadrons production and the statistical/thermal properties of the bulk by exploiting the following methods:

  - **Hadrons multiplicity measurement**: the charged particle multiplicity per unit of pseudorapidity is studied as a function of collision centrality and is used to estimate the energy density produced during the collision [14].

  - **Hadrons spectrum and yields**: the hadrons spectra at kinetic freeze-out are studied to infer the mechanisms of the hadronization phase at different p_t. Furthermore, the different hadron yields and the reconstruction of resonance decays are used to describe the final system hadrochemical composition, which is studied to mainly obtain information on the composition of the first collision stages.
  - **Strangeness enhancement**: this is included in the study of hadrons yields and it is a main signature of QGP formation. In detail, initial colliding ions have small strangeness content and provide no net strangeness. Therefore, the formation of strange hadrons as final collision states implies a higher strange production within QGP than the hadrons gas phase.
  - **Fluid Dynamics**: the plasma behaves as a perfect fluid that shows different pressure gradients providing an anisotropic expansion behavior. This expansion is a consequence of partons scattering after the collision, therefore the azimuthal anisotropies of hadron spectra can be useful to infer the shape of the fluid's early stage. The analysis of the azimuthal particle distribution shows elliptic flow evidence that remarks an "almond shape" form for QGP first stages [14].

- **Hard probes**: these processes consider particles generated at high Q^2 momentum transfer during the collision of the initial ions, before QGP evolution. The dynamic and cross-section of these products can be treated using perturbative QCD and their interactions with the medium are used to infer QGP properties. The main variable used in such an analysis is the nuclear modification factor:

$$R_{AA} = \frac{d^2N_{AA}/dp_t d\eta}{\langle T_{AA}\rangle d\sigma^{pp}/dp_t}, \tag{1.9}$$

  in which $d^2N_{AA}/dp_t d\eta$ [10] is the differential yield in A-A collision, $\langle T_{AA}\rangle$ is the average nuclear overlap and $d\sigma^{pp}/dp_t$ is the differential cross section for pp inelastic collision. Deviation from the unity of this quantity shows different behavior between heavy ions and p-p collisions, assuming no QGP is formed in pp collisions. Within the ALICE analysis, the main hard probes considered are:

  - **Heavy quarks hadrons and Quarkonia states**: as the mass of charm and beauty quarks are larger than typical medium temperature, their production is dominant within hard scattering at the early stage of the system evolution. Passing through the medium, these quarks go through elastic and inelastic collisions probing effectively the QGP and then forming heavy-quarks hadrons and quarkonia-bound states. The yields of these hadron states are investigated as they probe the collision dynamics at long and also short timescales.
  - **Jet quenching**: high p_t quarks or gluons are also produced in hard scattering at initial stages. These pass through hot and dense matter suffering energy and momentum losses due to gluon radiation and elastic collisions, producing showers. At the hadronization phase, these processes end by defining jets of colorless hadrons. Therefore, jet quenching is defined as the shower development and modifications due to the medium, studied considering yield suppression and the found geometry of jets.

[10] η is the pseudorapidity and p_t is the particles transverse momentum.

- **Electromagnetic probes**: several mechanisms during the system evolution can produce photons or dilepton states. The thermal photons produced by the QGP medium as $q\bar{q}$ annihilation, are used to infer the plasma temperature and are clear evidence of quark deconfinement. This interesting signature is a small signal in an energy spectrum dominated by background, as prompt photons, coming from initial hard-scattering, and photons produced in the hot hadronic phase.

A strong signature of the QGP formation in heavy-ions collision is the strangeness enhancement, as one of the first proposed. Recent 2017 results, shown in Figure 1.6, highlight a similar behavior for p-p collisions at high multiplicity (considering $\sqrt{s} = 7$ TeV). Indeed the plot, found for the ratio of strange particles to pions, shows higher values as a function of multiplicity $dN_{ch}/d\eta$ for p-p collision data, finding compatibility with the Pb-p data and reaching the highest values for Pb-Pb collisions. Therefore, these results may point to physics mechanisms that explain the suppression of strangeness in fragmentation, which must be investigated in future experiments [15].

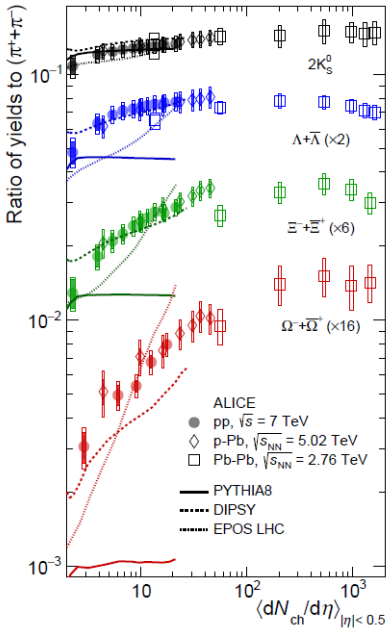


Figure 1.6: p_T-integrated yield ratios for strange and multi-strange particles to pions $(\pi^+ + \pi-)$ as a function of multiplicity, with rapidity $y < 0.5$. The plot shows the comparison between found data for different collisions and the Monte Carlo simulations performed for p-p collisions [15].

1.2 ALICE layout and its TOF detector

During the LS2 LHC (2019-2022) phase, ALICE collaboration aimed to improve the experiment capabilities to probe the QGP with the methods explained before, inspecting also p-p collisions at high multiplicity. Therefore, the upgrade involved the overhaul of some core detectors to improve the pointing resolution and the readout rate supported by ALICE. Furthermore, the ALICE readout system was also reinvented to supply continuous readout, obtaining larger data samples and coping with a high interaction rate.

The Runs 3 and 4 planning is to record data at interaction rates of 0.5 MHz to 1 MHz and 50 kHz for p-p and Pb-Pb collisions. This will allow the experiment to store an integrated luminosity respectively of 200 pb^{-1} and 13 nb^{-1} [16].

The following section explains the current ALICE detector layout, describing each sub-detector briefly and focusing on the LS2 upgrades. Furthermore, it includes a full overview of the ALICE-TOF detector layout which has not undergone any changes during LS2, while its readout electronic system was modified.

1.2.1 ALICE detector Layout

The ALICE experiment was designed to detect large particle multiplicities [11] in a wide momentum range (from $\sim$ 100 MeV/c to $\sim$ 100 GeV/c). As shown in Figure 1.7, ALICE is mainly composed of a barrel structure, built around the collision point, and a forward muon system covering higher mid-rapidity regions. L3 solenoid magnet is installed inside the barrel structure generating up to 0.5 T magnetic field, while the forward system is embedded in a dipole magnetic field of 3.0 Tm bending power [16].

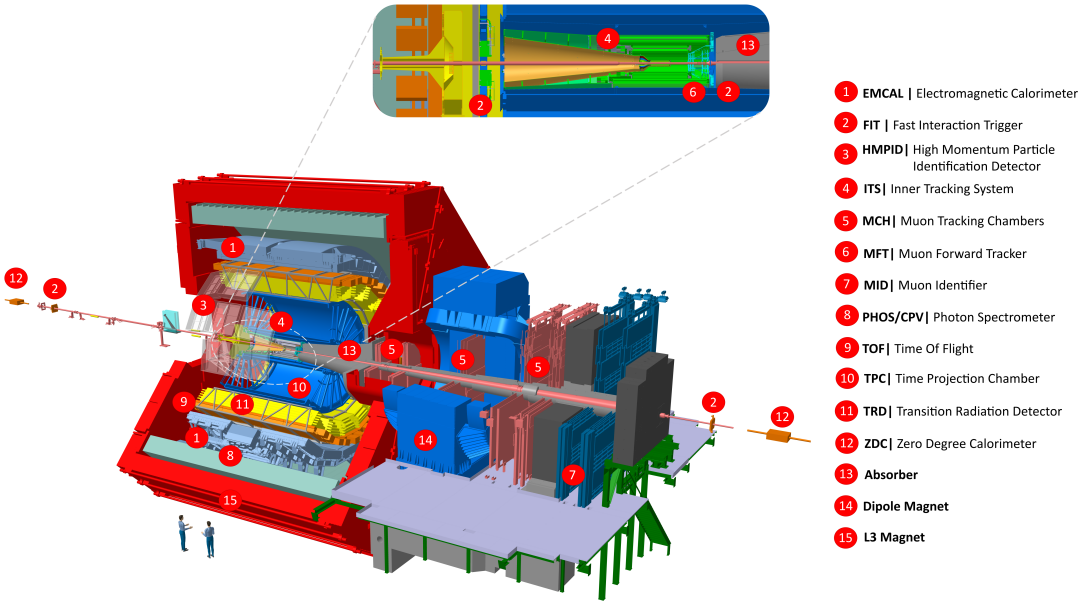


Figure 1.7: Internal view of the ALICE experiment, considering the direction of the beam pipe [16].

The ALICE barrel covers all the azimuth around the beam direction and provides a tracking system and outer sub-detectors for particle identification (PID). One of the LS2 upgrades regards the ITS2 (Inner Tracking System) detector, located between 2.2 cm and 39.5 cm from the beam pipe. Its layout consists of 7 concentric layers over

[11] $\sim$ 2000 charged particles per rapidity unit at mid-rapidity.

$\eta < |1.3|$ mid-rapidity region, built using ALPIDE[12] pixel sensors technology[13]. The main goal of the ITS restyling was to improve primary or heavy-flavor hadrons decay vertices reconstruction, allowing also an improved detection of low-p_t particles. Another requirement achieved is the high readout rate, to avoid occupancy saturation and support the high luminosity reached at ALICE.

The following TPC (Time Projection Chamber) in the radial direction is the main tracking sub-detector of the ALICE barrel, with a cylindrical shape of 88 m^3 volume and a length of 5 m along the beam direction. It covers $|\eta < 0.9|$ mid-rapidity region and mainly provides particle tracking from low 100 MeV/c up to 100 GeV/c. The readout system is placed on a central electrode, that divides the chamber into two halves organized in sectors. To cope with higher readout rates, the TPC readout sectors were upgraded replacing the Run 1 and Run 2 MWPC[14] technology with GEM (Gas Electron Multiplier) sensors. These last reduce the ions backflow into the chamber drifting volume, allowing for reduction of space charge effects that set limits on the achievable readout rate in previous Runs [16]. Besides tracking, TPC provides PID features as well, since it can measure particle stopping power ($\frac{dE}{dx}$).

On the outer TPC surface, the first sub-detector of PID system is the TRD (Transition Radiation Detector), covering the mid-rapidity range $\eta < |0.84|$. It uses a system of radiators and MWPCs for hadron-e discrimination, at p_t greater than 1 GeV/c, through $\frac{dE}{dx}$ measurements and the TR photons detection.

At an even outer radius, the TOF(Time Of Flight) detector is set around the barrel covering a mid-rapidity range $\eta < |0.9|$. It provides particle identification in the intermediate momentum range of 0.3-5 GeV/c and its structure will be explained later.

To achieve hadrons PID in the high p_t momentum range ($p_t > 1$ GeV/c), the HMPID (High-Momentum Particle Identification Detector) is finally set. It is designed with an acceptance of only 5% of the barrel phase space and uses proximity-focusing Ring Imaging Cherenkov (RICH) counters.

Furthermore, attached to the ALICE magnet coil, a large acceptance section of the barrel is covered by electromagnetic calorimeters: the EMCal (Electromagnetic Calorimeter) and the PHOS (Photon Spectrometer). The EMCal is a Pb-scintillator sampling calorimeter, covering a mid-rapidity range $\eta < |0.7|$ at $\phi = 107^o$. The PHOS is a high-resolution calorimeter made of lead-tungstate crystals, exploiting charge particle veto (CPV) and placed in front of its active area [17].

Finally at large pseudorapidity region ($-4.0 < \eta < -2.5$) the Muon Forward system is located to measure the resonance spectrum of the heavy-quark vector mesons and lighter ϕ, reconstructing their $\mu^-\mu^+$ decay channel. As was for the Run 2 setup, this sub-system provides a first absorber, to avoid hadrons and photons detection, followed by the MCH and MID detectors, as shown in Figure 1.7. The Muon Tracking Chambers consists of ten layers based on MWPC technology, located at different stations inside the magnet dipole. Then a second muon filter is set and four RPC-based layers compose the Muon Identification Detector. A further shield is set at the end of the MID to protect the chambers from particles produced at large rapidity. During the LS2 upgrade, as shown in the zoomed image of Figure 1.7, the Muon Forward Tracker was set between the interaction point and the absorber, including five layers based on ALPIDE chip technology. The MFT goal is to enhance the muon spectrometer's physics program, improving the

[12] ALICE Pixel Detector.

[13] Monolithic Active Pixel Sensors (MAPS).

[14] Multi-Wire Proportional Chamber

pointing and mass resolutions for the reconstruction of heavy-flavour hadron decays [16].

Another LS2 upgrade was installing the FIT (Fast Interaction Trigger) detector that provides five sensor arrays, located at different positions along the beam pipe direction. In particular, two of them are based on Cherenkov counters and were installed on both sides of the interaction point, as shown in Figure 1.7. These two mainly aim to estimate collision parameters as collision time, which is important in the TOF PID technique. A further layer with ring geometry, based on scintillators, is located on the other side of the muon system absorber. This is used with the Cherenkov layers to provide fast trigger signals for some ALICE detectors and help in collision parameter calculations. Then, each of the last two scintillator counters is set far away on either side of the ALICE detector to monitor the background and help in tagging diffractive events [18].

At a larger distance, ZDC (Zero Degree Calorimeter) detector was built to measure the centrality for nucleus-to-nucleus collisions. This setup is set at 116 m on either side of the interaction point and it exploits two calorimeters respectively to detect spectator neutrons and protons [17].

1.2.2 The TOF detector

The TOF PID technique identifies particle types through indirect mass measurements, achieved through the estimation of particle momentum p and velocity v. The tracking system is used to infer p, while the velocity $v = L/t$ is found through the L particle trajectory length and the time of flight t, measured directly by the TOF detector. As follows from the p and v relation given by special relativity, the particle mass m is calculated as:

$$m = \frac{p}{c}\sqrt{\left(\frac{c^2t^2}{L^2} - 1\right)}. \tag{1.10}$$

Considering two different mass particles with the same p and over the same track length L, the Δt value follows this equation (in the limit $\frac{m^2c^2}{p^2} < 1$):

$$\Delta t = |t_1 - t_2| = \left|\frac{L}{2c}\left(\frac{m_1^2c^2 - m_2^2c^2}{p^2}\right)\right|, \tag{1.11}$$

in which t_1 and t_2 are the measured TOF for particles one and two. Therefore, particle discrimination is accomplished in TOF detectors considering the ΔT separation provided in units of time resolution σ_{TOF}. As shown by equation 1.11, particles can be distinguished using the measured time if the particle momentum is not so large that the $|t_1 - t_2|$ difference becomes comparable to the σ_{TOF} time resolution of the detector.

Figure 1.8 shows achieved PID performance of the TOF detector, considering the last provided σ_{TOF} resolution of $\sim$ 60 ps (56 ps) using LHC Run 2 detector calibration [19]. Looking at the $3\sigma_{TOF}$ horizontal line, the detector can discriminate π/K and K/p up to momenta respectively of $\sim$ 3 GeV and $\sim$ 5 GeV.

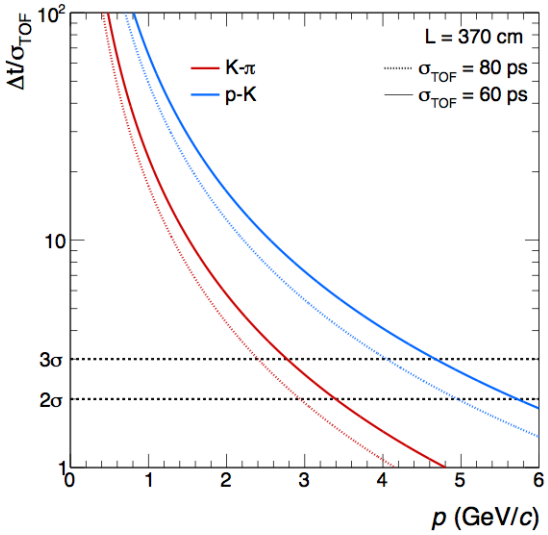


Figure 1.8: The plot shows the comparison between the simulations obtained considering σ_{TOF} equals to 80 ps and 60 ps, for π/K and K/p time difference separation as a function of particle momentum. Through a new calibration [19] in Run 2, TOF reached a 60 ps resolution extending the momentum range where it can discriminate particles.

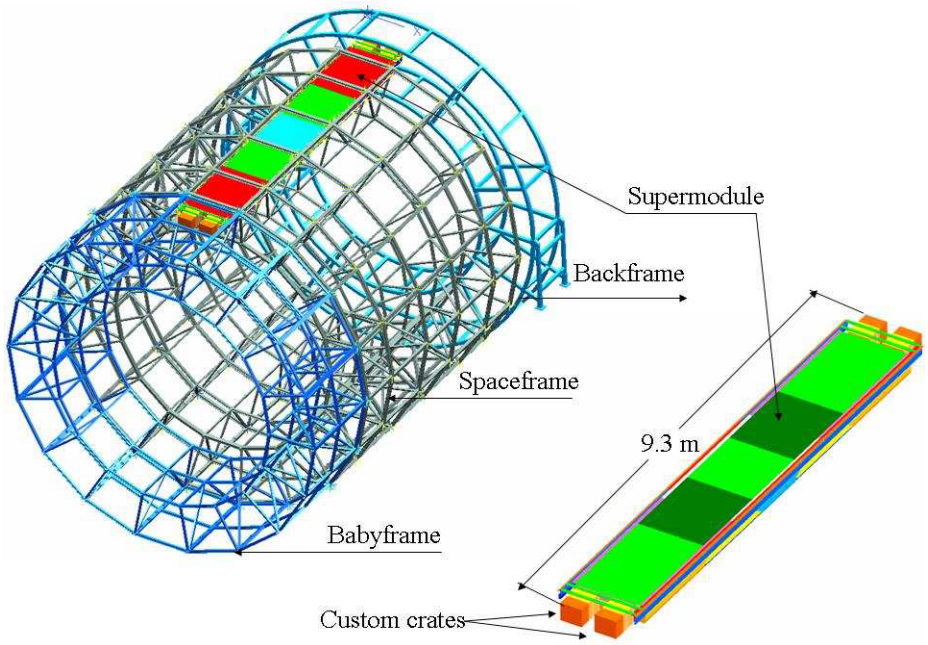


Figure 1.9: Schematic of a SuperModule inserted in the ALICE Spaceframe. The Babyframe and the Backframe are mainly used for electronics and services purposes (cooling, gas, high-voltage) [20].

The TOF layout takes a modular structure set in the outer cylindrical surface of the ALICE barrel Spaceframe, which contains the Tracking system and the TRD detector [20]. As shown in Figure 1.9, taking the beam pipe direction as a reference, the TOF uses aluminum structures to arrange modules longitudinally in groups of 5 and transversally in sectors of 18, for a total of 90 modules. Each of the longitudinal groups, composed

of five modules, is called SuperModule, which provides on either side two crates hosting DC-to-DC low voltage power supplies and the readout electronics.

Each TOF module contains sets of double-stack MRPCs (area of 120 x 7.4 cm^2), used as the single detecting units of TOF detector. The module supports external front-end electronics and services to manage the functioning of the MRPC strips. In general, the SuperModule structure supplies modules with a different number of strips, so for best coverage 15 strips are used in its central module while each of the four external ones takes 19 strips. The TOF detector instruments and reads 1539 MRPCs, which supply a total of 10^5 readout channels [19].

1.2.3 Multi-gap Resistive Plate Chamber (MRPC)

The Multi-gap Resistive Plate Chamber is a gas particle detector that operates in very high voltage regimes (some kVs). As shown on the left of Figure 1.10, the MRPC is designed considering two plates working as the anode (+HV) and the cathode (-HV), providing a multi-gap structure in between built using intermediate resistive plates. Indeed, the voltage is only applied on the outer plates letting the intermediate ones electrically float to their proper voltage value. The MRPC readout is based on pick-up pads placed on the outer surfaces since this detection technique is based on charge inductance on both the anode and cathode. The high voltage provides a uniform and high electric field for each gap, allowing a very fast avalanche formation between resistive plates. The final signal obtained, as charge induced on both the cathode and anode, is the sum of all the avalanches generated in the gaps.

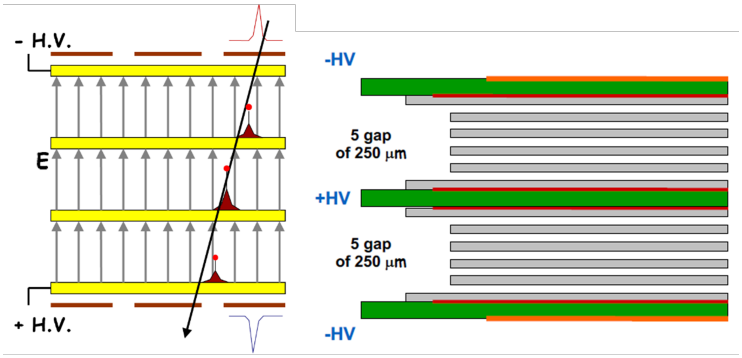


Figure 1.10: On the left, a schematic of a general MRPC is shown, while on the right a dual-stack MRPC similar to the ALICE TOF one is schematized (the +HV and -HV represent respectively the applied positive and negative high-voltages).

The ALICE MRPCs strips were developed in a double stack structure as shown in the right scheme of Figure 1.10. Each stack consists of 5 equally spaced gas gaps of 250 μm, providing specific voltages considering the 2 external cathode plates and the common anode. Each stack has external and internal resistive "soda-lime" glasses of respectively 550 μm and 400 μm thickness. The strip exploits an active area segmented into two rows of 48 pickup pads of 3.5x2.5 cm^2, for a total of 96 readout pads [20] [21]. Such a geometry was chosen to provide a direct differential signal from each MRPC pad to the front-end electronics and to allow a better pointing of the device towards the interaction region. The double-stack MRPC strips, installed on the ALICE TOF detector, ensure an efficiency close to 100% and a time resolution below 50 ps [21].

1.3 The TOF readout system

Particle Physics experiments are developed to detect signals from many events, among which the interesting ones must be classified. A logic signal called trigger implements a discrimination mechanism between interesting events and background, following specific criteria derived from detected signals (such as detector coincidence, number of detected events across the detector ecc.). During LHC Run 1 and Run 2, the ALICE readout was designed as fully triggered. It was based on a three-level trigger model that supplies first signals L0, L1 and a last L2 trigger to complete the data acquisition. These three required different latencies achieving a final readout rate of 1 KHz over a Pb-Pb interaction rate of 8 KHz.

During LHC Run 3, a continuous readout and compressing data strategy is exploited to support the increased LHC interaction rate. Therefore, to synchronize each readout and data processing branch, the data stream is divided into the so-called time frames (TF) of 128 LHC orbits length ($\sim$11 ms). Furthermore, each TF is subdivided into Heart-Beat frames (HBF), providing data packets in a specific time window of $\sim$89.4 μs. The ALICE readout upgrade regarded all the readout infrastructure starting from the new Online and Offline processing farm (O^2) design, which manages and compresses the received data from the setup. The upgrade also included a new design for some sub-detector readout chains to support continuous readout, while others still operate in triggered mode.

The TOF detector update considered its electronic readout system and took only minimum intervention to achieve the continuous readout. In particular, the main readout elements are still the TRM cards which were not modified or upgraded.

1.3.1 The TOF redaout chain

As mentioned in subsection 1.2.2, each TOF Supermodule provides on either side 2 VME crates, supporting a DC-to-DC converter for low-voltage power supply and the Supermodule readout electronics. Each crate can contain up to 12 boards: 1 DRM2 (Data Readout Module 2), 1 LTM (Local Trigger Module) and 9 or 10 TRMs (TDC Readout Module).

The first front-end system is shown in Figure 1.11 and it manages the differential signals from the MRPC pads through FEA (Front-end Analog card) cards, which communicate directly with the crate boards. Each FEA (Front-end Analog card) supplies 3 NINO ASICs to amplify and discriminate the MRPCs output signals. Each NINO chip provides 8 input differential channels to match the MRPC pad differential output [20]. The FEA card collects the NINO outputs and provides a connection through shielded cables to the TRMs. Each differential output is then digitized by the HPTDC (High-Performance TDC) ASICs, hosted by the TRMs. Figure 1.11 also shows the FEAC (Front-end Analog Controller) custom card, used to collect signals from 12 FEA cards. Then LTM card manages the FEAC signals for trigger purposes and monitors the FEAs low-voltage supply.

The readout electronic system works through the VME bus provided by the crate. The VME is an asynchronous communication protocol that supports master/slave topology. Therefore, the system is based on a single master topology, in which the DRM2 board acts as the VME master to manage the data transactions with the TRMs slaves. The LTM board instead works as an independent interface between the front-end system and

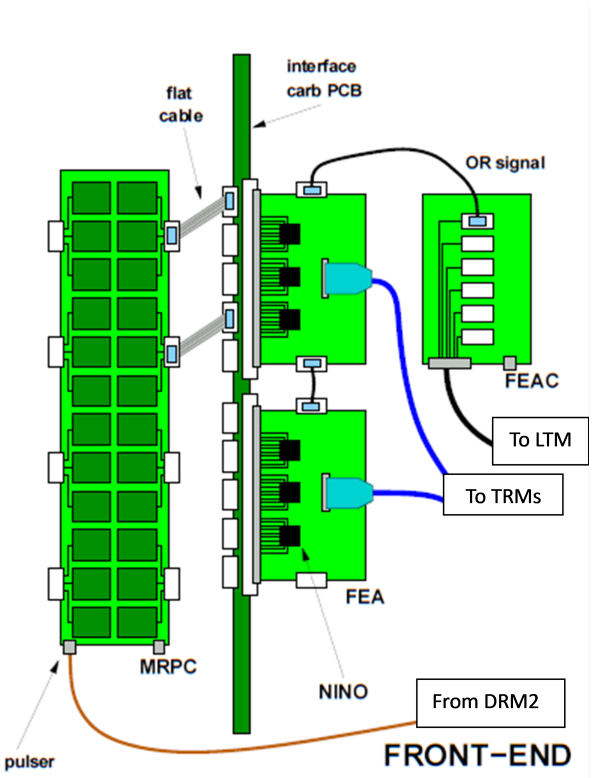


Figure 1.11: Front-end electronics provided for one MRPC, showing the connection with specific boards inserted in the readout crate [20].

the CTTM (Cosmic and Topology Trigger Module) ALICE trigger system. In detail, each board type hosts different components to accomplish specific operations:

- The **TRM** hosts 30 HPTDCs for a total of 240 differential input channels fed by the front-end output signals. An Actel APA 750 FPGA controls the data stream received from these ASICs, while some external SRAM memories buffer the data until they are read.
- The **DRM2** uses a Microsemi Igloo2 FPGA to manage the TRMs readout and data stream towards the DAQ system. The FPGA provides different operations and features, through the control of some components on board [22]:
  - An SSRAM of 1Mbx36 memory is used to buffer the TRMs data, read by the DRM2 through the VME bus.
  - The GBTx ASIC drives the data toward the DAQ system. This ASIC provides a fast link through a VTRx optical transceiver, reaching a bandwidth of 3.2 Gb/s. The same link receives the 40 MHz LHC clock and the input trigger signal, which is used to start the TRMs readout.
  - The board also features other 2 slow-control links to the FPGA. The first is a CONET2 optical link[15] to configure the connected TRM boards, monitor data taking and read voltage and temperatures. The other one is an Ethernet link connected to an Atmel ARM processor for reprogramming remotely the FPGA.

[15]Developed by CAEN

- The **LTM** board hosts 2 different FPGAs with different implemented logic to monitor the voltage and temperature of the front-end system and generate trigger signals sent to the CTTM and then to the TCP (Trigger Central Processor).

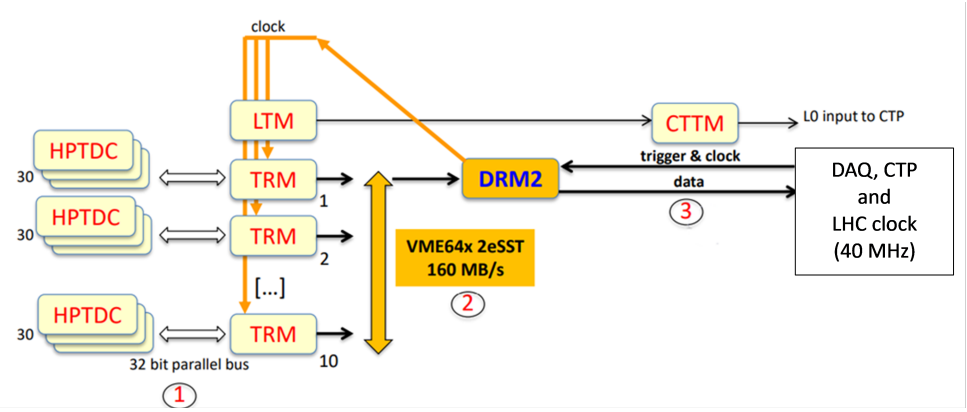


Figure 1.12: Schematic of the readout system, considering each step from the HPTDCs to the final ALICE DAQ system [23].

The main TOF upgrade, performed during the LS2 period, is the DRM2 board introduction inside the readout system, which is responsible for the TRMs readout and the LHC input clock distribution to all the crate boards. In detail, the readout mechanism is shown in Figure 1.12, highlighting each specific step starting from the MRPCs signal detection [22]:

1. The TRMs cards digitize the signals from the FEE electronics using the hosted HPTDCs, waiting for a trigger.

2. The DRM2 receives an input trigger through the GBTx link and sends it directly to the HPTDCs via the VME interface. Then the data are read and stored on the DRM2 SSRAM.

3. Finally the data are sent through an optical link to the ALICE DAQ system, using the GBTx ASIC.

Such a system has been implemented to mimic a continuous readout through a periodic trigger of 33 KHz, as explained in the following sections. Furthermore, the VME64 readout (40 MB/s) has been upgraded to the VME64 2eSST [16], allowing the DRM2 to read data from TRM cards with a bandwidth of up to 160 MB/s. The new design of the TOF readout matches the features of TRM boards, which have been used since the start of LHC Run 1.

1.3.2 The TRM board and the HPTDC

The TRM is a 9U VME slave card that hosts 30 HPTDC ASICs, organized in two separate 32-bit parallel readout chains. Its layout consists of 10 piggyback cards hosting 3 HPTDCs each, arranged on both sides of the board. To control the data stream from

[16]The DRM card reads 64-bit data on the falling and rising edges of the DTACK signal, considering a frequency of 10 MHz.

the ASICs, the board provides a central FPGA that implements the VME interface and acts as [24]:

- **Readout controller**: it manages the readout of both the TDCs chains as it receives the first trigger input via VME interface.
- **Event manager**: considering some parameters describing each triggered event, the FPGA decides to discard the data or collect them in an event packet with a specific header associated.

Considering the logic block scheme of Figure1.13, these FPGA processes are shown separately to explain better the data stream during the TRM readout. As mentioned, during Run 3 a unique trigger signal is used to start the readout of the HPTDCs data buffers. Then the readout controller sends the data from both TDCs chains to the event manager, which scans the data related to each event. All the valid event packets are buffered in an external dual port RAM waiting to be read by the DRM2 through the VME interface, passing through the output FIFO.

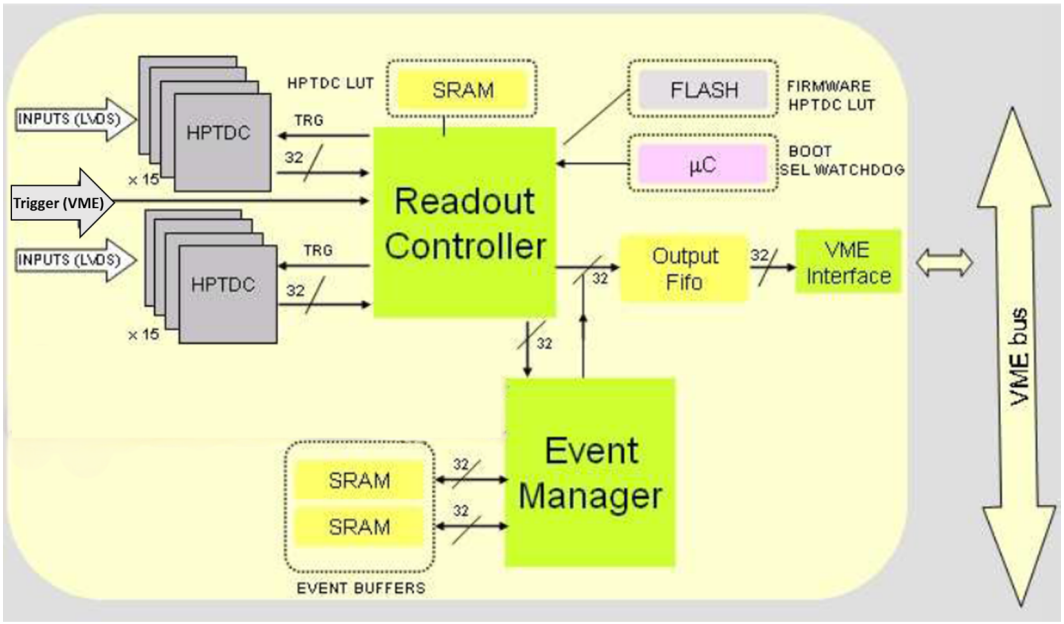


Figure 1.13: Logic block scheme of the TRM board, showing each process implemented on the core FPGA [24].

Specifically, the HPTDC is a multi-channel TDC with a programmable binning (24.4 ps to 781 ps), developed in a 0.25 μm CMOS technology by the CERN/EP microelectronic group [25]. This ASIC was built to provide multi-hit measurements in a multi-events environment. Figure 1.14 shows its architecture highlighting with different colors its 2 main sections: the Timing Unit and the Data Processing Unit.

The Timing Unit works considering a precise reference clock of 40 MHz (LHC clock), which feeds a PLL generating configurable 40/80/160/320 MHz clock frequencies. The time measurements are performed using counters running with the generated frequencies and exploiting the state of an internal DLL (Delay Locked Loop). The DLL consists of a 32 delay elements line, that samples the hit signals at each step to provide a binning of $T_{clock}/32$. A further interpolation, within a DLL delay cell, is used in the VHRM (Very High Resolution Mode) mode, exploiting an R-C delay line of 4 equal steps. This mode

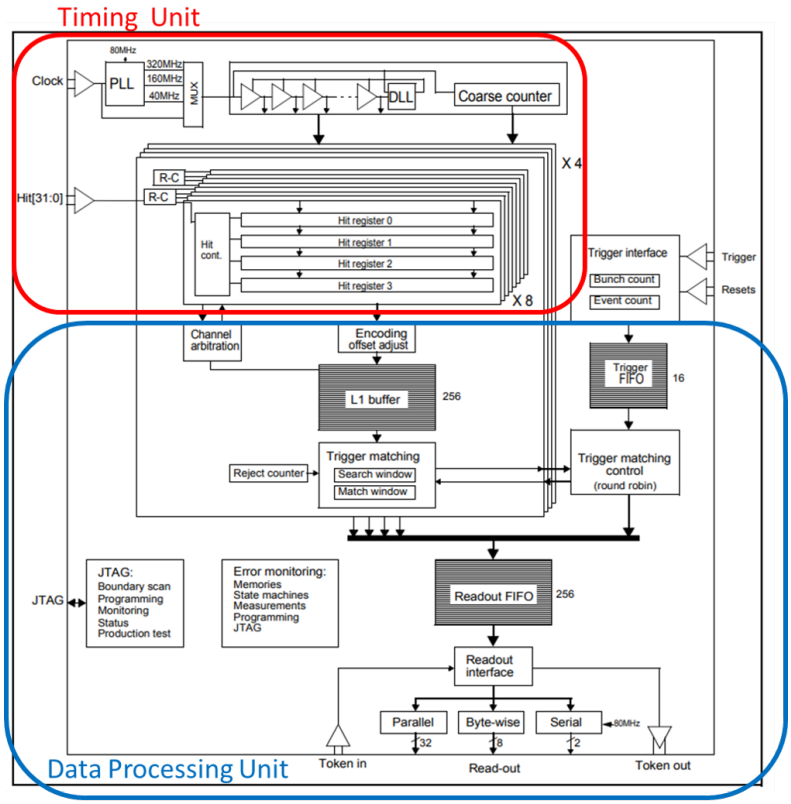


Figure 1.14: HPTDC blocks diagram, in which the Timing Unit and Data Processing Unit are indicated in red and blue respectively [25].

leads to a 24.4 ps binning, but limits from 32 to 8 the number of available TDC input channels. As each of the 30 HPTDCs, installed on the TRM, works in the VHRM mode, the NINO ASIC was built as an 8-channel output device to feed directly a corresponding TDC. Each of the 32 input differential channels can perform time measurements for both the leading and trailing edges of a hit signal. Then a 4-measurement buffer is provided for each of them, limiting to 5 ns (10 ns guaranteed) the distance between 2 consecutive measurements as indicated by the constructor [25].

As shown in Figure 1.14, the channels are organized, inside the ASIC architecture, into 4 groups of 8 channels, and the Data Processing Unit mainly contains the processes, synchronous with a 40/80/160 MHz clock, implemented to manage the data up to the readout. Each group stores the channel measurements in a 256-word deep buffer (L1) waiting to be serviced by the trigger-matching unit (optionally disabled for a free-running acquisition). This function associates the timestamp of a trigger signal (measured with a 25 ns binning) to the hits measurements. The matching is provided considering the configurable time window and trigger latency, which is the time needed for the trigger arrival. Trigger time tags are stored in a unique 16-deep Trigger FIFO waiting to be used inside the matching function.

Then all the data that defines an event including hits and specific event words, are written in a FIFO collecting all the 4 groups' data. All the accepted data from the TDC can be read using a parallel or a serial interface, in words of 32 bits. As mentioned, the TRM uses the parallel interface configuring chains of 15 TDC slaves that are read by the core FPGA programmed as the master chip.

1.3.3 Implementation of the TOF continuous readout

During LHC Run 3, the continuous readout system of ALICE TOF detector is implemented exploiting the trigger matching function and the buffer features of the HPTDC. The trigger matching is shown in Figure 1.15, where at each trigger input a trigger latency is set backward in time defining the trigger time tag that must be matched to specific hit measurements, found within a configured matching window. The search for hits, matching a trigger, stops when a hit older than a wider search window is found or the L1 buffer is empty. All the matched hits are shifted to the readout FIFO while the others are rejected. The HPTDC architecture is designed to work considering the bunch

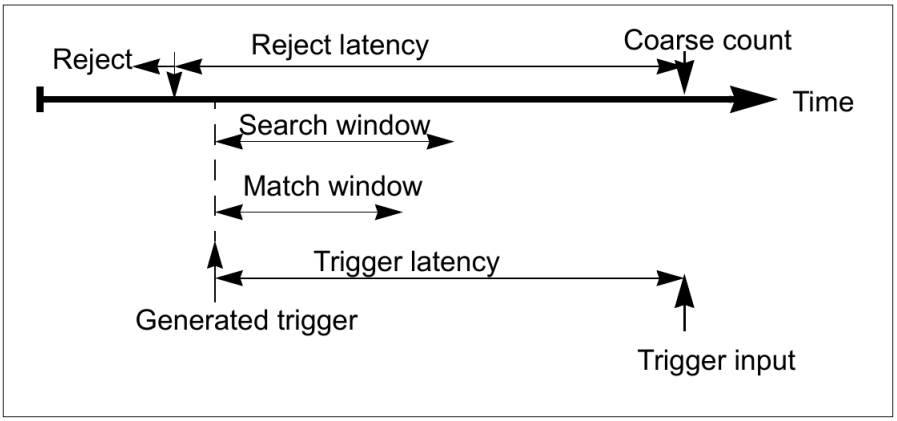


Figure 1.15: HPTDC trigger matching for hits on channels [25].

structure of particle beams, so the trigger time tag identifies a bunch ID, providing a binning of 25 ns since particle bunches cross at LHC with a 40 MHz frequency. Therefore, the latency and matching window time lengths are defined using this specific clock period as a time unit.

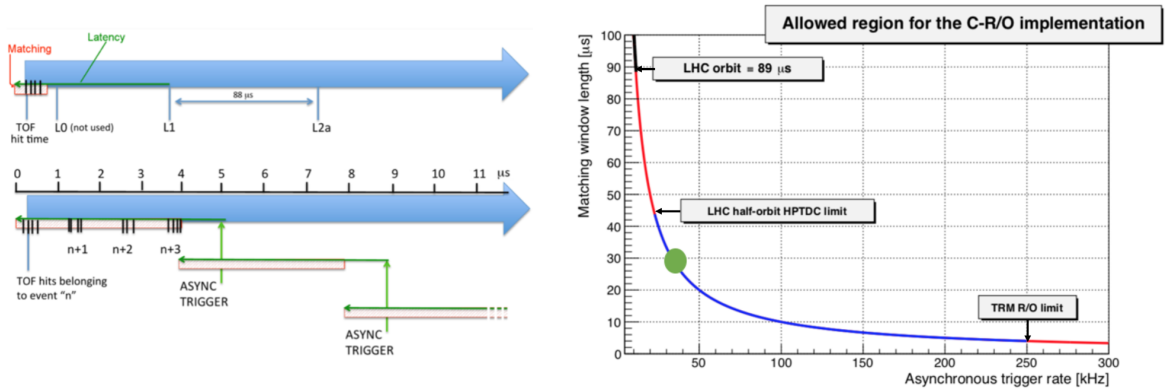


Figure 1.16: On the top-left the matching trigger algorithm, configured for Run 1 and Run 2, is shown. On the bottom-left, a periodic trigger and a specific HPTDC configuration are used to mimic a continuous readout, as for Run 3. On the right, a plot of the allowed values for matching window and trigger rate is shown, in which the green point is the today chosen operational point at LHC [26].

The left plot of Figure 1.16, shows first the trigger matching application during Run 1 and Run 2. Due to the few KHz trigger rates, the trigger latency is set to 6500 ns

(representing the L1 trigger latency), while a matching window of 600 ns was used to collect all the hits matched to the triggered collision. Using a similar configuration, the continuous readout is implemented considering a single periodic trigger with frequency f_T and setting a matching window of $m_W = 1/f_T$ [26]. This idea is shown on the bottom-left plot of Figure 1.16, in which a 250 KHz trigger and a matching window of 4 μs are set. Therefore, as the HPTDC allows matching multiple triggers with hit measurements, a larger latency of 5 μs is used to acquire continuously all the hits registered by the TOF detector.

However, this readout system is constrained by some limits and the allowed f_T and m_W values are shown in blue in the right plot of Figure 1.16. On the one hand, the HPTDC does not allow matching window values larger than half of the LHC orbit. On the other hand, the trigger period must be less than the TRM time for a read cycle of the 30 HPTDCs, which defines a trigger frequency limit of 250 KHz. In the end, the readout time on average must be lower than $1/f_T$, and a good working point was found at a trigger frequency of 33 KHz shown as a green point on the plot. This result was also achieved thanks to the VME64 2eSST upgrade of the VME interface, which increases the TRM readout rate by the DRM2 board.

1.3.4 The picoTDC as a successor of the HPTDC: towards TRM2

Since the TOF detector needs to be reliable and operative until the next shutdown of the ALICE experiment (at the end of Run 4) it is important to provide at least a partial replacement for the 684 TRM modules installed. Especially, several TRM components are no longer repairable or are out of production, which is the case of the HPTDCs and the Actel APA 750 FPGA. It is anticipated that a relatively large amount of TRM boards (50 cards) will not be available in Run 4, while during Run 3 the available spares have already been ended.

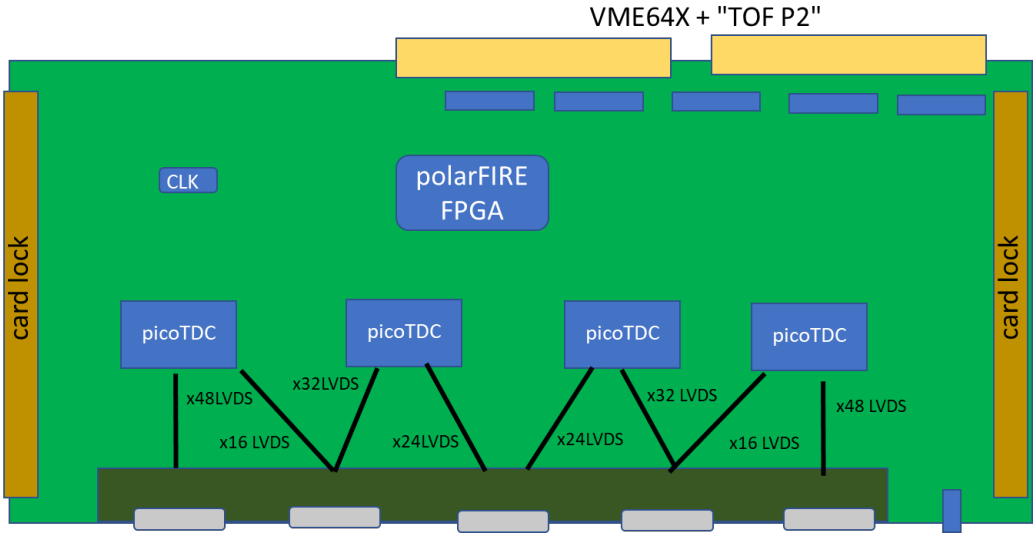


Figure 1.17: Layout scheme of the TRM2 board.

A new project for a TRM2 board started and the first card layout is shown in Figure 1.17. Similarly to the TRM, such a board is a 9U VME64 card and shows new-generation

components. In detail, the Microchip PolarFire FPGA is the best candidate for the core FPGA that controls the board features. This is connected to 4 PicoTDC ASICs that replace the old HPTDCs, in the data digitization. The PicoTDC is a configurable binning TDC, built with a 65 nm CMOS technology, and it shows an architecture similar to the one of the HPTDC that supports continuous readout performance. Such an ASIC provides some new interesting improvements that must be considered in the TRM2 layout project:

- PicoTDC can use all the 64 differential input channels, with all the possible configured binning. Therefore, as shown in Figure 1.17, only 4 ASICS are used to cover the needed 240 channels supplied by the TRM board.
- Fewer piggy-backs can be arranged providing 5 VHDCI connectors for each card side. Each VHDCI connector must feed 24 PicoTDC channels.
- Each PicoTDC is independent of the other, therefore the readout can not be implemented using a chain as in the TRM case considering the HPTDCs.

To test the performance of both the PolarFire FPGA and the PicoTDC ASICs, a test board was designed by the INFN electronics laboratory and the ALICE group of Bologna. The PicoTDC board project aims to test specific configurations of these ASICs set via the PolarFire FPGA.

Executive summary

The ALICE detector is designed to study the QGP state of matter, considering the framework of heavy-ion collisions at relativistic energy. Within the ALICE setup, its TOF detector is used for particle identification in the intermediate p_t momentum range. Before LHC Run 3, the TOF electronic readout system was upgraded to support continuous readout. This upgrade did not include the TRM boards used to digitize the signals coming from the TOF setup. As many TRM components are out of production and the TRM spares are progressively reducing, a new project for a TRM2 board began. This card will provide a new design including a PolarFire FPGA and the new PicoTDC ASICs, which must be tested using a first test board.

Chapter 2

The PicoTDC board and the IPbus protocol

The PicoTDC board is shown in Figure 2.1 and provides a PolarFire FPGA to implement the logic that controls two PicoTDC ASICs, hosted on board. Furthermore, the board supplies different interfaces exploited to communicate directly to the FPGA. Therefore, the FPGA firmware and related software routines can be designed to provide user commands that control the different board features through Ethernet or USB communication protocols.

This chapter provides first a detailed description of the PicoTDC ASIC, since it is the main TRM2 element. Then, it describes the board layout designed by the INFN electronics laboratory of Bologna focusing on the Ethernet connection subsystem, since both the software and firmware were developed using the IPbus protocol. Finally, a brief introduction to the IPbus protocol is provided, explaining how the IPbus is built over the Ethernet protocol.

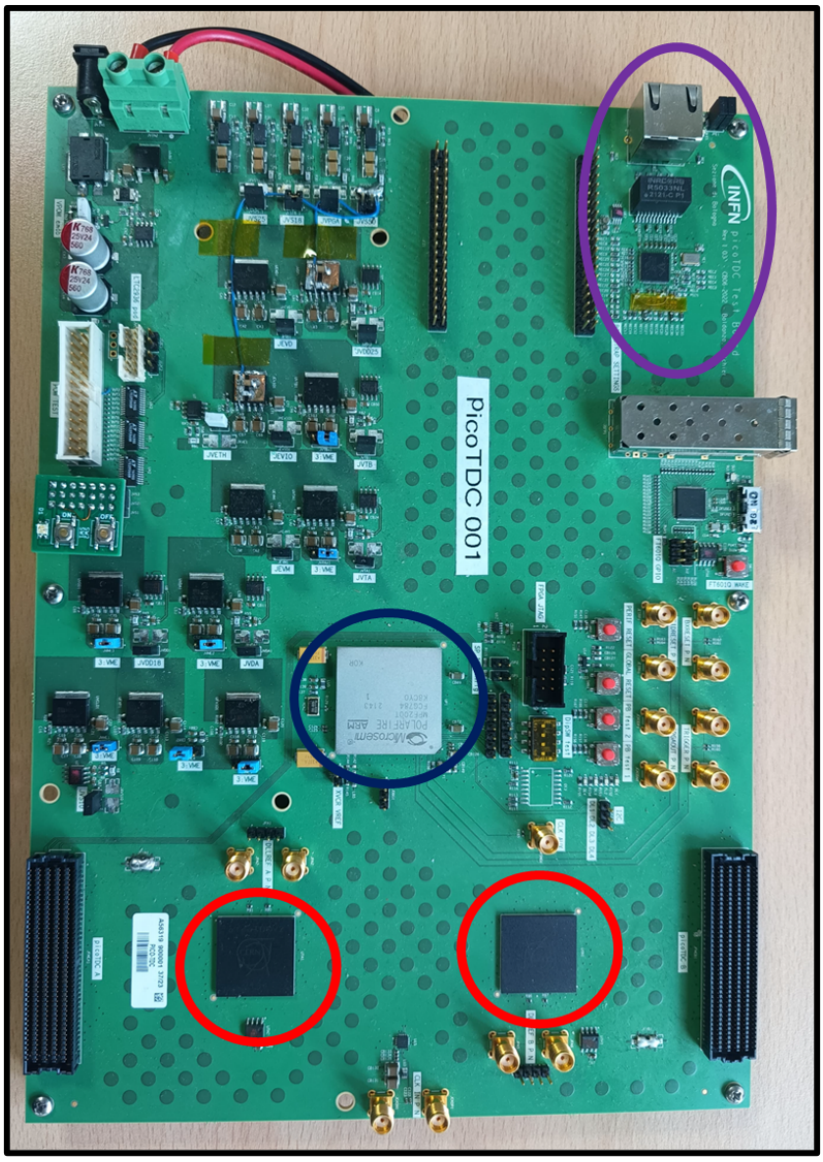


Figure 2.1: PicoTDC board in which the PolarFire FPGA (blue), the 2 PicoTDCs (red) and the Ethernet connector subsystem (purple) are highlighted.

2.1 PicoTDC overview

The PicoTDC is an ASIC designed by CERN that provides high-resolution measurements with a high channel count. Similar devices are used in many scientific domains, especially in HEP physics to build PID and TOF detectors, and tracking systems at high particle rates. The PicoTDC was designed to support applications and R&D activities with different high-time resolution detectors. Therefore, this ASIC does not include any analog front-end and discriminator circuit, since they must be optimized for each specific sensor type.

As mentioned in subsection 1.3.4, the PicoTDC is a 65 nm CMOS technology ASIC and mainly reproduces the HPTDC architecture, divided into two sections: the Timing Unit and the Data Processing Unit. The former decodes the hit signals received in each input channel, while the second elaborates the data and collects them into FIFO memories, waiting to be read out. The PicoTDC, as a new generation chip, provides better resolution and a higher number of input channels than the HPTDC, which has been extensively used in HEP experiments and is now out of stock.

2.1.1 Architecture

This ASIC can detect both the leading and trailing edges of a signal, providing the measures for the arrival time of each edge or a direct Time Over Threshold (TOT) measurement of the signal. For best resolution performance, the TDC must rely on a precise time reference of a differential external clock. Such a clock feeds an on-chip PLL and must run by construction at 40 MHz (quite synchronous with the LHC bunch crossing frequency that runs at 40.08 MHz).

The PLL performs clock multiplication generating frequencies from 40 MHz up to 1.28 GHz. All the clocks used inside the TDC come from the PLL since they must be perfectly synchronized.

As shown in Figure 2.2, the TDC arranges 64 differential acquisition channels into 4 groups with the same data flow, which ends in a related readout FIFO with 8 differential output lines.

The Timing Unit in Figure 2.2 includes, for each input channel, the hit decoding and the first derandomizer logic. The hit decoder implies a 2-stage DLL to sample the hit signal for each time tap at each clock cycle, detecting the leading or trailing edge of the signal (the hit decoder can be configured to sample both leading and trailing edges or just one). In detail, the DLL works synchronously with the 1.28 GHz clock, and its first stage reaches a fine resolution of 12.2 ps (6 bits). Its second stage provides an additional interpolation within each first-stage delay cell, enabling a better resolution of 3.05 ps (2 bits). The final time measurement includes the DLL state (6 or 8 bits), a medium time measurement given by the final state of the PLL feedback divider (5 bits), and a coarse time measurement, found by a counter synchronous with the 40 MHz clock (13 bits). In the end, the total dynamic range takes 24 bits for a 12.2 ps binning and 26 bits for the 3.05 ps finest binning, while the full-scale range is 204.8 μs [27]. The Timing Unit also includes, for each channel, a local derandomizer that runs at the DLL frequency. Such a memory can stand up to 4 measurements, before being written into the next 512-word channel buffer synchronously with a 320 MHz clock. If the derandomizer is full, the next data will be ignored. Furthermore, the TDC allows a 781 ps minimum time between two consecutive time measurements; otherwise, the hit values written inside the channel

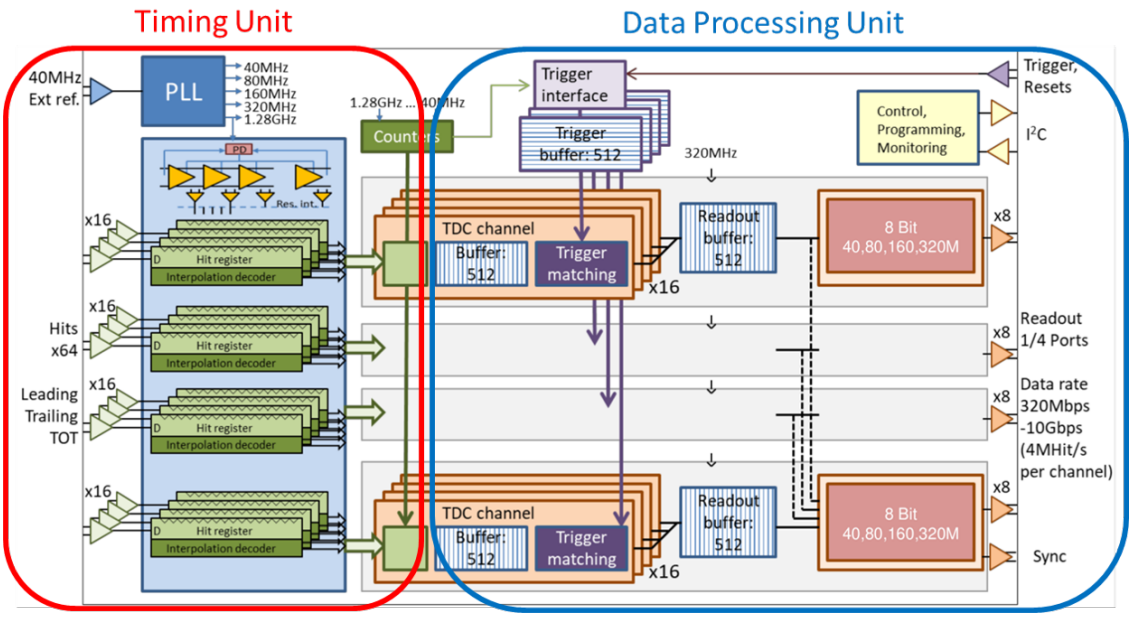


Figure 2.2: PicoTDC internal scheme architecture explaining from left to right both the Timing Unit (red) and the Data Processing Unit (blue) of the device.

buffers can be corrupted [27].

The Data Processing Unit is clocked with the generated 320 MHz frequency and arranges the following logic in 4 groups of 16 input channels. Each channel, as mentioned, stores its time measurements in a 512-word deep buffer waiting to be read out by the trigger-matching function of the related group. In detail, the trigger information is generated using the coarse counter, running at 40 MHz, at each trigger input signal. Considering each group, this information is stored in the related 512-word deep FIFO. Then, as for HPTDC, the PicoTDC can associate trigger signal information to the related hits measurements. Therefore, considering the TDC configuration, each group data can go through 2 processes:

- **The free-running mode**: the trigger-matching and the trigger input are disabled, so the data are written directly into the next readout buffer.
- **The trigger mode**: the trigger-matching function of each group uses the trigger information stored in the trigger FIFO, which includes a trigger time tag, an event ID and a bunch count ID. The matching is performed by selecting the hits related to a specific tag, within a configurable time window. Then all the data describing a triggered event are sent to the next readout buffer considering a particular data sequence.

Finally, the data of each group are stored in the corresponding 512-word deep readout FIFO, using a 32-bit format. All the group readout buffers can be read through 1 to 4 byte-wise[1] readout ports at a configurable rate of 320/160/80/40 MHz.

The PicoTDC can be configured and monitored via an I2C connection by accessing its internal registers, using 16-bit addresses. As I2C is a communication protocol with a master/slave structure, a master must address the desired connected slave using a specific

[1]Each port provides 8 differential lines.

7-bit address. Therefore, within its configuration and control routines, the PicoTDC acts as an I2C slave and its 7-bit address is settable via 7 external pins (I2C_ADDR[6:0]) [27].

2.1.2 Phase Locked Loop (PLL)

An analog PLL is a device capable of generating multiple clock frequencies in response to an input reference clock. As shown in Figure 2.3, this circuit compares the frequencies of the `clk_in` signal and an adjustable feedback signal, generated by the divider. When these two match in phase and frequency, we have a final steady state indicating that the PLL is locked and the desired frequency is produced.

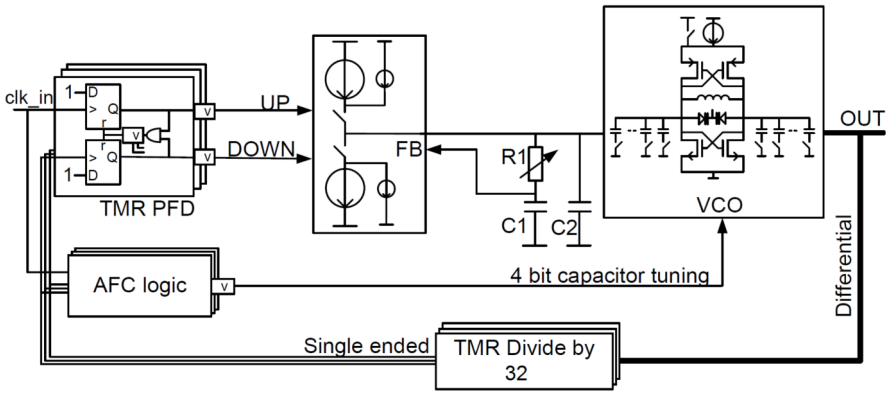


Figure 2.3: PLL block diagram [27].

Considering the scheme in Figure 2.3, the TDC PLL includes as the first object on the left a TMR PFD, which is a phase frequency detector built using the triple modular redundancy technique. In electronics the TMR technique generates reliable ASICs for critical environments, such as the one exposed to radiation, preventing ASIC malfunctioning. The PFD checks the difference in frequency and phase of the two inputs and activates a charge pump that supplies voltage impulses proportional to the discrepancy. As the second step, an L-C based Voltage Control Oscillator (VCO) is set to modulate the output frequency in response to an input voltage. Closing the loop, a TRM frequency divider provides feedback for the PFD and the AFC block (Automatic Frequency Calibration). The AFC logic is used as calibration for the VCO switchable capacitor to optimize the PLL performance in response to changes in temperature, power and voltage [28].

The PLL lock phase is obtained after ∼10 ms and its measured jitter is 340 fs [27].

2.1.3 Delay Locked Loop (DLL) and interpolators line

The analog DLL is the technological evolution of a delay line looped circuit. Specifically, it constrains the total delay buffer to be equal to an external clock period, using a phase detector followed by a charge pump that controls the voltage of each delay gate. This structure allows to cure the metastability of the delay line circuit caused by voltage and temperature variations. Then the hit signal is sampled inside a capture register for each time tap, leading to the time measurement with a resolution equal to the tap delay. To increase the measurement dynamic range without using too large delay lines, the loop structure and separated counters are exploited providing coarser measurements that must be added to the DLL one.

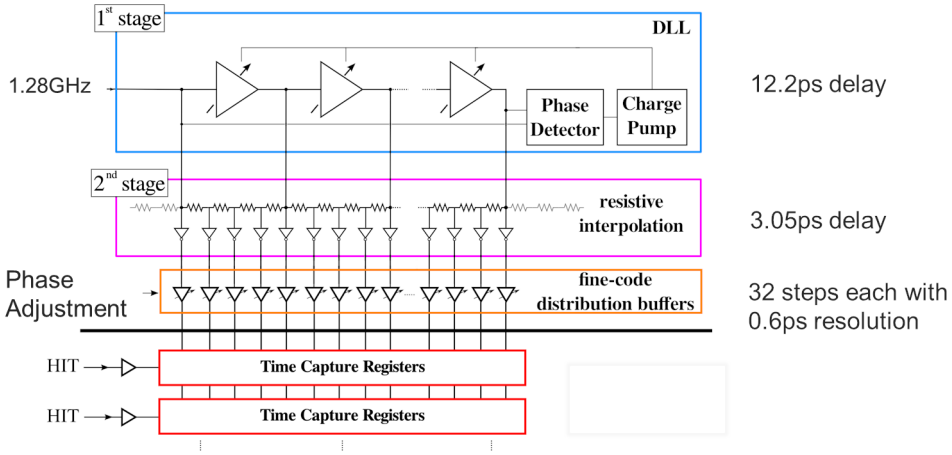


Figure 2.4: This DLL scheme explains its two main consecutive stages, showing the adjustment taps line before the capture register of two different channels [28].

The PicoTDC DLL, as shown in Figure 2.4, exploits a 64-taps delay line with a 1.28 GHz reference clock, reaching a resolution of 12.2 ps. The DLL also has a second stage line that arranges 4 R-(parasitic C) interpolation time taps at each delay step of the first stage. In detail, the total number of time taps becomes 256, reaching a binning of 3.05 ps. As explained, the final PicoTDC time measurement is the sum of all the values, found by the TDC coarser counters, and the DLL measurement, considering its coarse or fine binning.

In the DLL final stage, each delay time buffer disposes of a built-in adjust feature with a resolution of 0.6 ps to correct the possible mismatches [28]. The tap adjustment is done directly on the output of the first two stages and is used in the fine resolution mode (3.05 ps binning), requesting an individual calibration for each supplied TDC. Dividing the channels into 2 halves ([0-31] and [31-63]), it is possible to set all the adjustment tap values in a 2570-bit deep I2C register (0xFFFC) to configure each half independently [27].

The DLL must be initialized after the PLL lock state and its correct locking phase takes ∼10 ms.

2.1.4 The Data Processing Unit

As mentioned, the Data Processing Unit of the TDC architecture runs at 320 MHz and is built to implement buffering and trigger extraction features for each 16-channel group. In detail, each channel buffer takes the oldest hit in the related derandomizer and works until its 512 words-deep memory fills up. The buffer logic is implemented considering the TDC configuration, and provides [27]:

- **Paired measurement**: the TDC is configured to perform TOT measurement. Therefore, only one 32-bit word is stored in the buffer collecting the information of both the detected leading and trailing edges. The word format contains 16 or 19 bits to represent the leading edge of the pair and provides 8 or 11 bits for the width.

- **Single measurement**: the TDC is configured to perform time measurements of one or both the signal edges. Therefore, each measurement for a detected leading

or trailing edge is stored in the channel buffer considering a 32-bit word, in which 24 or 26 bits represent the measurement.

For single-channel calibration purposes, a configurable 26-bit offset can be added to the measurement before the channel buffer storing.

At this stage, the trigger matching function, implemented for each group, reads the time measurements at random access inside the corresponding channel buffers. As shown in Figure 2.5, each trigger input signal is associated, through the trigger interface, with a time tag found as the difference between the trigger signal arrival time and the configured latency width. This trigger time tag is decoded using the 40 MHz coarse counter and can be optionally subtracted to have values relative to the event of interest occurrence.

The buffered data are matched to the corresponding trigger time tag, (stored in the 512 deep-word triggers FIFO) considering a configurable time window. The data within the time window are selected as belonging to a specific triggered event and are written to the next readout FIFO, with header and trailer words. Considering the TDC configuration, the headers contain event information such as the Event ID, the Bunch ID, the trigger time tag, and status bits showing the nearly full TDC buffers. The trailer instead provides the same Event ID plus a word count (see subsection 4.3.2).

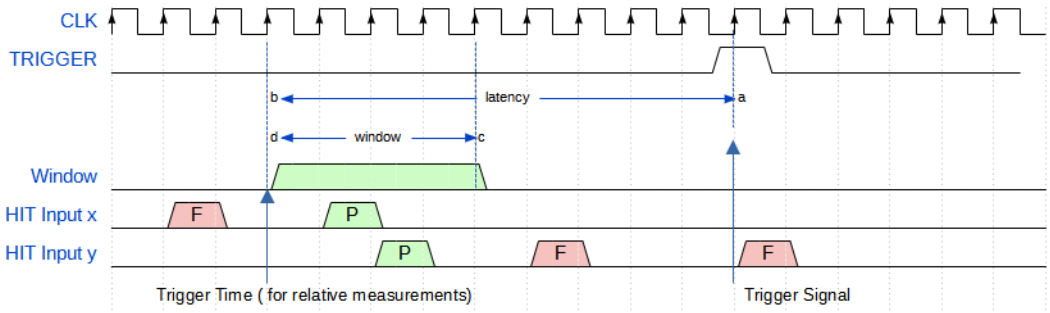


Figure 2.5: The trigger matching function scheme considers as processed (P) only the hits matching the window, while the others are called failed (F) [27].

The latency and time window lengths are defined in steps of 40 MHz clock cycles, as the trigger time tag is decoded through the 40 MHz coarse counter. The maximum configurable value for the latency is half of the maximum coarse count ($2^{13}/2 = 4096$ clock cycles) by construction. However, it is recommended to have a maximum trigger latency of $2^{11} = 2048$ clock cycles (51.12 μs) [27]. Furthermore, the trigger latency must be set longer than the matching window to prevent the matching function ends before all the related hits are written inside the channel buffers. The trigger matching function stops when newer hits arrive after the search limit or when no more data are available in the buffers. The hits, that do not match with the latest trigger, are removed from the channel buffers. If no trigger is waiting in the FIFO, the hits are also removed from the buffer, after a trigger latency plus one clock cycle, to prevent the logic from overflowing.

As the trigger FIFO runs full, the time tags are discarded, while the event counter keeps running to maintain synchronization with the data acquisition. All the logic and buffer memory implemented in the Data Processing Unit can be reset through a global reset, sent via an I2C command or a dedicated external pin.

This digital logic is properly designed for experiments and applications with large drift time or closely spaced triggers, since it can match detected hits to multiple triggers. Furthermore, the TDC provides an event counter and a bunch counter both running at

40 MHz, reproducing the LHC bunch structure. Used only in the triggered mode for header and trailer information, these two 13-bit counters can be reset to their configured offset through the rollover or as a command response (using the I2C bus or dedicated external pins).

The final readout phase is managed by each group, in which the readout priority is arranged in a round-robin fashion to guarantee a fair bandwidth between the 16 channels [27]. Each group is the input of a 512 words-deep FIFO that organizes the output data flow as 8-bit communication, running at a frequency of up to 320 MHz. As the readout FIFOs run full, the TDC can be configured to reject all the possible arriving events data or to block the trigger matching function, until new space in the readout FIFOs becomes available.

2.2 Board features overview

The board layout includes three main sections, as shown in Figure 2.6. Starting from the figure top-left, the power supply is the first section and provides all the specific voltages to power up each board component.

Looking at the bottom of the figure, the 2 PicoTDCs are linked to the PolarFire FPGA. The 2 ASICs can receive sub-LVDS signals through 2 FMC (FPGA Mezzanine Card) connectors, where other boards including the front-end electronics (discriminator and amplifier) and the related sensor might be plugged.

The last section, on the right, includes the possible connectors that interface with the integrated FPGA for different purposes. On the right of the FPGA, a JTAG connector and many I/O auxiliary facilities are provided for FPGA programming and reset. Furthermore, a USB-C link is provided through an FTDI chip or plugging a Cypress mezzanine, while an Ethernet link is managed instead through a PHY chip or exploiting an optical connector. This section focuses mainly on the Ethernet subsystem, showing its hardware features and design.

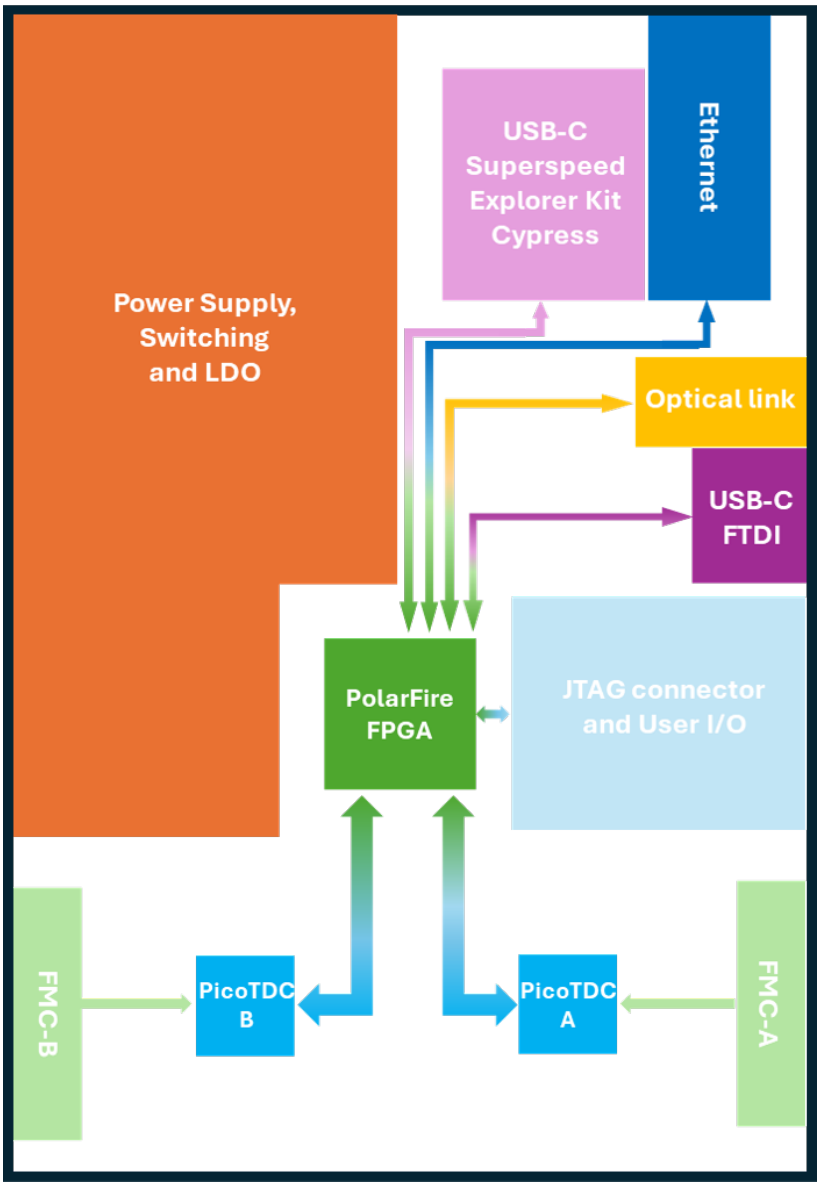


Figure 2.6: This PicoTDC board scheme shows its main components providing a subdivision into sections.

2.2.1 Power supply section

The board must be supplied with a voltage between 9 V and 14 V to reach the desired voltage level for each component. As the first control interface, the supply section includes a push button connected to three voltage monitors programmed to cut the circuit current until the button is pressed.

At that point the current flows into five switching converters, to generate the five independent voltage levels for the five different power rails of the board. Furthermore, the LDOs (Linear Dropout Regulator) generate other voltage levels along the lines, reducing the current noise caused by the switching inductance and dissipating more power in exchange.

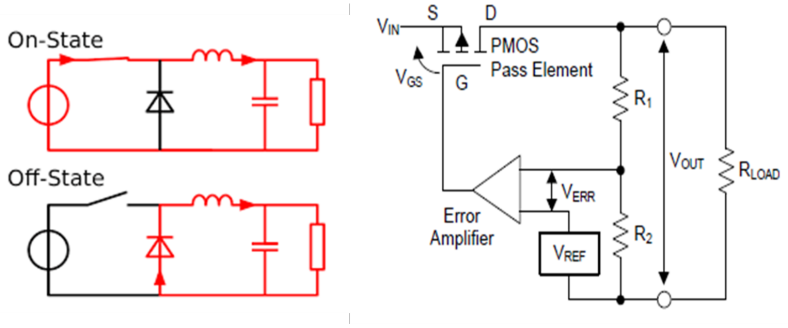


Figure 2.7: Simple circuits schematics of a buck converter and an LDO circuit [30].

Figure 2.7 shows on the left a simple schematic of the DC-DC buck converter circuit. It consists of a two-loop circuit designed to set a defined voltage drop on the load, using the inductance features of creating opposite voltage drops in response to current changes [29]. When the switch is closed, the current flows into the inductance and the diode is cut from the circuit. Then, in the open switch phase, the current flows only in the second loop since now the diode is forward biased and the inductance behaves as a current generator. The voltage on the load floats around a common value due to the coupling with the capacitor. Thus, working on the switch frequency, it is possible to obtain an almost constant voltage drop on the load for the whole time. Considering the buck converter continuous mode, the output voltage V_{out} as a function of V_{in} is defined by the duty cycle of the switch, which provides the D coefficient for the equation [29]:

$$V_{out} = DV_{in} \text{ with } 0 < D < 1. \tag{2.1}$$

The right scheme of Figure 2.7 represents the LDO (Low-Dropout Voltage regulator) circuit, which usually requires a transistor and a differential amplifier. The LDO senses any change in the output load resistance to provide a constant voltage at the regulator output [30]. The circuit uses the amplifier to compare a reference voltage V_{ref} and a fraction of the output voltage V_{out}, found using two resistances R_1 and R_2 in series. Its output considers any variation V_{ERR} between these two voltages and it is connected to the GATE of a PMOS transistor, used as a pass element. Then considering the V_{in} input, any voltage variation results in a change of the V_{GS} voltage that causes an increase in current I_{DS} through the transistor. This mechanism keeps the final output voltage at the constant value of:

$$V_{out} = V_{ref}\left(1 + \frac{R_1}{R_2}\right), \tag{2.2}$$

if V_{in} is high enough to keep the amplifier or the transistor out of saturation [30].

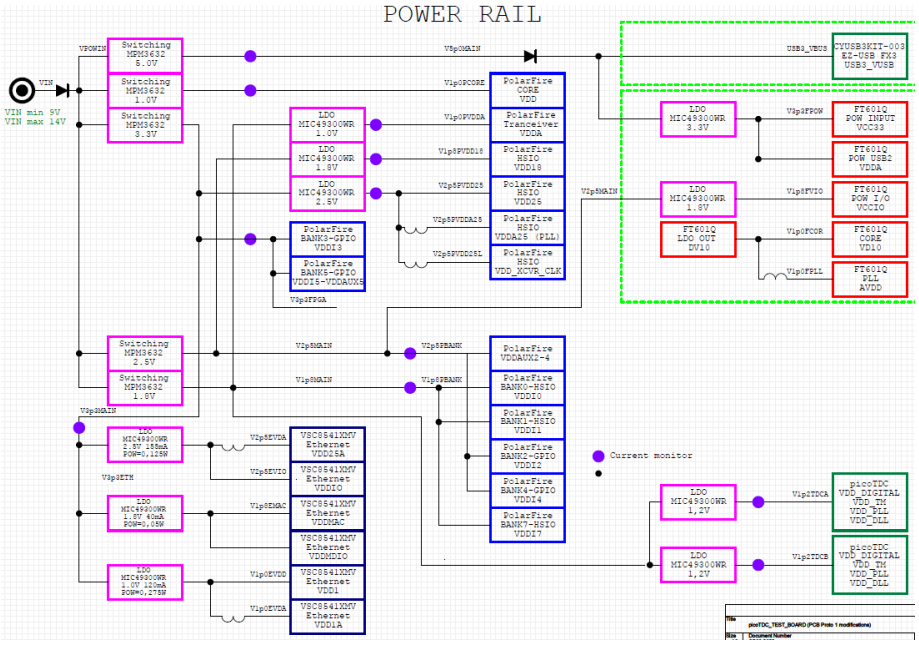


Figure 2.8: Voltage supply lines scheme.

Figure 2.8 shows how the power rails on the board are built, highlighting which components are linked along each line. Considering only the Ethernet subsystem for the connectors:

- The PHY chip (dark blue boxes) for the Ethernet is powered by three different lines on the 3.3 V rail, which exploits LDOs for 2.5 V, 1.8 V and 1.0 V levels.
- The FPGA banks, transceiver and power supply interfaces (light blue boxes) are powered using lines at 3.3 V, 2.5 V and 1.8 V; while its core is powered by the 1.0 V line.
- The two PicoTDCs (green boxes in the bottom-right corner) are linked to the 1.8 V line through 2 LDOs that keep their voltage at 1.2 V.

The LDOs and switching circuits are represented as pink boxes, while the USB electronics is shown in the top-right corner. In detail, the FTDI system is shown using the red boxes and the Cypress Mezzanine connector considers only the dark green box. An important observation regards also the power line of the FPGA bank 7, drawn in Figure 2.8. As mentioned in the following subsection, the PolarFire FPGA, hosted on board, provides a structure of only 6 banks. However, the FPGA power supply includes a further bank, without any I/O pins, that must be powered.

2.2.2 FPGA PolarFire

An FPGA (Field Programmable Gate Array) is a silicon chip that provides a user-programmable matrix of logic blocks to implement logic functions and algorithms. The one on board is a PolarFire MPF200T FCG784E, developed by Microchip Technology.

This device is a flash memory FPGA, which ensures low static power consumption and a reliable system even under radiation exposure.

This PolarFire FPGA mainly includes a logic fabric, a lot of user-programmable I/Os, a 16-lane transceiver communicating with the fabric, a security encryption co-processor and a system controller [31]. Microchip also offers soft IPs to implement communication protocols, such as Ethernet.

The fabric features 192K logic elements (4 LUTs and 1 FFD each) and 588 math blocks, which use an 18x18 MACC (Multiply-Accumulate) to create digital filters. Furthermore, the FPGA fabric integrates 4 different types of memories :

- LSRAM is a volatile memory, based on interleaving unit cells of 20 KBytes of static RAM with SECDED (Single Error Correction and Double Error Detection).
- μRAM is a smaller 64x12Byte deep RAM, implemented using latches and SEU (single event upset) immune capabilities.
- μPROM is a non-volatile memory, that is writable during programming and readable at runtime. By construction it is SEU-immune and for such a reason is very useful for instantiation of parametric and initialization data.
- sNVM features 56 KBytes of non-volatile memory which is readable and writable at runtime by a user service call, using the system controller. It is useful to initialize LSRAM and μRAM with secure data.

The configuration memory is SEU (Single Event Upset) immune, where the SEU is a signal instability inside the fabric due to radiation.

Considering the clock network, the FPGA provides a global network to route clocks and resets within large sections, with low skew, and regional networks that take clock domains only for limited silicon sections. As clock management tools it provides 8 DLLs and 8 PLLs, not considering the PLLs used for the transceiver lanes.

Up to 368 I/Os support both differential and single-ended I/O standards. As digital logic, for the I/Os to the fabric, the device includes I/O delay chains, registers and control logic for I/O modes. As shown in Figure 2.8, the FPGA PolarFire MPF200T FCG784 provides 6 different I/O banks hosting user I/Os that share the same VDDI power supply and voltage reference V_{ref} [32]. The FPGA banks are numbered from 0 to 5 and their I/O pins are instrumented for different tasks inside the firmware:

- **Bank-0** provides the HSIO[2] pins for Ethernet and USB-C FTDI connections. Furthermore, two pins are used to implement the I2C connection for the configuration of both PicoTDCs, exploiting a voltage I2C translator to interface with the ASICs that need logic-level signals with power rails at 1.2 V [33].
- **Bank-1** provides the HSIO pins directly linked to the Cypress mezzanine connector for a USB-C connection.
- **Bank-2** provides the GPIO[3] pins for the PicoTDC A connections.
- **Bank-3** provides fixed pins for FPGA programming via JTAG/SPI connection and global reset of the FPGA features.

[2]High Speed I/O
[3]General Purpose I/O

- **Bank-4** provides the GPIO pins for the PicoTDC B connections and the input of the on board 40 MHz differential clock.
- **Bank-5** provides auxiliary GPIO pins for clock and reset inputs. Furthermore, other pins are linked to JVS strips on board and used as firmware debugging facilities.

On the PolarFire east side, the 4 XCVR interfaces are supplied for transceiver communication from 250 Mb/s up to 12.7 Gb/s, with 4 lanes each [31].

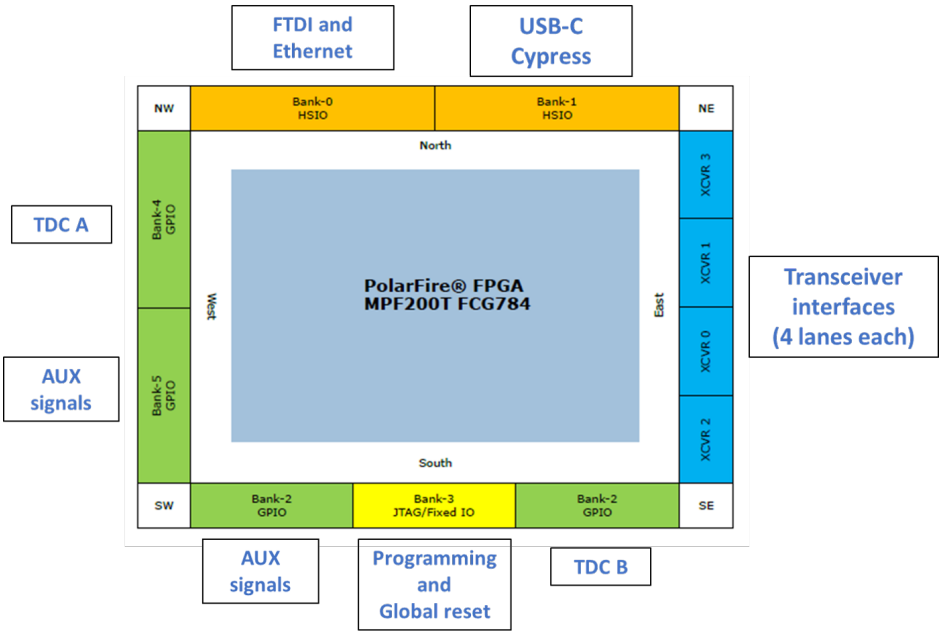


Figure 2.9: Banks locations inside the PolarFire MPF200T FCG784 FPGA, in which each bank is associated with a specific task implemented inside the firmware [31].

Furthermore, the FPGA provides a system controller based on a Cortex_M3 ARP internal processor and is used to ensure correct FPGA power-up and functioning, responding to the system service calls. The system service provides information on the FPGA state, allowing the user to call specific system controller actions.

For programming features, the FPGA PolarFire can behave as a slave or a master. In the first mode, the device's flash memory can be programmed via a JTAG connector or by using an SPI external master. The other mode allows the system control to check for an external SPI flash memory and update or reboot the firmware [31].

In summary, the PolarFire FPGA is designed for the best achievable power consumption using CMOS configuration cells, which provide power advantages over SRAM FPGA technology [31]. Furthermore, these kinds of FPGA were chosen over the SRAM ones for their immunity to Single Event Upset in the configuration bits.

2.2.3 Ethernet subsystem on board

An Ethernet network is a framework where each connected device can transfer data with the defined nodes. This communication is based on the OSI (Open Systems Interconnection) model shown in Figure 2.10, which provides a 7-layer structure with an increasing connection abstraction level from bottom to top [34]. Each node of an Ethernet

network offers such a layer structure and each layer relies on independent and transparent communication with the same layer of other nodes. Specifically, the OSI model aims to provide functions and information to the highest framework layer passing through all the previous ones.

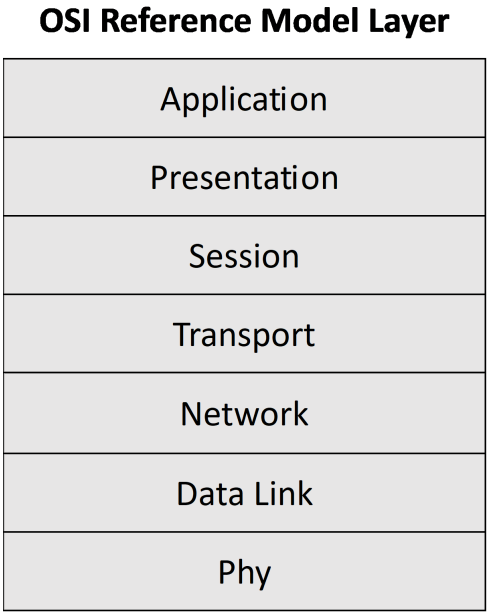


Figure 2.10: OSI model 7-layer structure.

The lowest abstraction layer of the OSI model is the Physical level (PHY), which corresponds to the device interface supporting the electrical communication over a physical connection with another device. Considering our case for a 1 Gb/s Ethernet communication, such a role is performed by the VSC8541-05 chip, which must be configured in a 1000BASE-T mode to allow a physical connection using a twisted pair Ethernet cable.

In general, within the framework of a node, the PHY communicates its information to the Data Link layer through an internal sub-layer called MAC (Medium Access Control). The MAC represents an implemented entity of each Ethernet node, which is responsible for the data transmission to and from the PHY chip [34]. In our case, the MAC is implemented inside the FPGA firmware using specific VHDL modules.

In our case, the Ethernet communication considers a full-duplex data transfer with a 1 Gb/s bandwidth, and the MAC/PHY interface can be implemented as [34][35]:

- **GMII (Gigabit Media Independent Interface)**: it considers a clock running at 125 MHz frequency, providing an SDR[4] data communication over 8 lines in both directions,
- **RGMII (Reduced Gigabit Media Independent Interface)**: it provides a DDR[5] data communication over 4 lines considering the same clock frequency implemented for the GMII interface.

[4]Single Data Rate.
[5]Double Data Rate.

- **SGMII (Serial Gigabit Media Independent Interface)**: it provides a DDR data communication over a single differential line, considering a 625 MHz differential clock. On board, this interface must be configured using the 1000BASE-X[6] mode via the transceiver connector, which bypasses the PHY chip supporting a direct connection with the FPGA transceiver interface. Therefore, the PHY/MAC interface is implemented directly inside the FPGA.

These three interfaces are shown in Figure 2.11, in which all the data and stream control signals are indicated for transmission and reception. Furthermore, the function of each implemented signal is explained in Tables 2.1, 2.2 and 2.3 respectively for GMII, RGMII and SGMII interfaces, in which the MAC side is used as the reference to assign signal direction.

Signal	Direction	Description
TXD[7:0]	Output	Transmitted data
TX_EN	Output	It indicates that transmitted data are available.
TX_ER	Output	It indicates the transmission of a data error.
RXD[7:0]	Input	Received data
RX_ER	Input	It indicates a received data error.
RX_DV	Input	It indicates a valid data received.
RX_CLK	Input	Clock used for the reception data stream, running at 125 MHz
GTX_CLK	Output	Clock used for the transmission data stream, running at 125 MHz

Table 2.1: GMII interface signals, taking the MAC side as I/O reference.

Signal	Direction	Description
TXD[3:0]	Output	Transmitted data
RXD[3:0]	Input	Received data
TX_CTL	Output	Control signal for transmitted data
RX_CTL	Input	Control signal for received data
RXC	Input	Clock used for the reception data stream, running at 125 MHz
TXC	Output	Clock used for the transmission data stream, running at 125 MHz

Table 2.2: RGMII interface signals, taking the MAC side as I/O reference.

[6]It considers a physical connection using fiber-optic cables and sometimes over shielded copper cables.

Signal	Direction	Description
TXD	Output	It provides in a single differential line the GMII TXD [7:0], TX_ER and TX_EN signals.
RXD	Input	It provides in a single differential line the GMII RXD [7:0], RX_ER and RX_DV signals.
RX_CLK	Input	Differential clock used for the reception data stream, running at 625 MHz
TX_CLK	Output	Differential clock used for the transmission data stream, running at 625 MHz

Table 2.3: SGMII interface signals, taking the MAC side as I/O reference.

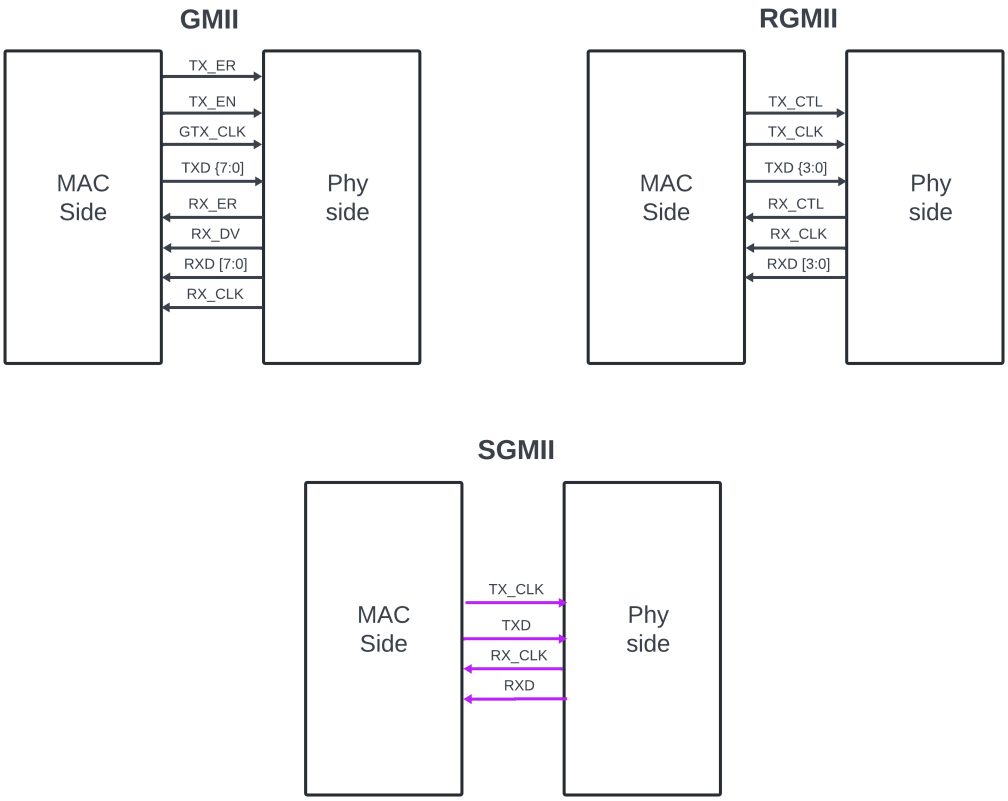


Figure 2.11: GMII, RGMII and SGMII data transfer interfaces, in which the differential and single-ended signals are shown respectively in purple and black.

The PHY chip placed on board implements the RGMII interface, which is built to produce a data rate equal to the GMII, using a reduced number of lines. Making a comparison between the two tables 2.1 and 2.2, the following characteristics of the RGMII signals are highlighted:

- The number of data lines in both directions is reduced from 8 to 4
- In data reception, the RX_CTL signal implements the functions provided by the two GMII signals RX_ER and RX_DV.

- In transmission, the TX_CTL signal implements the functions provided by the two GMII signals TX_EN and TX_ER.

As mentioned before, the RGMII uses DDR communication as it sends and receives words on both the rising and falling edges of the clock. This is done to keep the same GMII data rate using half of the data lines, as GMII transfers data only on the rising edge of the clock.

Figure 2.12 considers the RGMII interface for data transmission and reception. The first case on top represents data transmission, which uses the TXD[3:0] bus to send bytes stream from MAC to PHY, synchronously with TXC. RGMII splits the one-byte word sending its first 4 LSbs on the rising edge and the other four on the falling one. The TX_CTL signal considers on the rising edge the information of the GMII TX_EN while on the falling edge the information of TX_ER. On reception instead, the data are sent from PHY to MAC using the RXD[3:0] bus, with the same mechanism explained for transmission and synchronous with the RXC clock. The control signal RX_CTL behaves as the GMII RX_DV on the rising edge and RX_ER on the falling one.

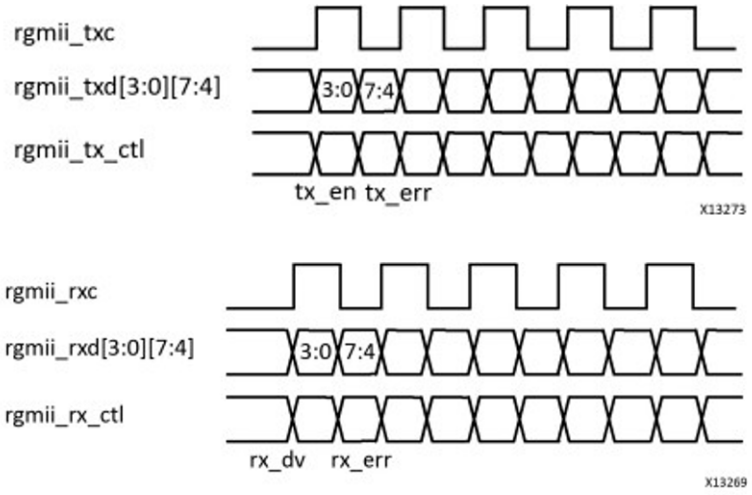


Figure 2.12: RGMII data transmit and receive operations between MAC and PHY [36].

The VSC8541-05 PHY chip is placed on board and it is directly connected to the FPGA. On the other hand, the chip is linked to an RJ-45 connector via a transformer, used to modulate the input and output voltage levels. This PHY device can be configured using the hardware strapping mode or writing its internal registers via its SMI (Serial Management Interface) interface [37]. The on board chip uses the first method and associates configuration instructions to some pull-up/pull-down logic values, sampled on the rising edge of the PHY NRESET input signal (active-low reset). To achieve a correct performance any device must not drive the external pin signals related to the pull-up and pull-down loads, until the NRESET deasserts. Therefore, NRESET must last at logic '0' for more than 2 ms to configure the chip properly. At the end, the chip is configured to provide RGMII and full-duplex communication, while supporting the 1000BASE-T connection mode.

2.3 The IPbus communication protocol

The IPbus (developed in 2009 by J. Mans et al.) is a communication protocol used to control different hardware features provided on a board, exploiting logic modules implemented inside the firmware of an FPGA. The protocol is based on a complete software-firmware suite to control a large network of devices. To address different endpoints along the network, the IPbus communicates over the Ethernet protocol.

2.3.1 Introduction to xTCA architectures and IPbus

The electronic systems, employed for data acquisition in particle physics experiments, were mainly built using the VME bus standard and characterized by crates based on a single controller bus topology. Other solutions have been implemented in recent years such as xTCA (Telecommunications Computing Architectures), in which each slave board is no longer passive but behaves as an independent endpoint communicating and interacting with the other network devices. The main net control can be distributed over all the devices, for a full mesh topology, or managed by one or two central hubs, for a single/dual star topology. The xTCA standards use the Ethernet as a communication protocol over the net, implementing a high fast-link of 1-12 Gb/s.

However, contrary to VME, this standard does not specify a communication technology to access the hardware memory of boards through external software applications. A hardware control system must be very reliable and predictable, as it is responsible for all the configuration, debugging, readout and control hardware routines. Furthermore, the system must be scalable to provide good performance from the simple scenario of one board to the final framework of a large experiment. For these reasons, the IPbus protocol was developed as an IP-aware hardware system that perfectly fits the firmware logic blocks provided by an FPGA [38]. Built on top of the Ethernet frames, IPbus is scalable since it is possible to use switches and routers to enlarge the net, and it is also very reliable since Ethernet allows many systems to be online for solving possible transaction failures.

A test system was set for a realistic network to investigate the possibility of using IPbus in the electronic system of the CMS experiment at CERN for LHC Run 2 in 2015 [38]. This test for one software client controlling one target device shows a transaction latency of 250 μs for single-word read/write transactions. Such a result is larger than the one for the VME one-word transaction. However, IPbus allows the concatenation of many words for higher efficiency in read/write block transactions, providing a throughput of 0.5 Gb/s with a data payload of more than 1 MB [38].

2.3.2 The on-chip IPbus protocol

The on-chip implementation of the IPbus protocol is based on the Wishbone SoC bus, a System-On-Chip design methodology providing a common interface between different IP cores [39]. The Wishbone standard was developed to improve the reusability and compatibility of IP cores, solving integration problems that limit the user in the development of System-On-Chip. This design provides a master/slave architecture in which each slave can be addressed by the master for data transfer, considering variable width, from 0 to 64 lines, for both data and address buses. The communication between master and slaves is clock synchronous and is controlled through a handshaking mechanism, which uses

specific and optional signals deeply explained in [39]. Among these handshake signals, the `strobe` and the `ack` are asserted respectively by the master for a valid data transfer and by the slave to end the data transfer.

Bus type	Signal	Direction	Width	Description
ipb_in	ipb_addr	Master to slave	32	Address bus
	ipb_wdata	Master to slave	32	Data to be written to slave
	ipb_write	Master to slave	1	It is asserted for a write cycle and deasserted for a read cycle.
	ipb_strobe	Master to slave	1	It qualifies address and data; its assertion marks start of cycle.
ipb_out	ipb_rdata	Slave to master	32	Data read from slave
	ipb_ack	Slave to master	1	Acknowledge flag: assertion marks the end of cycle.
	ipb_err	Slave to master	1	Error flag: assertion marks the end of cycle.

Table 2.4: IPbus protocol signals description [40]

Indeed, the on-chip IPbus supports a hierarchical topology design with a single master connected to multiple slaves, where data transfer relies on the 32-bit address and data buses. As the system works synchronously with a single reference clock running at an unconstrained frequency, within our implemented system the IPbus clock must run at 31.25 MHz to exploit all the available 1 Gb/s bandwidth defined by the Ethernet subsystem at a lower abstraction level.

The IPbus data transfer is implemented through the signals shown in Table 2.4, employed at different levels in the two data transactions defined in the IPbus protocol: the write cycle and the read cycle. Specifically, the system implements for both the bus cycles a common address bus `ipb_addr` and the two different data buses: `ipb_wdata` and `ipb_rdata`. As indicated in the Wishbone SoC standard for the `strobe` and `ack` signals, the master always starts a cycle asserting the `ipb_strobe` and the slave ends the transaction asserting the `ipb_ack` or the `ipb_err` in case of transaction error. Considering the Figure 2.13 from left to right, the 2 bus cycles are shown:

- **The write cycle**: the master addresses a specific slave through `ipb_addr` bus while the data are sent on the `ipb_wdata` bus, asserting both the `ipb_strobe` and `ipb_write` signals. Then the slave, after data reception, ends the transaction asserting the `ipb_ack` signal.

- **The read cycle**: the master sets the wanted slave address on the `ipb_addr` bus, asserting the `ipb_strobe` and keeping the `ipb_write` low. Then the slave sends the requested data on the `ipb_rdata` bus ending the cycle by the `ipb_ack` assertion.

For both transactions, the slave handshake allows one or more wait clock cycles before the assertion of the `ipb_ack` signal. Furthermore, the `ipb_strobe` and the `ipb_ack` signals must simultaneously deassert to start a new data transaction. The IPbus allows the master to tie high the `ipb_strobe` between two transactions, while it guarantees to

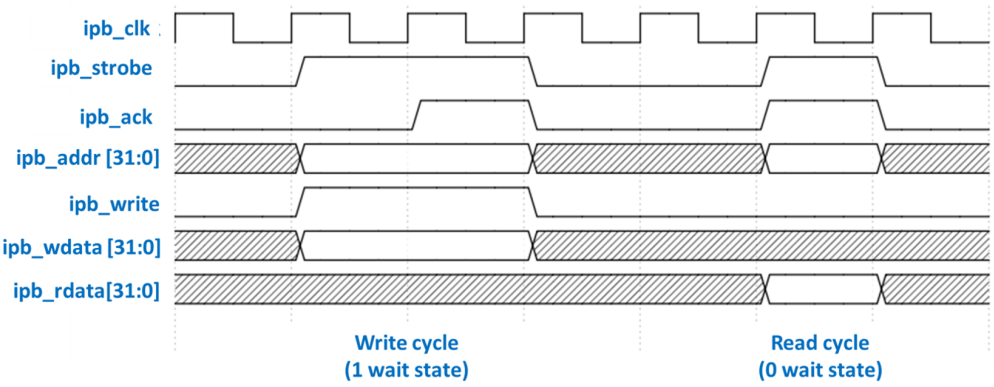


Figure 2.13: Scheme of an IPbus write cycle with 1 wait state and a read cycle with 0 wait states [40].

deassert the strobe on the clock cycle following the `ipb_ack`. On the other hand, the slave is allowed to tie `ipb_ack` to `ipb_strobe` if a zero-wait state is always possible.

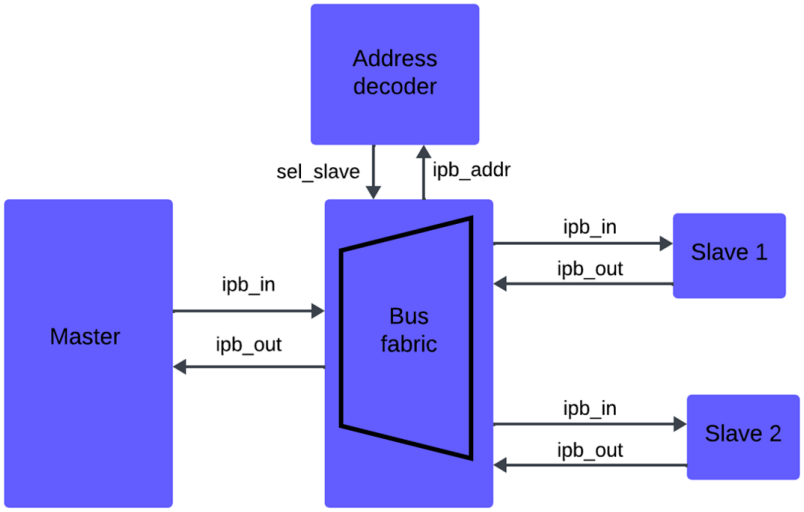


Figure 2.14: Scheme of the on-chip IPbus.

Figure 2.14 shows the IPbus system architecture, where the single master controls the slaves through a multiplexer interconnection, implemented by the `Bus fabric` and the `Address decoder` logic modules. The whole protocol is designed using point-to-point connections, where the buses `ipb_in` and `ipb_out` collect the Table 2.4 I/O signals, considering the slaves as reference. To start a data transaction, the master sends a specific slave address through the `ipb_in` connected to the `Bus fabric`, which uses the `Address decoder` to obtain the corresponding `sel_slave` identifier. Then the bus fabric enables the data transfer for the slave associated with the found `sel_slave`. Such a mechanism allows the slave to decode only its internal address space within the 32-bit IPbus address, leaving the decoding of the slave address position to the `Bus fabric`.

2.3.3 IPbus protocol at software level

As mentioned before the IPbus communication is built over the Ethernet protocol, more precisely it exploits the UDP/IP layered model to transfer data among a network of devices. Such a model is mapped approximately as the OSI one since it provides the 5-layer structure shown in Figure 2.15, in which the disposition of the layers shows a higher abstraction level from bottom to top. Each layer communicates only with its neighbors and provides information to the lower one, to build the final frame for data transfer across the network. Therefore, each layer defined for each network host accomplishes a different task as follows [41]:

- The **Physical layer** is the physical connection that drives the signals on the network, providing the bits communication.
- The **Link layer** builds the final frames moving across the network. It transfers data considering the physical (MAC) address, which uses a 6-byte width and characterizes each device.
- The **Network layer** generates the packets that must reach specific sub-nets or nodes of a network. It considers the 4-byte virtual (IP) address, which identifies each node.
- The **Transport layer** establishes the connection between applications on different hosts, using 2-byte port addresses.
- The **Application layer** generates the data and requests the connection for data transfer between different hosts.

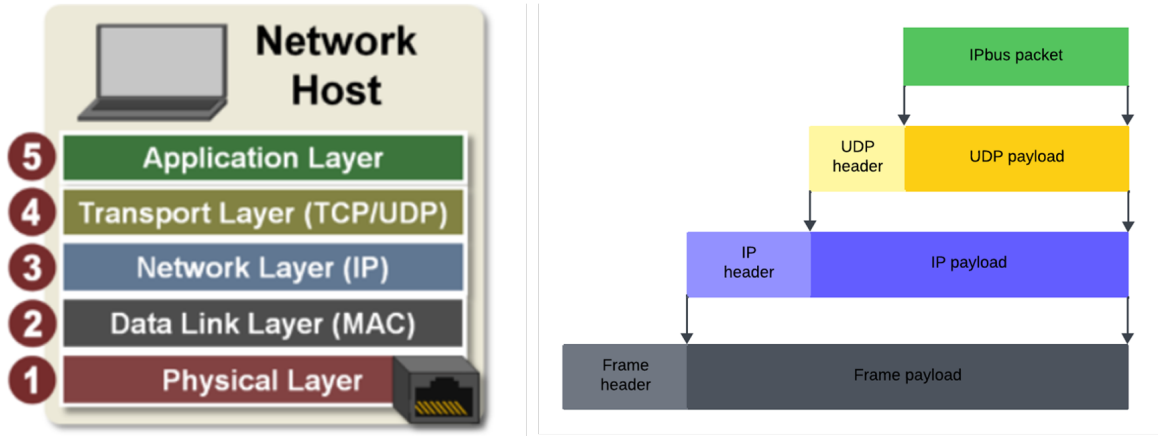


Figure 2.15: On the left the UDP/IP model structure[41] is shown, while on the right, there is a scheme for the Ethernet frame encapsulation, in which the color of each data packet is associated with the specific layer.

As shown in the right part of Figure 2.15, the IPbus protocol is defined within the Application layer, in which some software services construct a specific IPbus packet and ask for a connection with another host. Therefore, this scheme explains how the UDP/IP model builds the final Ethernet frame starting from the highest abstraction layer up to the Link layer. In detail, each data packet is built considering a corresponding protocol that uses a packet structure of two sections:

- The **header** contains both the source and destination addresses of the transaction plus some useful information that describes the protocol data packet.
- The **payload** defines the specific data packet of the lower layer, starting from the IPbus packet that includes the main data payload of the transfer.

The Link and Network layers manage the data through the Ethernet and the IP (Internet Protocol) protocols, while the Transport layer can be associated with one out of two protocols [42]: the TCP (Transport Control Protocol) or the UDP (User Datagram Protocol).

The TCP is a connection-oriented protocol, since it generates a bi-directional channel between the source and destination hosts before any data transfer begins. The channel closes when it is no longer necessary for communication. TCP provides handshaking mechanisms to re-transmit packets in case of transaction errors or connection congestion, ensuring a time-ordered data stream.

On the other hand, UDP is a connectionless protocol, as it does not need any channel generation to deliver data from one port to another. It provides the basic information for the Transport layer, mainly the source and destination port addresses plus a field for transaction errors. Therefore, it supports simple and fast communication in exchange for less reliability, as it just drops the packet in case of error or closes the connection in case of congestion.

In the IPbus framework, the UDP protocol was chosen to leave the complexity of error mitigation at the software level ensuring fast communication with the FPGA-implemented firmware. At the software level the final Ethernet frames are managed by the Control Hub application and the μHAL library, used at different levels to control the IPbus transaction between clients and targets.

Control Hub [38] is a client software application that provides arbitration mechanisms between multiple μHAL-designed applications. It communicates with each application using the TCP (Transmission Control Protocol) protocol to ensure a reliable connection and avoid undesired data stream stops, from and to the target. This application is designed for independent and efficient communication between multiple clients and targets.

The μHAL library [38] is the Hardware Access Library, which works as an end-user Python/C++ API to perform IPbus transactions. It is built to map the IPbus firmware implemented logic, allowing the concatenation of many client IPbus read or write requests to reach the best efficiency. Each μHAL application performs UDP data transfer to control addressed IPbus slaves implemented on board. Furthermore, the μHAL library and the IPbus protocol are designed to mitigate transaction errors during communication.

Each device-μHAL interface can run in local or remote mode, in which the client communicates with the target hardware using directly the UDP transport protocol or exploiting the Control Hub application.

2.3.4 Ethernet frame and IPbus packet structures

Considering the Physical layer of the UDP/IP model, the Ethernet frame, sent as a bit stream through a physical connection between two hosts, is shown in Figure 2.16. The host MAC sub-layer generates this frame structure, which provides two more fields besides the frame header and payload [43]. The first is the Preamble, which covers a length of 7 bytes with a repeated 0x55 pattern. The other represents the SFD (Start

Frame Delimiter), which shows the 0xD5 value. The MAC sub-layer uses these two to find or mark the Ethernet frame start during reception or transmission.

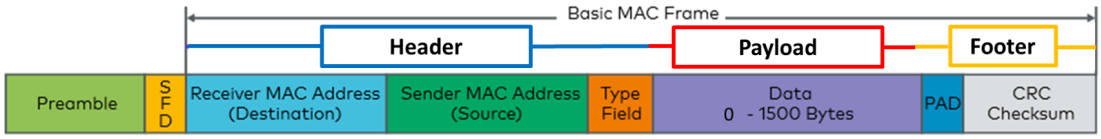


Figure 2.16: Structure of an Ethernet frame sent from one MAC source to another MAC destination.

Considering again Figure 2.16, it is possible to recognize the frame header and payload. The header takes a total length of 14 bytes and it implies two main regions. The first 12 bytes include the source and destination MAC addresses. Then, the Type field takes only 2 bytes and indicates the data payload length in bytes or the frame type in case of particular frame transitions. The payload provides instead the specific data content of the frames, with a length constrained between 0 and 1500 bytes. At the frame end, the MAC sub-layer can provide optionally the footer section which takes two more fields: the pad and the CRC. The pad sets a minimum frame length of 64 bytes, adding 0 to 46 bytes, with pattern 0x00, at the payload end. The CRC (Cyclic Redundancy Check) is an error-detecting code of 4 bytes, found using a cyclic redundancy algorithm that considers all the frame bytes. This is checked by the receiving Ethernet MAC, which decides whether to drop or not the frame.

Bits	0–3	4–7	8–15	16–18	19–31
0	Version	Internet Header length	Diff Service	Total Length	
32	Identification			Flags	Fragment Offset
64	Time to live		Protocol	Header Checksum	
96	IP source address				
128	IP destination address				
160	UDP source port (optional)			UDP destination port	
192	Length			Checksum (optional)	
224 +	Payload (IPbus packet)				

Figure 2.17: Structure of an IP packet, considering the UDP/IP model within the IPbus framework.

In the IPbus framework, the payload of the Ethernet frame includes the IP packet shown in Figure 2.17. This packet structure considers the IPv4 version and highlights

the bytes section related to different UDP/IP layers, using different colors. Indeed the blue region in the scheme describes the IP header, in which the 4-byte IP addresses are shown together with other information, such as [44]:

- **Version**: it provides the IP version (in this case it is 4).
- **Header Length**: it defines the header length, fixed at 20 bytes in this framework.
- **Diff Service**: it specifies the Differentiated services used in the data transfer.
- **Total Length**: it specifies the total IP packet length from 20 to 65535 bytes.
- **Identification**: it identifies the possible fragments of a packet.
- **Flags and Fragment Offset**: these are used in the fragmentation process of the IP protocol, with the Identification field. The fragmentation process is important in Ethernet communication as it splits large IP packets to transmit many Ethernet frames, without exceeding their maximum length. These fragments are then collected at the destination providing the full IP packet information.
- **Time to live**: this field indicates the lifetime of the IP packet, which is set to avoid the frame lying in the network without finding a destination.
- **Protocol**: it identifies the protocol used within the packet payload. Our case considers code 17 since it indicates the UDP transport protocol.
- **Header checksum**: it is an errors control field that scans the header but not the payload section of the packet.

The following IP payload includes the UDP frame, exploiting the same packet structure as the other layers. Within the scheme of Figure 2.17, the UDP header is shown in yellow and provides the basic information for the Transport protocol as explained in the subsection before. The UDP payload is shown in green and provides the final IPbus packet, built for reliable communication with the IPbus subsystem implemented inside the FPGA.

Indeed, the IPbus protocol provides a resend-request feature to overcome UDP packet drops. This feature is defined at the software level since the IPbus protocol relies on a request/response mechanism between the host client and the target device, in which the response may or not contain an error code. Considering the IPbus version 2.0, each data packet is arranged as the previous UDP/IP packets setting a header and a payload, which collects a series of response or request transaction packets with their specific header [45].

Figure 2.18 shows first the byte scheme of the IPbus packet header, which provides: the IPbus protocol version, a packet identifier to track the transactions, a special field to define the transmission byte order and the last field relative to the packet type. The transaction header provides instead the protocol version and an ID, which allows the client/target to track the specific operation. The total number of 32-bit words, within the transaction, is set using 8 bits, which limits the transaction size to a maximum of 255 words. Therefore, in the case of a transaction exceeding this limit, the packet is split across two or more transactions. Furthermore, the transaction header includes the IPbus transaction type (e.g. Read/Write) and another field "Info Code", which defines the data direction and the presence of error. Regarding these last 4 bits all successful responses have an "Info Code" of 0x0 and all the requests must have 0xf, while other values are used as response error codes [45].

IPbus packet header

31 24	23 16	15 8	7 0	
Protocol version	Rsvd.	Packet ID (16 bits)	Byte-order qualifier	Packet Type
0x2	0	0x0 – 0xffff	0xf	0x0 – 0x2

IPbus transaction header

31 28	27 16	15 8	7 4	3 0
Protocol Version (4 bits)	Transaction ID (12 bits)	Words (8 bits)	Type ID (4 bits)	Info Code (4 bits)

Figure 2.18: Schemes of the IPbus packet header and transaction header [45].

Executive summary

The PicoTDC board is a custom card designed by the INFN Bologna with two PicoTDC ASICs directly connected to a PolarFire FPGA. To test the feature of the PicoTDCs, the board supports a 1Gb/s Ethernet communication supplying an Ethernet connector linked to a PHY chip, which is RGMII interfaced with the FPGA. The software and the firmware implemented are based on the IPbus protocol, built over the Ethernet protocol as indicated by the UDP/IP model. The IPbus was designed to access the hardware feature on a board through software applications. In detail, the Ethernet protocol provides the information to move across a device network, while IPbus interfaces with specific firmware modules to control the addressed device.

Chapter 3

PicoTDC board firmware project

The firmware project was built using the Libero Soc software, developed by Microchip and the main design is based on an example firmware for a KC705 Xilinx board, provided by the CERN IPbus Git repository [46]. This architecture was then adapted to the features and IP cores, provided by Libero Soc for a PolarFire FPGA. The firmware employs the Ethernet connection to control an on-chip IPbus, designed for the configuration and readout of the PicoTDCs. Starting from an existing project [47], the IPbus slaves were developed to fit the features of the PicoTDC board.

VHDL is used as the main hardware descriptive language, defining a hierarchical topology as requested by the IPbus. As shown in Figure 3.1, using Libero SoC synthesis and implementation tools, the designed architecture requires a small percentage of the FPGA resources. After the firmware implementation, 180 I/O pins are locked and the 15.4% of the logic elements is employed. For what concerns the FPGA memory, almost the 50% of the LSRAM memory is used within the firmware, while the μSRAM occupied is negligible. Furthermore, to configure some IP cores, the sNVM memory is also used as shown in Figure 3.1, in which a page takes 236 bytes of memory.

This chapter explains the whole firmware architecture, but some small variations and simplifications on its structure are supplied to better understand the data stream. In each section VHDL modules are grouped based on the specific features they implement, not considering other modules less relevant to the chapter's purpose. The source code of the firmware project can be found within the repository in [48], which is evolving as new features are added or bugs are solved.

```
Design: C:\Users\arcadia\Desktop\PROJECT1\designer\picoTDC_top\picoTDC_top
Finished: Mon May 20 15:10:47 2024
Total CPU Time:     00:00:41            Total Elapsed Time: 00:00:29
Total Memory Usage: 3495.7 Mbytes
                        o - o - o - o - o - o - o

Resource Usage
+---------------+-------+--------+------------+
| Type          | Used  | Total  | Percentage |
+---------------+-------+--------+------------+
| 4LUT          | 26286 | 192408 | 13.66      |
| DFF           | 21832 | 192408 | 11.35      |
| I/O Register  | 0     | 362    | 0.00       |
| Logic Element | 29130 | 192408 | 15.14      |
+---------------+-------+--------+------------+

I/O Placement
+--------+-------+------------+
| Type   | Count | Percentage |
+--------+-------+------------+
| Locked |  180  | 100.00%    |
| Placed |  0    | 0.00%      |
+--------+-------+------------+

Completed writing pin report files.
```

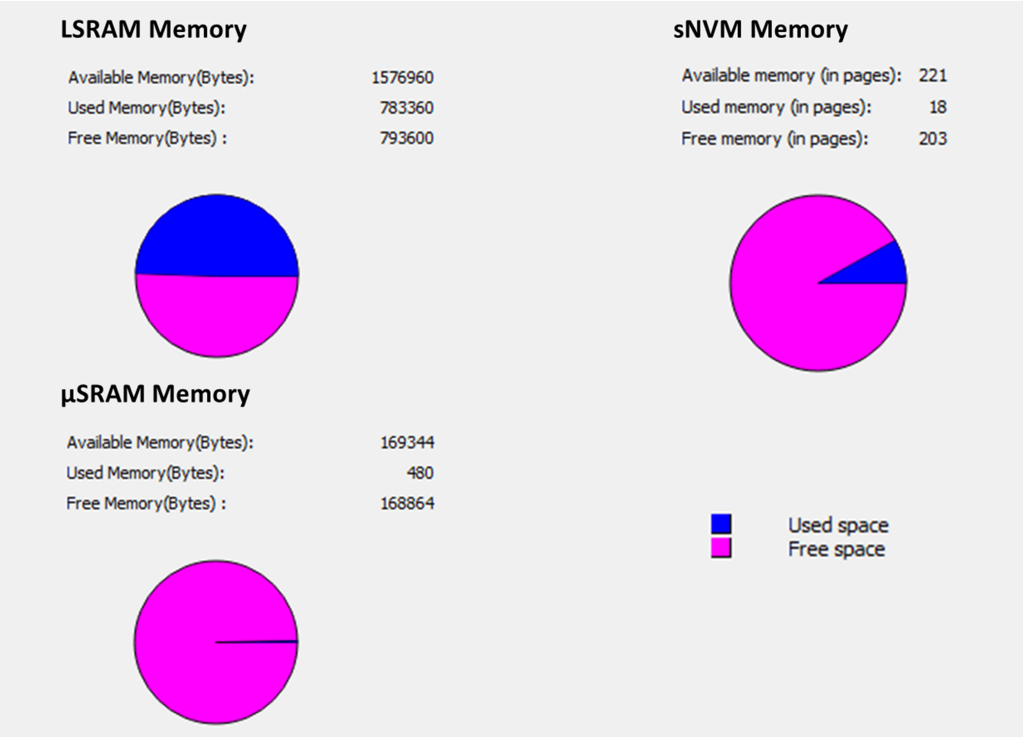


Figure 3.1: Libero SoC report of the FPGA resources employed, after firmware implementation.

3.1 Firmware structure

The firmware structure was designed to support the communication between the PHY chip and the PicoTDCs. In detail, the PHY chip manages the Ethernet data transfer with the FPGA, through the RGMII interface. As MAC and IP addresses of the board are hard-coded within the firmware, some VHDL structures are implemented to scan received frames and construct the transmitted frames. As mentioned, the Ethernet frame contains the IPbus packets used by the on-chip IPbus to perform IPbus transactions from and to the IPbus slaves, which are implemented to control the PicoTDCs and board operations.

The VHDL language supports a modular design, which means VHDL organizes specific logic functions and processes inside single units, called modules. This feature provides simpler designs since VHDL modules work independently inside their inner structure, but they have a specific purpose within a design.

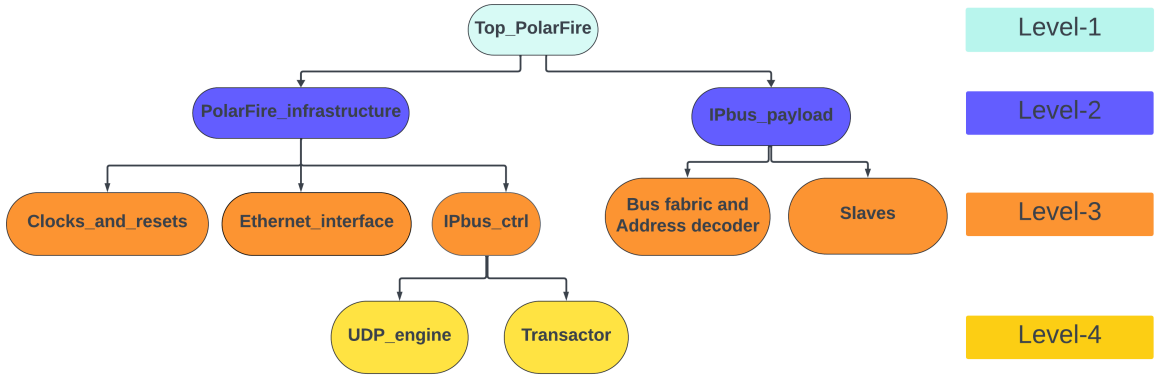


Figure 3.2: Hierarchical topology of the implemented firmware, with a specific color assigned to each level.

Therefore, starting from the I/O ports of the FPGA, the firmware architecture provides a hierarchical topology as shown in Figure 3.2, in which modules are organized by level and color:

- **Level-1**: it provides the first interface with the I/O pins employed for the firmware.
- **Level-2**: it supports the communication between the Ethernet and the IPbus subsystems.
- **Level-3**: it provides the features of the Ethernet or IPbus subsystem.
- **Level-4**: it collects all the modules implemented to accomplish a single Ethernet or IPbus subsystem feature.

Figure 3.2 does not show all the Level-4 modules of the firmware, but considers only the main two that build the connection between the `PolarFire_infrastructure` and the `IPbus_payload`. Figure 3.3 visually explains how each module, at the lower levels in Figure 3.2, is nested within the Level-1 `Top_PolarFire`, which is the interface that communicates with the PHY chip and the two PicoTDCs. Then the two Ethernet and IPbus subsystems correspond respectively to the `PolarFire_infrastructure` and the `IPbus_payload` modules, nested inside the `Top_PolarFire`. These two modules

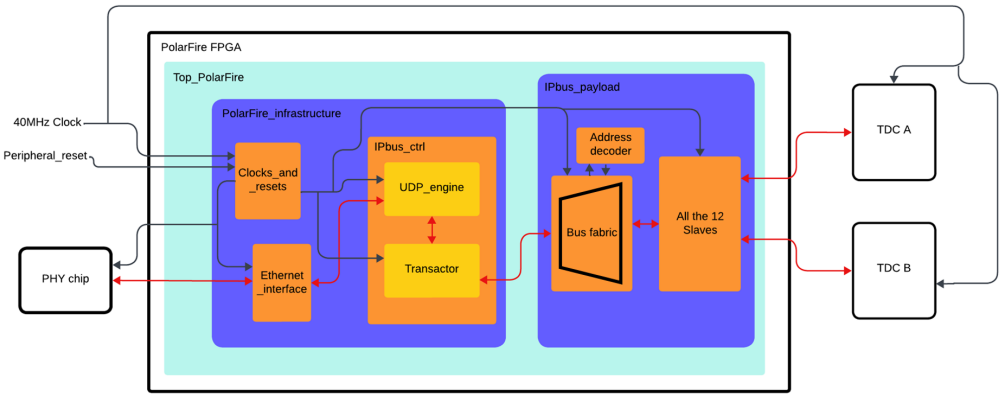


Figure 3.3: Firmware architecture showing the hierarchical topology and the modules contained in the Level-1 `Top_PolarFire` module. The grey lines show the reset and clock network, while the red ones indicate the data stream.

share the resets and clocks network, indicated by the grey lines, and build the data stream, shown in red, between the PHY chip and the two PicoTDCs.

The `PolarFire_infrastructure` is designed to manage the Ethernet frame transfer. Its inner structure includes three modules that cooperate to manage the data stream between the PHY chip and the `IPbus_payload`:

- `Clock_and_Resets`: it receives as input the `Peripheral_reset` external signal and the 40 MHz on board clock, which also feeds the 2 TDCs. It generates the resets and clocks network used within all the firmware.
- `Ethernet_interface`: it manages the Ethernet frames traffic with the PHY chip, scanning for transaction errors.
- `IPbus_ctrl`: it manages the IPbus data transfer with the `IPbus_payload` through the `Transactor` internal module. The `UDP_engine` communicates with the `Transactor` as a bridge between the `Ethernet_interface` and the IPbus subsystem.

The `UDP_engine` is designed to manage and control all the Ethernet frame data packets, corresponding to the 5-layer UDP/IP model. This module is responsible for controlling each packet considering the MAC (00-0c-29-7d-ae-c7), IP (192.168.200.32) and UDP (50001) addresses of the board, which are hard-coded within the firmware.

Therefore, the `UDP_engine` is responsible for IPbus transaction packet transfer with the `Transactor` module, which generates the IPbus read or write cycles within the on-chip IPbus. Considering the `Transactor` as the IPbus master, the connection with the `IPbus_payload` builds the IPbus system explained in subsection 2.3.2. Indeed, as shown in Figure 3.3, `IPbus_payload` module includes the link between the `Bus fabric` and all the 12 slaves modules. Only the connection between slaves and TDCs is shown for simplicity, while the IPbus slaves also provide other functionalities to control the board system.

3.1.1 PolarFire_infrastructure overview

As already mentioned, the internal structure of the PolarFire_infrastructure includes 3 modules that must work together to manage the Ethernet traffic information and control the IPbus_payload module.
Analyzing each internal module individually, the Clock_and_Resets generates the needed clock frequencies and resets to regulate the operational speed of the other two modules. As shown in Figure 3.4, its internal structure includes some Level-4 modules, such as two PLL cores and the Reset_module.

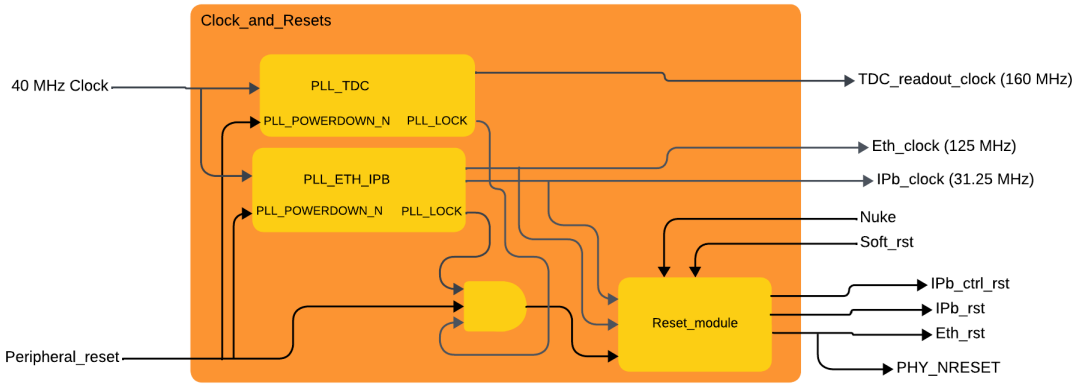


Figure 3.4: Internal structure of the Clock_and_Resets module, which includes from left to right two PLLs cores and the Reset_module block.

The Libero IP catalog provides the PLL cores, which exploit the built-in PLLs of the FPGA fabric [49]. Using the input clock running at 40 MHz, each core was configured to generate clocks feeding different firmware subsystems:

- PLL_TDC: it generates the TDC_readout_clock, which runs at 160 MHz. This clock feeds the IPbus slaves responsible for the TDCs readout, so they can correctly sample the data coming from the TDCs. Therefore, the PLL_TDC core is configured to produce this output clock in phase with the 40 MHz input clock, which is also distributed to the PicoTDCs to generate the 160 MHz readout rate.

- PLL_ETH_IPB: it generates the clocks respectively used within the Ethernet subsystem and the IPbus subsystem:

  - Eth_clock: it runs at 125 MHz to allow a 1 Gb/s bandwidth, as required by the RGMII specifications (125 MHz $\times$ 4 bits = 1Gb/s). Furthermore, such a clock is used for the Ethernet frame data transfers.

  - IPb_clock: it runs at 31.25 MHz and feeds the on-chip IPbus. This frequency is requested by the IPbus specification to match a 1 Gb/s Ethernet bandwidth, as the IPbus works through a 32-bit data bus and the Ethernet interface of the UDP_engine is based on an 8-bit bus running at 125 MHz (125 MHz/4 = 31.25 MHz).

Furthermore, the PLL core features other two signals that control its functioning: the PLL_POWERDOWN_N and the PLL_LOCK. The PLL_POWERDOWN_N input signal is active-low and is fed by the Peripheral_reset, which is also active-low. This signal keeps

the PLL in reset mode, when asserted, and stops the frequency generation. Therefore, the `Peripheral_reset` is linked to a button on board, which instantly stops the data transactions when pressed. The `PLL_LOCK` is an active-high output signal, which indicates the steady state of the generated clocks. Both the `PLL_LOCK`s and the `Peripheral_reset` signal are used as input for an AND logic port to generate an input signal for the `Reset_module`. The resulting signal is asserted whenever the `Peripheral_reset` is asserted or one PLL is not locked.

The `Reset_module` is designed to asynchronously assert all the reset output signals shown in Figure 3.4, when the AND output signal is asserted. Otherwise, the `Nuke` and `Soft_rst` signals, generated by the IPbus slaves, are sampled considering respectively the `Eth_clock` and `IPb_clock` rising edges. When `Nuke` is asserted all the output reset signals are asserted, while the `Soft_rst` only controls the `IPb_rst`. In particular, considering the black arrows in Figure:

- `IPb_rst` is the reset within the `IPbus_payload` module.
- `IPb_ctrl_rst` is a reset used within the `IPbus_ctrl` module.
- `Eth_rst` is the reset within the all Ethernet subsystem (it is used within all the `PolarFire_infrastructure` module).

The `Eth_rst` signal is designed to be asserted for ∼4 ms. Therefore, the `PHY_NRESET`, sub-net of the `Eth_rst`, works with other signals to generate the PHY chip reset, used during the hardware strapping configuration of the PHY chip.

The `Ethernet_interface` and the `IPbus_ctrl` are the main modules that define the link between the PHY chip and the IPbus subsystem. Figure 3.5 shows how these two modules manage the data stream, highlighting the used signals with red color. The convention used for data direction is defined by the `rx` and `tx` labels. All signals sent from the PHY chip to the `IPbus_ctrl` module are called `rx` as for reception, while the opposite direction is defined by `tx` term as for transmission.

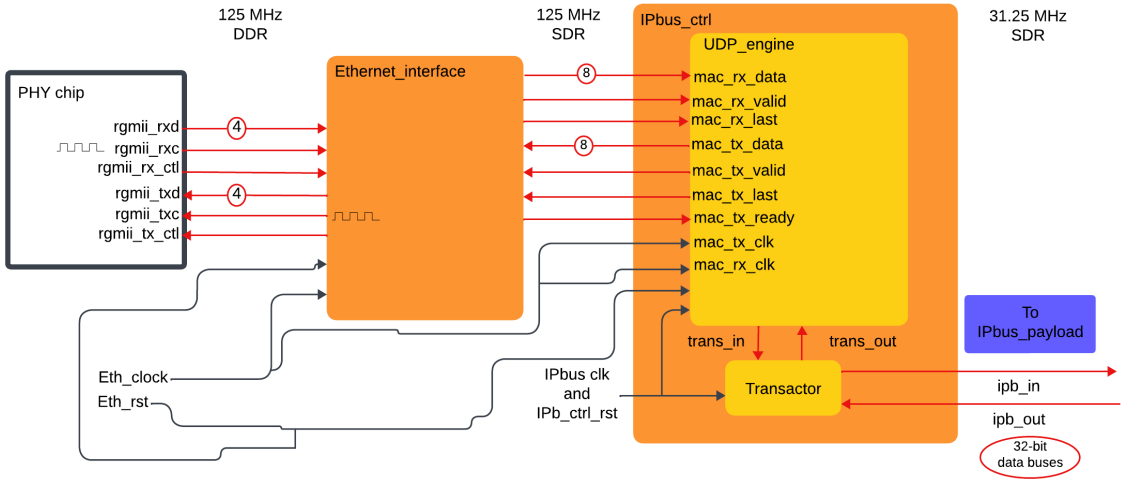


Figure 3.5: Connection between the `Ethernet_interface` and the `IPbus_ctrl` modules.

Figure 3.5 shows the RGMII interface provided by the `Ethernet_interface` module. As mentioned, the RGMII supports DDR data transfer using 4-bit buses, in both tx

Signals	Direction	Width	Description
mac_rx_data	Input	8	Data bus used for reception
mac_rx_clk	Input	1	Reception clock running at 125 MHz
mac_rx_valid	Input	1	It indicates the valid received Ethernet frame.
mac_rx_last	Input	1	It indicates the last byte of the received Ethernet frame.
mac_tx_data	Output	8	Data bus used for transmission
mac_tx_clk	Output	1	Transmission clock running at 125 MHz
mac_tx_valid	Output	1	It indicates the valid transmitted Ethernet frame.
mac_tx_last	Output	1	It indicates the last byte of the transmitted Ethernet frame.
mac_tx_ready	Input	1	It indicates if the Ethernet_interface module is ready to receive a frame.

Table 3.1: I/O signals for the IPbus_ctrl module considering its interface with the Ethernet_interface.

and rx directions, considering a 125 MHz clock. Within the Ethernet_interface, all the RGMII signals are sampled using the PLL-generated Eth_clock for transmission or the rgmii_rxc (sourced by the PHY chip) for reception.

Considering the interface between the Ethernet_interface and the IPbus_ctrl modules, the data transfer is based on SDR communication using 8-bit data buses synchronous with 125 MHz clock signals. The I/O signals for the IPbus_ctrl module are shown in Table 3.1, in which the mac_rx_clk and mac_tx_clk clocks are both fed using the generated Eth_clock.

Figure 3.5 also provides the IPbus_ctrl internal architecture, which manages the Ethernet frame traffic for IPbus packets or other applications, as explained in the following sections. Considering an IPbus packet transfer within an Ethernet frame, the UDP_engine internal module, supplied by the CERN Git repository, manages each UDP/IP 5-layer model packet. For what concerns the reception, the UDP_engine receives, on the mac_rx_data bus, an Ethernet frame scanned to search for the IPbus packet. Then considering the PLL-generated IPb_clock, this module sends to the Transactor all the Ethernet frame information, on the trans_in bus explained in Table 3.2. The Transactor searches for IPbus transaction packets and builds signals to start IPbus read or write cycles in the connected IPbus subsystem. A similar mechanism is defined in transmission, in which the Transactor module provides, on trans_out bus of Table 3.2, specific information related to each IPbus response or read transaction. Then UDP_engine module uses this information to build the corresponding IPbus packet with all the related transaction packets, nested within an Ethernet frame. The Ethernet frame construction is performed considering each UDP/IP model layer through the stored information of the corresponding source and destination addresses. Finally, the obtained frame is sent to the Ethernet_interface through the mac_tx_data bus.

Bus type	Signal	Direction	Width	Description
trans_in	pkt_ready	Input	1	It indicates a received Ethernet frame containing IPbus packet.
	rdata	Input	32	Data bus for received Ethernet frame
	busy	Input	1	It indicates the busy state of the UDP_engine for reception.
trans_out	raddr	Output	16	Index of the Reception RAM register to provide the corresponding 32-bit word on the rdata input bus
	pkt_done	Output	1	It indicates the construction of a transaction packet.
	we	Output	1	It indicates a valid output transaction packet.
	waddr	Output	16	Index of the transmission RAM register to store the 32-bit words, sent on the wdata bus
	wdata	Output	32	Data bus for transmitted IPbus transaction packets

Table 3.2: Transactor I/O signals considering the interface with the UDP_engine module.

3.1.2 IPbus_payload overview

Figure 3.6 shows the implemented IPbus subsystem, providing the connection between the `Transactor` and the `IPbus_payload` internal modules. The data stream, shown in red, provides the same bus format explained in subsection 2.3.2 up to each IPbus slave. Therefore, the `Transactor` acts as an IPbus master which uses the `ipb_in` and the `ipb_out` buses to manage the read and write IPbus cycles, synchronous with the `IPb_clock` running at 31.25 MHz.

The on-chip IPbus is based on a multiplexer mechanism, which enables the `ipb_in` or the `ipb_out` buses for the addressed slave, using the information provided on the `ipb_addr` 32-bit bus. Such a job is managed by the `Bus fabric`, which acts as the interface between the `Transactor` and all the IPbus slaves. At the beginning of every IPbus cycle, the `Bus fabric` evaluates the requested slave address, using the `Address decoder`. Specifically, the `Address decoder` works as a look-up table, in which each possible slave address is associated with a corresponding `sel_slave` integer value from 0 to 11 since the implemented on-chip system counts a total of 12 IPbus slaves (how each slave and their internal registers are addressed is explained in section 3.3).

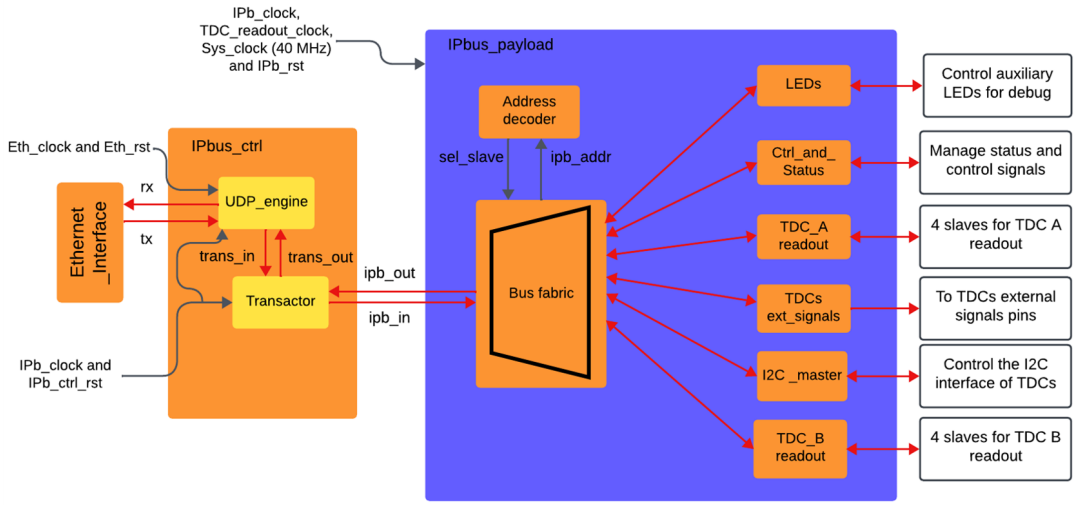


Figure 3.6: On-chip IPbus implemented within the firmware, in which the slaves are connected to their specific feature and the data path is shown in red.

Figure 3.6 shows all the 12 slaves implemented, specifying their following connections and features within the on-chip and off-chip systems:

- `LEDs`: it controls some auxiliary LEDs hosted on board, which can be used for debugging the IPbus subsystem.
- `Ctrl_and_Status`: it manages some status and control signals within the firmware. Using IPbus commands, the `Eth_rst` and `IPb_rst` signals can be checked while `Nuke` and `Soft_reset` signals, for system reset, are controlled.
- `I2C_master`: it works as the I2C master for the configuration of both TDCs.
- `TDC_Areadout` and `TDC_Breadout`: they include 4 slaves each, to implement an independent readout interface for each of the 4 readout ports featured by the 2 chips.

- `TDCsext_signals`: it manages, by IPbus command, the reset and trigger signals of both the TDCs.

Each slave module in the figure shows for simplicity a bidirectional link with the `Bus fabric`, meaning for a 2 buses connection: `ipb_in` and `ipb_out`. As the `TDC_Aread-out` and `TDC_Breadout` include 4 IPbus slaves each, the corresponding bidirectional arrow indicates 4 connections.

3.2 The Ethernet frame path

The FPGA and PHY chip communication must rely on a VHDL module implementing the MAC sub-layer functionalities, described in subsection 2.2.3 within the OSI model framework. The `Ethernet_Interface` module acts as the UDP/IP Data Link layer of the PicoTDC board since it is designed to check and control Ethernet transactions at a low abstraction level. This block provides an RGMII interface directly linked to the PHY chip and drives the Ethernet frames up to the `IPbus_ctrl`, which is responsible for checking the correct addressing of the board among a device network. As already mentioned, the `IPbus_ctrl` has an internal structure that manages each Ethernet data packet and, besides the IPbus facilities, it supports many other common utilities useful in an Ethernet network of devices, as explained in the chapter.

In summary, the `Ethernet_Interface` is designed for full-duplex Ethernet communication and exploits a full bandwidth of 1 Gbit/s, through the RGMII interface. These features match the PHY chip configuration. The Ethernet frame is provided to the `IPbus_ctrl` module that evaluates each packet and consequently acts.

3.2.1 The Ethernet_interface

The PolarFire FPGA supports a 1 Gb/s Ethernet solution using IP cores defined inside the Libero IP catalog [50] and instantiated inside the `Ethernet_interface` module as shown in Figure 3.7. The architecture works synchronously with the `Eth_clk` and the PHY chip `rgmii_rxc` clocks, which both run at 125 MHz.

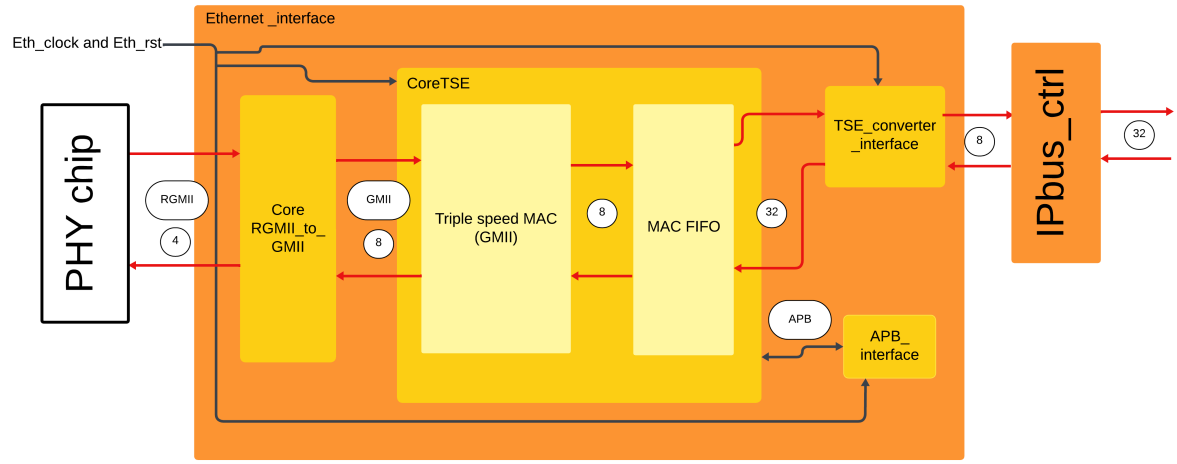


Figure 3.7: Internal structure of the `Ethernet_interface`, where the data path is highlighted in red.

The core provided by Microchip for building the UDP/IP Data Link layer is called CoreTSE. This core contains, as an internal module, the Triple speed MAC, able to work at 10, 100 or 1000 Mb/s considering its configuration. In a general Ethernet connection, the MAC (Media Access Controller) is a compulsory element that filters the Ethernet frames and manages the outgoing or incoming Ethernet transactions. The CoreTSE also includes the MAC FIFO module used to mitigate transition errors and provide data queueing for high data throughput, during transmission and reception.

Considering the Figure 3.7, the RGMII_to_GMII IP core [51] is placed on the left as the interface between the PHY chip and the CoreTSE. Since CoreTSE provides a GMII MAC, RGMII_to_GMII core is necessary to convert the PHY chip RGMII signals. The same issue is found in the CoreTSE interface for the IPbus_ctrl since the MAC FIFO arranges data transactions through 32-bit buses, while the IPbus_ctrl uses only 8 lines data buses in both directions. Therefore, the TSE_converter_interface module is designed as a medium interface to match the IPbus_ctrl signals (in table 3.1) and the CoreTSE interface, shown in Table 3.3. This module works synchronously with the Eth_clock and provides internal processes for data transfer, able to convert data bus width and control signals.

The last module instantiated within Ethernet_interface is the APB_interface, which uses internal IP cores to provide an APB protocol (Advanced peripheral bus) communication with the CoreTSE. In detail, whenever the Eth_rst signal deasserts, the APB_interface module configures the CoreTSE internal 32-bit registers. The CoreTSE must be configured to provide a GMII full-duplex Ethernet communication, running at 1 Gb/s.

Transaction	Signals	Direction	Width	Description
MRX: from CoreTSE to IPbus_ctrl	MRXCLK	Input	1	Reception clock running at 125 MHz
	MRXACPT	Input	1	It indicates when the CoreTSE is ready to receive data.
	MRXRDY	Output	1	It indicates a received valid frame.
	MRXSOF	Output	1	It marks the first received frame word (Start Of Frame).
	MRXEOF	Output	1	It marks the last received frame word (End Of Frame).
	MRXDAT	Output	32	Received frame data bus
	MRXBYTEVALID	Output	2	It indicates the number of MSBs (Most Significant Bytes) not valid for the last received frame word.
MTX: from IPbus_ctrl to CoreTSE	MTXCLK	Input	1	Transmission clock running at 125 MHz
	MTXACPT	Output	1	It indicates when the CoreTSE is ready to transmit data.
	MTXRDY	Input	1	It indicates a transmitted valid frame.
	MTXSOF	Input	1	It marks the first transmitted frame word (Start Of Frame).
	MTXEOF	Input	1	It marks the last transmitted frame word (End Of Frame).
	MTXDAT	Output	32	Transmitted frame data bus
	MTXBYTEVALID	Output	2	It indicates the number of MSBs (Most Significant Bytes) not valid for the last transmitted frame word.

Table 3.3: Table showing the I/O signals of the CoreTSE, considering its interface for the IPbus_ctrl module.

3.2.2 The CoreTSE IP core

In a network with several devices, an Ethernet frame sent from node to node provides a structure that must be recognized by the receiving Ethernet MAC of each system. In our case, such a structure is defined within the UDP/IP model and it is explained in subsection 2.3.4, providing Preamble and SFD fields followed by the Basic Ethernet frame (consisting of header, payload and footer).

During the Ethernet frame reception, the Ethernet `MAC` inside the `CoreTSE` scans the frame searching for the Preamble and the SFD. When the SFD is found, these two fields are stripped and the Basic Ethernet frame is sent towards the `IPbus_ctrl` module. On the other hand, for transmitting a frame to another node, the `MAC` adds SFD and Preamble fields to a Basic Ethernet frame built by the `UDP_engine` module, inside the `IPbus_ctrl`.

Furthermore, during data transfer, the `MAC` processes the whole Ethernet frame length looking for errors inside each field. For what concerns the transmission, the MAC configuration allows the construction of a CRC checksum, which is added, within the footer field reporting the found errors. Considering the reception and the `FIFO` configuration, the `MAC` asserts some error flags checked by the `MAC FIFO` to drop the bad frames.

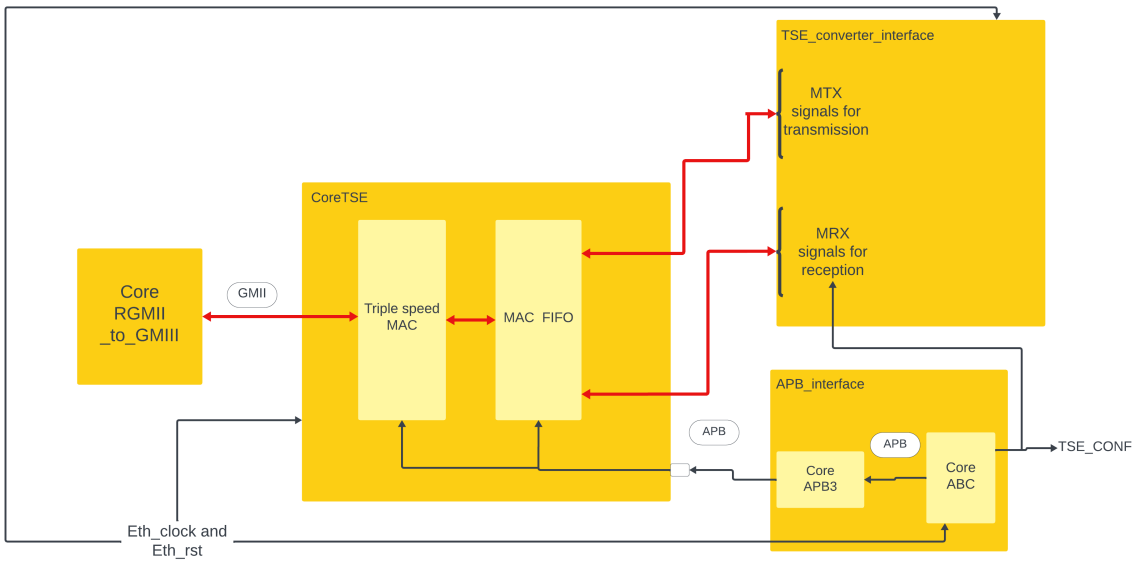


Figure 3.8: `CoreTSE` internal structure and connections, in which the APB link to the `APB_interface` module is shown.

In detail, the `CoreTSE` configuration is accomplished via two IP cores instantiated inside the `APB_interface` module, based on the AMBA-APB[1] to access the internal 32-bit registers of the `CoreTSE` (with corresponding 10-bit addresses) [52]. The APB (Advanced Peripheral Bus) protocol is described within the specification [53] and provides a simple synchronous communication between a Requester and a Completer. Such a protocol builds a bridge between the Requester and many Completer modules, representing a main processor and peripherals. The APB protocol provides a unique address bus and 2 data buses, one for each direction, with configurable widths (they consider a maximum of 32 bits), to read and write specific registers of connected APB peripherals.

[1] Advanced Micro-controller Bus Architecture.

As shown in Figure 3.8, the APB_interface design relies on the CoreABC and CoreAPB3 cores connection. These two act respectively as the Requester and the APB bridge for the CoreTSE APB interface, which is the only APB Completer.

The CoreABC (APB Bus Controller) [54] is a configurable microprocessor, that supports many options settings to define its APB interface and I/O signals. Considering our case, its APB interface provides a 16-bit address bus and 32-bit data buses, while among the I/O pin-out only the TSE_CONF output signal is enabled. As a microprocessor, such a core can perform a list of operations defined within an internal memory that must be configured. Therefore, a list of instructions is set within a ROM (Read-Only Memory) memory considering the CoreABC hard mode [54].

The CoreAPB3 [55] works as a multiplexer, connecting up to 16 Completer modules with a single Requester. This core is configured to match the APB interface of the CoreABC module and provide a unique Completer slot for the CoreTSE. Therefore, only 10 of the 16 bits of the APB address bus are employed to read or write the CoreTSE registers.

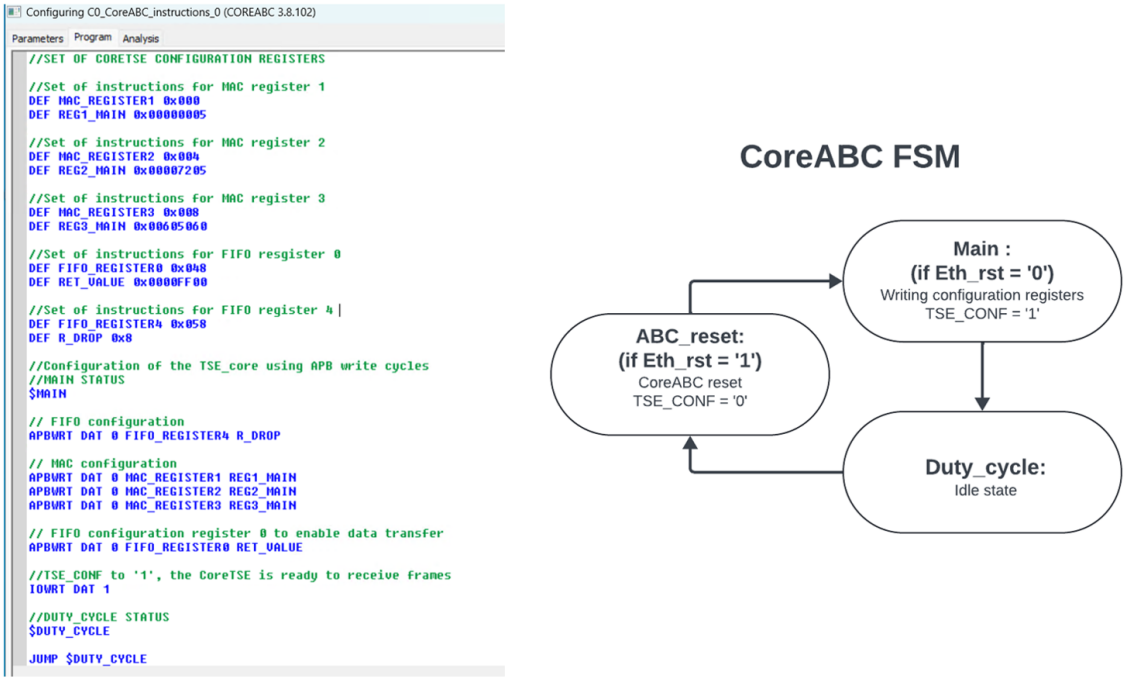


Figure 3.9: On the left the Libero SoC GUI interface provides the assembly code to initialize the CoreABC. On the right, the scheme of the implemented FSM, within the code, is shown.

At CoreABC initialization, the Libero SoC GUI allows defining the instruction lists memory through specific assembly commands. The code implemented is shown on the left of Figure 3.9, providing the definitions of desired addresses and 32-bit values for the CoreTSE configuration set by the following instructions implementing an FSM [2]. As the CoreABC works synchronously with the Eth_clock and uses the Eth_rst as reset signal, the FSM of Figure 3.9 is generated providing the following 3 states:

- ABC_reset: whenever the Eth_rst is set at logic '1', the CoreABC is fully reset. Therefore, the APB signals and the TSE_CONF are set to '0'.

[2]Finite-State Machine

- `Main`: it provides APB write cycles for all the defined registers whenever the `Eth_rst` deasserts. Then the value of `TSE_CONF` output signal is set to '1', indicating the end of `CoreTSE` configuration.

- `Duty_cycle`: after the configuration of all the defined registers, the state machine waits here until `Eth_rst` signal asserts again.

The `ABC_reset` state is not shown in the code of Figure 3.9, since it is intrinsically implemented within the `CoreABC` module definition.

The `CoreTSE` is configured after the whole Ethernet subsystem is reset. Besides the command for the needed Ethernet GMII communication, the MAC is also configured to append PAD and CRC fields to each frame, allowing a connection with any external device that needs a minimum frame length of at least 64 bytes and can not produce a CRC code. The MAC FIFO is configured to drop any received frame that provides a metastable 8-bit word, as marked by the `gmii_rx_er` signal.

Considering the `CoreTSE` connection to the `TSE_converter_interface` module, Figure 3.8 shows the `MRX` and `MTX` interfaces, which collect the signals of Table 3.3 respectively for `TSE_converter_interface` frame reception and transmission. The data transfer in both directions is synchronous with the 125 MHz `Eth_clock`, which feeds both the `MTXCLK` and `MRXCLK` clocks.

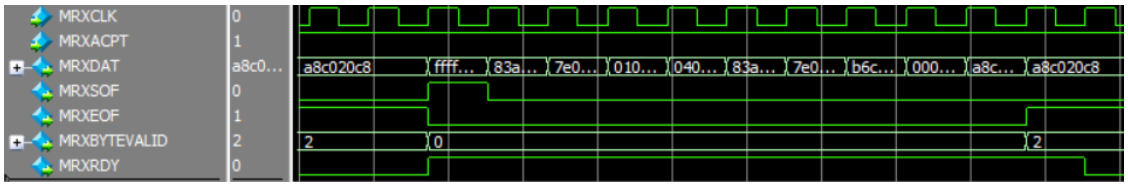


Figure 3.10: MRX data transfer cycle where all the signals are synchronous with the MRXCLK.

Figure 3.10 describes a received frame sent from the `CoreTSE` to the `TSE_converter_interface` module. For data reception, the `MRXACPT` signal has to be high during all communication. Therefore, such a signal is fed by the `TSE_CONF`, since whenever the `CoreTSE` is configured it is ready to send data. Then, during the frame transfer along the `MRXDATA` bus, `CoreTSE` asserts `MRXRDY` signal for the whole frame length and sets both `MRXSOF` and `MRXEOF`, to mark the first and last 32-bit words of the frame. Furthermore, the `MRXBYTEVALID` bus indicates the number of not-valid MSBs within the last word of the frame, synchronous with the `MRXEOF` assertion.

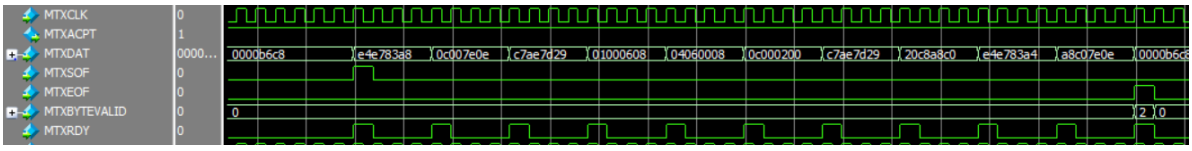


Figure 3.11: MTX data transfer cycle where all the signals are synchronous with the MTXCLK.

On the other hand, Figure 3.11 describes a frame transmission from the `TSE_converter_interface` to the `CoreTSE`. As shown in Table 3.3, all the used signals are `CoreTSE` inputs except for the `MTXACPT` which is an output signal set to '1' during all the transmission, as it indicates the `CoreTSE` is ready to receive data. The `MTXSOF` and

MTXEOF mark respectively the beginning and the end of the frame, while the MTXRDY is set at "1" for all the transmitted valid data. The MTXBYTEVALID is always set synchronously with the MTXEOF assertion and indicates, as before, the number of not-valid MSBs in the last word of the frame.

As shown in Figures 3.10 and 3.11, the TSE_converter_interface module manages MRX and MTX interfaces providing different data streams. In detail, the MRXDATA bus sends a 32-bit word at each MRXCLK rising edge, while the MTXDATA needs four clock cycles, of the MTXCLK clock, to set a new 32-bit word. This behavior is explained in the following subsection and is caused by the internal conversion performed by the TSE_converter_interface module, to match an 8-bit data bus to a 32-bit data bus running with the same 125 MHz clock frequency.

3.2.3 The TSE_converter_interface

The TSE_converter_interface is designed as a bridge between the IPbus_ctrl and the CoreTSE modules, considering both directions. This module is mainly based on Finite State Machine (FSM) processes, which are sequential circuits that keep track of the current state and react to the input and the current state value. As already mentioned, the TSE_converter_interface works as a converter for the signals collected in Tables 3.1 and 3.3, considering FSMs synchronous with the common Eth_clk running at 125 MHz.

Figure 3.12: Comparison between the MRX cycle and the mac_rx cycle, in which the signals considered in the related FSM are labeled.

As shown in Figure 3.12 in a data transfer from CoreTSE to IPbus_ctrl, marked as the rx direction, a data stream based on 4-byte words has to be converted into a byte-wise stream. To fulfill such a purpose, a single clock FIFO receives as input the MRXDATA bus and the CoreTSE control signals, shown in Figure 3.12. The FIFO output have to provide a byte-wise data stream associated with some control signals that indicate the beginning and the end of the received frame. Therefore, the FIFO input bus was designed to provide a FIFO output of 12 bits width, in which:

- the bits [11:4] contain the data information of one byte at a time, for a 4-byte word sent on the MRXDATA bus.
- the bits [3:2] contain the input information of the MRXBYTEVALID bus for each data byte.
- the bit position 1 and 0 are used for the MRXSOF and MRXEOF signals for each data byte.

This output is used, within an FSM, to generate the mac_rx_last and mac_rx_valid control signals related to the mac_rx_data bus, for the IPbus_ctrl module interface. Figure 3.13 shows the FSM, in which the following I/O signals are considered:

- `SOF`(Start of Frame): it is the signal related to the `MRXSOF` information within the 12-bit data stream.
- `EOF`(End of Frame): it is the signal related to the `MRXEOF` information within the 12-bit data stream.
- `bytevalid`: it provides the value of `MRXBYTEVALID` bus within the 12-bit data stream.
- `rst`: it represents the `Eth_rst` generated by the `Clock_and_Resets` module.
- `valid`: it represents the `mac_rx_valid` output signal.
- `last`: it represents the `mac_rx_last` output signal.

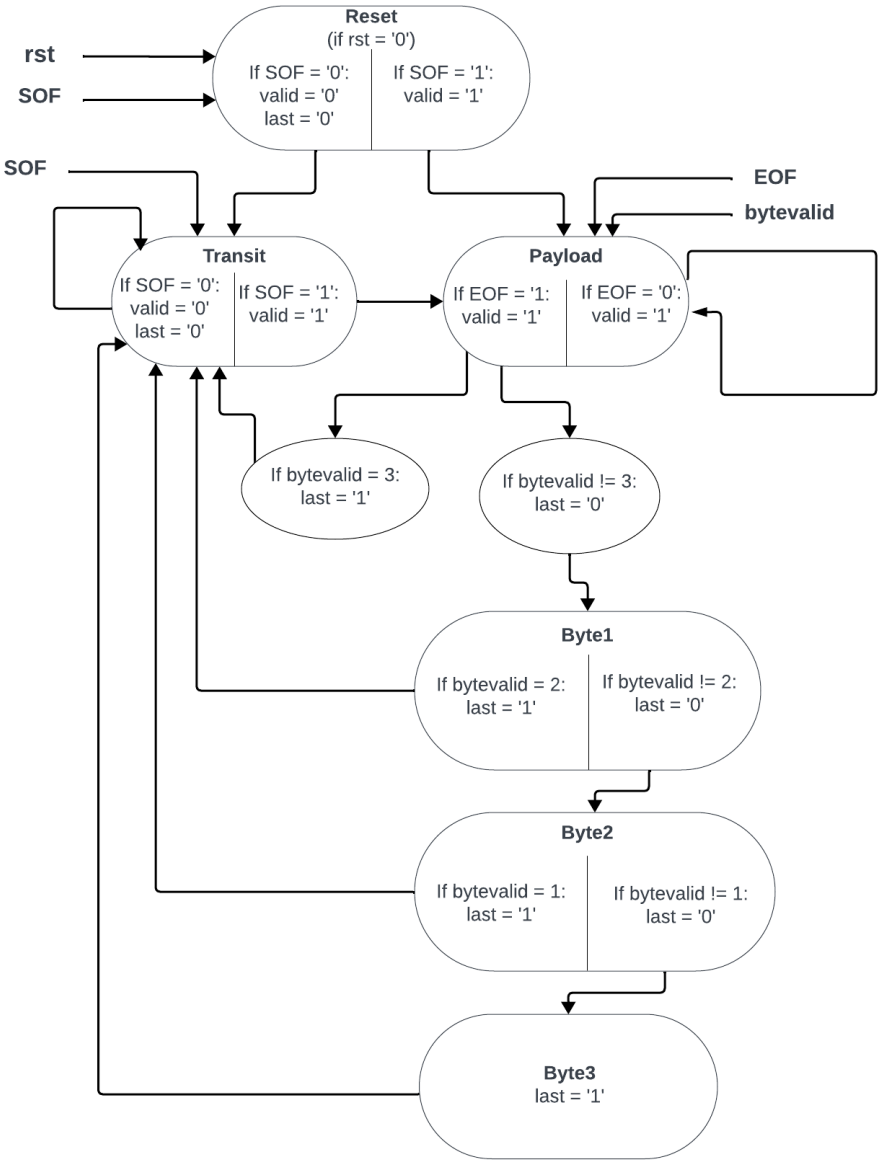


Figure 3.13: FSM design to build the `mac_rx_valid` and the `mac_rx_last` signals.

When the rst is set at logic '1' the FSM is switched off, keeping every output signal at logic '0' and setting the FSM state at Reset value. Therefore, the FSM works only when the rst signal is de-asserted, following the structure shown in Figure 3.13 that features these 6 states:

- Reset: it is the first state after rst deasserts, in which the beginning of a frame is searched. If no frame is found the state switches to Transit otherwise the Payload is chosen as the next state, setting the valid signal at '1'.
- Transit: it always searches the SOF signal at the '1' logic state, to find the beginning of a frame. If no frame is found the FSM stays in the Transit state, otherwise it switches to Payload setting the valid signal at '1'.
- Payload: it searches for the EOF signal at the '1' logic state, as it indicates the last word of the frame. It also evaluates the bytevalid value and if 3 MSBs are not valid the state goes back to Transit state, otherwise the Byte1 state is considered.
- Byte1: it evaluates the bytevalid value for the case of 2 not valid MSBs within the last frame word. Therefore, if bytevalid value is equal to 2 the state switches to Transit, otherwise the Byte2 state is considered.
- Byte2: it acts as the Byte1 state evaluating the case for bytevalid equal to 1 and considering as next states Transit or Byte3 states.
- Byte3: it considers the case of all valid bytes within the last frame word, with a bytevalid value equal to 0. The state goes back to the Transit state.

Such a process finds the frame beginning to assert the valid signal and waits for the consequent EOF signal to mark the last word of the frame. When the end of frame is found the bytevalid is evaluated to assert the last signal and go back to the Transit initial state. At this point, both valid and last signals are de-asserted and the loop restarts again searching for a new frame.

On the other tx direction, a data stream, sent from the IPbus_ctrl module to the CoreTSE, has to provide a conversion from the 8-bit data bus mac_tx_data to the 4-byte wise MTXDATA bus, as shown in Figure 3.14. Furthermore, the mac_tx_last and mac_tx_valid signals must be used to obtain the MTX control signals for the CoreTSE module.

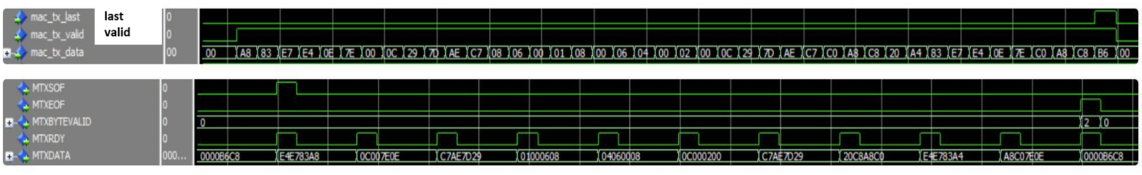


Figure 3.14: Comparison between mac_tx cycle and the MTX cycle, in which the signals used in the related FSM are labeled.

To manage such data stream conversion, the FSM in Figure 3.15 considers the following I/O signals:

- valid: it represents the mac_tx_valid input signal.

- `last`: it represents the `mac_tx_last` input signal.
- `rst`: it represents the `Eth_rst` input signal.
- `new_valid`: it is a new output `valid` signal shaped to have a length multiple of 4 clock cycles, as the `mac_tx_data` 8-bit bus transmits synchronously with the same clock used by the `MTXDATA` 32-bit bus.
- `final`: it is an output signal that marks the time between the `last` signal assertion and the beginning of another frame.
- `bytevalid`: it is an output bus that assumes the specific values of the `MTXBYTEVALID` bus until the FSM processes a new frame.

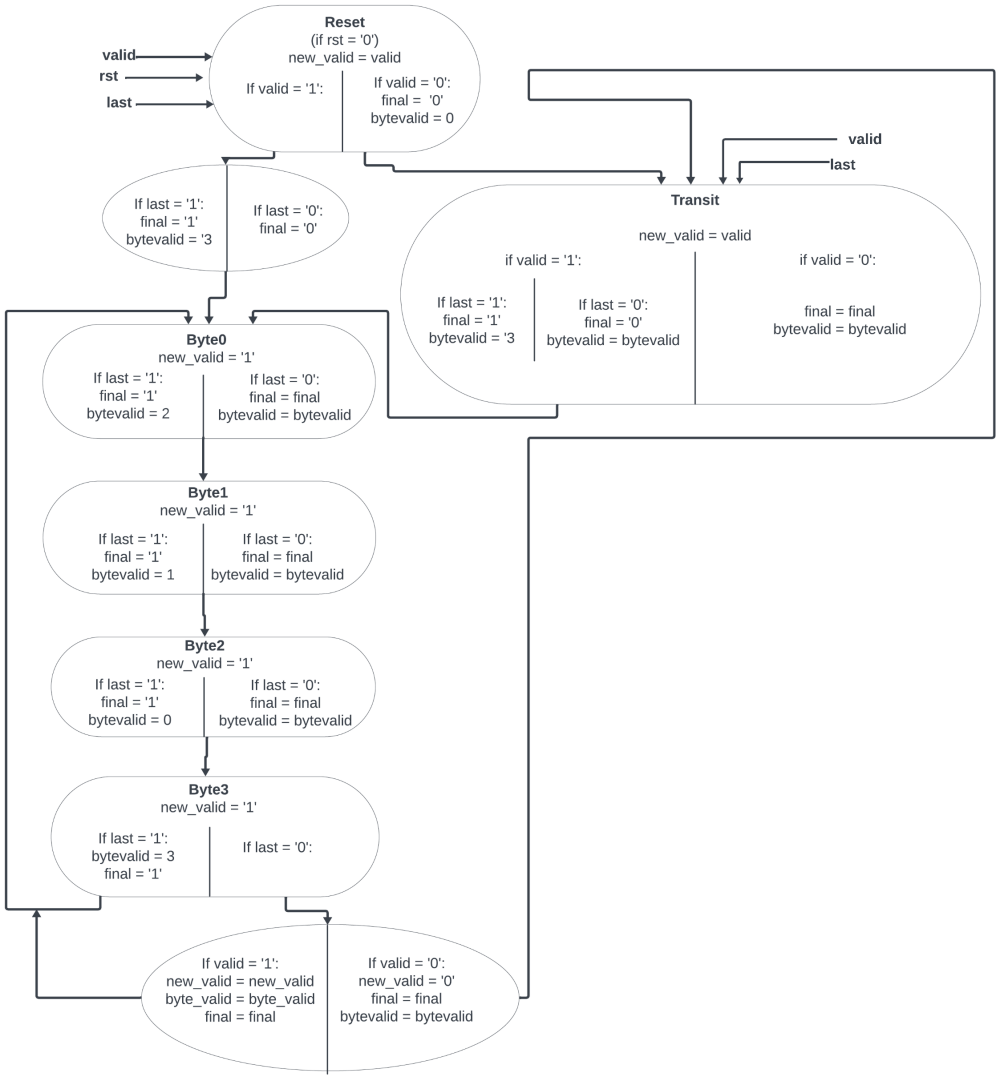


Figure 3.15: FSM design to generate the `new_valid`, `bytevalid` and `final` output signals.

In detail, the output contains only preliminary signals to generate the final `MTX` interface of the `CoreTSE`. As before, the FSM works only when the `rst` signal is set at

'0', otherwise all the outputs are kept at '0' and the initial value Reset is set for the FSM state. Figure 3.15 shows the FSM design that is based on 6 states as explained below:

- Reset: when rst signal is de-asserted, the FSM searches for an incoming frame evaluating the valid input signal. If a frame is found the state switches to Byte0 state, otherwise the Transit state is chosen.
- Transit: the FSM waits in this state until a frame arrives, meaning for the valid assertion that switches the state to Byte0.
- At this point, the FSM works within the frame length performing the following loop as follows:
  1. Byte0: it searches for the last signal assertion to initialize the bytevalid signal with value 2 and assert the final signal. Otherwise, the bytevalid keeps the value found in the previous state.
  2. Byte1: it works similarly as Byte0 state, initializing the bytevalid signal with value 1.
  3. Byte2: it works similarly as Byte0 state, initializing the bytevalid signal with value 0. This state ends the loop for a 4-clock cycle length, which is the unit that defines the new_valid signal length for a frame.
  4. Byte3: it evaluates the last signal to exit the loop or restart from the Byte0 state. When last is equal to '0', the valid signal is evaluated to know if the FSM is still analyzing a frame (valid = '1') or not. In the first case the loop restart from Byte0, otherwise the FSM switches the state to Transit searching for a new frame.

Specifically, the Reset and Transit states also include the last signal evaluation to deassert the final signal at the beginning of a new frame, when last = '0', and analyze weird frames with only one valid byte, when last = '1'.

In summary, at the beginning of a frame, the new_valid signal asserts within the Transit state. Then the next loop starts and, whenever last = '1', the final signal asserts, while bytevalid signal takes a value considering the current FSM state. When Byte3 state is found and valid = '0', the loop ends deasserting the new_valid signal and switching the FSM state to Transit.

The output FSM signals feed a process based on a 4-stage shift register, which implements byte memory units using Flip-Flops synchronous to the Eth_clock. When new_valid asserts, each memory unit stores a 1-byte incoming word at each clock cycle. Then, a 4-byte packing mechanism is built through a counter ranging from 0 to 3. Whenever the counter takes the value of 3, a 4-byte word is sampled on the MTXDATA bus, asserting the corresponding MTXRDY for one clock cycle, to mark 1 valid word. Finally, the control signals MTSOF, MTXEOF and MTXBYTEVALID are generated using the counter and both the FSM signals: bytevalid and final.

3.2.4 The UDP_engine

The Ethernet frames are managed by the IPbus_ctrl module, in which the UDP_engine organizes the functions of each layer within the UDP/IP model. The UDP_engine

was developed by CERN to manage the Ethernet traffic for the IPbus application and provide auxiliary utilities, besides UDP transactions. A simplified scheme of its internal structure is shown in Figure 3.16, in which the module `UDP_IF` collects all the logic blocks that implement the following features:

- **Board addressing**: it scans the received frame to check for board correct addressing at each UDP/IP model level. Furthermore, it builds Ethernet frames in response to a service request, considering the source address defined in the firmware and, as the destination address, the value of the source address stored during the request reception.
- **IPbus transaction**: within an Ethernet frame, it recognizes a UDP packet providing UDP destination port 50001 (0xC351). The module scans the UDP payload for an IPbus packet header to manage the following IPbus transaction requests. Then, corresponding IPbus read and write cycles are performed within the IPbus subsystem, and the related Ethernet frames, containing an IPbus packet with all the IPbus transaction responses, are built and sent to the requester client.
- **IPbus reliability**: it scans received UDP frames that provide IPbus packets (UDP destination port 50001) for resend and status IPbus requests, to manage UDP transaction timeout. Then, status and re-send responses are built and sent back to the client to restore the communication.
- **PING utility**: it builds the response to an ICMP[3] frame and sends it back to the host that makes the request. The ICMP request and response frames provide an IP packet with a specific payload containing an ICMP header and the corresponding data. The PING utility is mainly used to verify the presence of an IP-addressed device on a network.
- **ARP utility**: it builds the response of an ARP request and sends it back to the client host. The ARP request and response are particular Ethernet frames with a Type field 0x0806, in which the payload provides the main ARP packet as defined by the IP protocol suite. The ARP utility is used to get the MAC address of a device knowing its IP address.
- **RARP utility**: it works similarly to the ARP, providing a similar frame structure with a Type field 0x0835. The RARP utility is used to get the IP address of a connected device knowing its MAC address.

The `UDP_IF` block evaluates each field of the Ethernet frame, as it needs to recognize a utility frame pattern and trigger only the related processes. Considering the IPbus transaction case, the `UDP_IF` module looks for the IPbus packets and sends the whole Ethernet frame to the connected `Transactor` module. The `Transactor` searches the IPbus transactions for read and write IPbus cycles and acts as the master within the IPbus subsystem. Figure 3.16 shows how the information of IPbus transactions is managed through the `Transactor` module connection, which works considering the `trans_in` and `trans_out` buses explained in Table 3.2.

[3]Internet Control Message Protocol

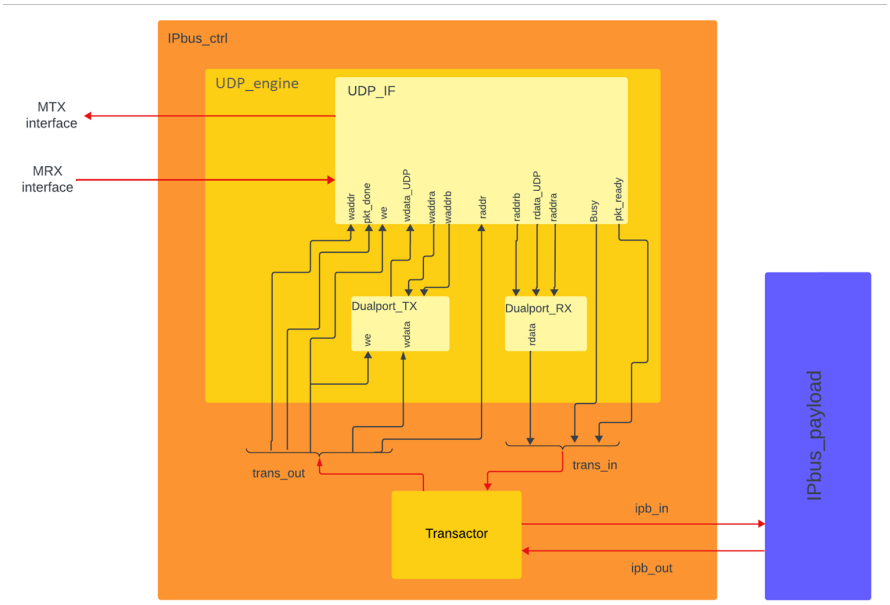


Figure 3.16: `UDP_engine` internal structure, showing the connection with the `Transactor`.

During the reception, the `Transactor` works considering the `Dualport_RX` module that is a dual port RAM storing the frame sent from the `UDP_IF`. The addressing of the input and output memory buffers is controlled respectively through the `raddra` (13-bit) and `raddrb` (11-bit) buses. So, a valid Ethernet frame stored in the RAM is marked by the `pkt_ready` signal and is transmitted to the transactor on the `rdata` bus, while `UDP_IF` menages the addresses of the `Dualport_RX` buffers and keeps `Busy` asserted. During this process, the `UDP_IF` uses the `raddr` counter, to manage the `raddrb` address of the RAM output buffer. When an IPbus transaction request has been processed, the `pkt_done` signal is asserted and the`Transactor` is ready to process another IPbus transaction packet.

A similar mechanism is defined considering the `Dualport_TX` dual port RAM for transmitting the IPbus transaction packets, from the `Transactor` to the `UDP_IF` module. During the packet transmission to the memory buffer the data are marked by the `we` signal and the `wraddr` value is used, within `UDP_IF`, to manage the `waddra` (11-bit) bus for the RAM input buffer. Furthermore, the `we` signal is also sampled by the `UDP_IF` module to control the `wdata_UDP` data stream, using the `waddrb` (13-bit) address of the RAM output buffer. At the end of each transaction packet transmission the `pkt_done` signal is asserted.

3.3 IPbus slaves used for TDCs

The IPbus system is implemented within the firmware to manage 12 IPbus slaves, set inside the `IPbus_payload` module. Each slave controls specific hardware features, as explained in subsection 3.1.2. Even if the IPbus is based on a 32-bit address bus, our system uses only 8 bits out of 32 to address a specific slave. Figure 3.17 shows how the 32-bit address width is managed, employing only the least significant byte divided into two 4-bit sections:

- `Slave_addr`: it is the specific slave address.

- `Reg_addr`: it indicates the internal register address of the selected slave.

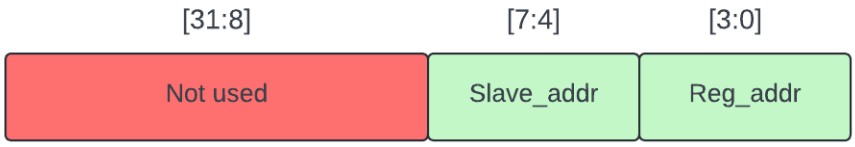


Figure 3.17: Bits scheme employed for slave addressing, within the 32-bit address featured by the IPbus protocol.

All the 12 slaves are listed in Table 3.4 with their corresponding `Slave_addr` and the number of internal registers (if `Reg_addr` number is set to 1 it means this specific slave has no internal registers). Inside this section, the slaves used as an interface with the PicoTDCs are described, as they are more relevant to the thesis topics. These slaves mainly work synchronously with the `IPb_clock` running at 31.25 MHz, and provide different features.

Name	**Slave_addr**	**# Reg_addr**
`LEDs`	0x1	4
`Ctrl_and_Status`	0x2	2
`I2C_master`	0x3	7
`TDCsext_signals`	0x4	1
`PicoTDCA_readout`	0x5	4
`PicoTDCA_readout2`	0x6	4
`PicoTDCA_readout3`	0x7	4
`PicoTDCA_readout4`	0x8	4
`PicoTDCB_readout`	0x9	4
`PicoTDCB_readout2`	0xA	4
`PicoTDCB_readout3`	0xB	4
`PicoTDCB_readout4`	0xC	4

Table 3.4: Table that associates the IPbus slave to its corresponding `Slave_addr`, indicating also the number of internal registers.

3.3.1 The TDC external signals generator

As described in subsection 2.1.4, the PicoTDC ASICs provide the event and bunch counters, built with a 13-bit width. The first one keeps track of the triggered events during the full time of a measure. The other instead is implemented to support the ASIC usage inside a collider environment, described by a beam bunch structure. Therefore, the bunch counter keeps the TDC synchronized with the number of particle bunches passed through the accelerator.

These two counters run synchronously with the 40 MHz reference clock and can be reset to a configurable value, using proper external pins. Specifically, the PicoTDC provides four main external differential input signals, used to control some of its features:

- **Event reset**: it resets the event counter.
- **Bunch reset**: it resets the bunch counter
- **Digital reset**: it fully resets the TDC digital section (it does not reset the event and bunch counters).
- **Trigger**: it provides the trigger signal to the TDC trigger interface.

Therefore, the `TDCsext_signals` slave is implemented to generate all these four signals for both TDCs.

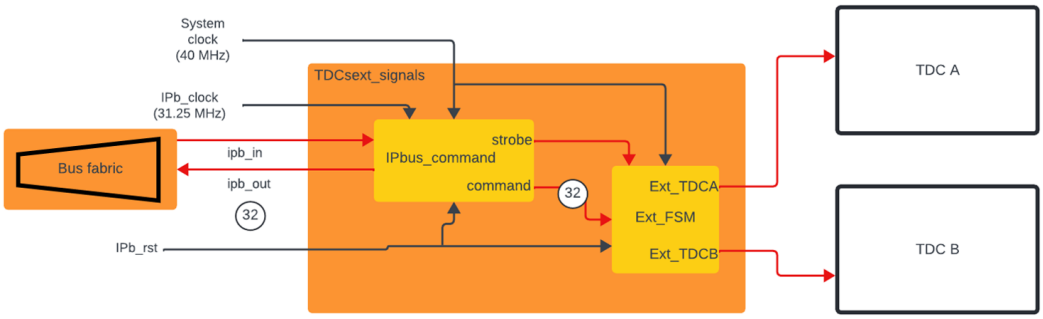


Figure 3.18: Internal structure of the `TDCsext_signals` slave, showing the connections with both the PicoTDCs.

Figure 3.18 shows the internal structure of `TDCsext_signals`, that control respectively the `Ext_TDCA` and `Ext_TDCB` outputs collecting the 4 external signals respectively for TDC A and TDC B. This module can convert the information provided by the 32-bit data of an IPbus write cycle into instructions for building the desired signals.

The `TDCsext_signals` architecture contains the `IPbus_command` module that mainly implements a Dual clock FIFO IP core [56] to match the `IPb_clock` and `System clock` subdomains, running respectively at 31.25 MHz and 40 MHz. As the FIFO implements the same 32-bit width both in input and output, the final `IPbus_command` output includes the `Strobe` signal, indicating the arrival of a command, and a 24-bit data bus running at 40 MHz, which provides a 24-bit command for the `Ext_FSM` FSM module.

As the `IPb_rst` works as an asynchronous reset for all the `TDCsext_signals` module, when `IPb_rst` = '1', all the output signals are set at '0' and the FSM is initialized at the `Acquisition` value. If `IPb_rst` = '0', the FSM works as shown in Figure 3.19 decoding the command at each `Strobe` signal assertion and switching between its states: `Acquisition` and `Transit`. The FSM waits in the `Acquisition` state until one valid command is provided. Then the state switches to `Transit`, where the instructions are decoded to generate the wanted signal after one clock cycle. Finally, the loop is closed at the `Acquisition` state where the FSM waits for another valid command. The final 4 output signals are then generated as follows:

- **Bunch reset and Event reset**: they are built as 1 clock cycle pulses.
- **Digital reset**: it is built considering two different modes:
  - **Soft mode**: it asserts the digital reset for a 1 ms time length.

  - **Hard mode**: it asserts the digital reset until the next command to switch it off is sent.

- **Software trigger**: it is built considering two modes:
  - **Single pulse**: the FSM sends 1 clock cycle pulse.
  - **Continuos trigger**: the FSM generates a continuous pattern of 1 clock cycle pulses with a configurable period distance.

The `Ext_FSM` drives up to 3 reset signals and a software trigger, which was used to test the trigger mode of the two ASICs during the resolution measurement.

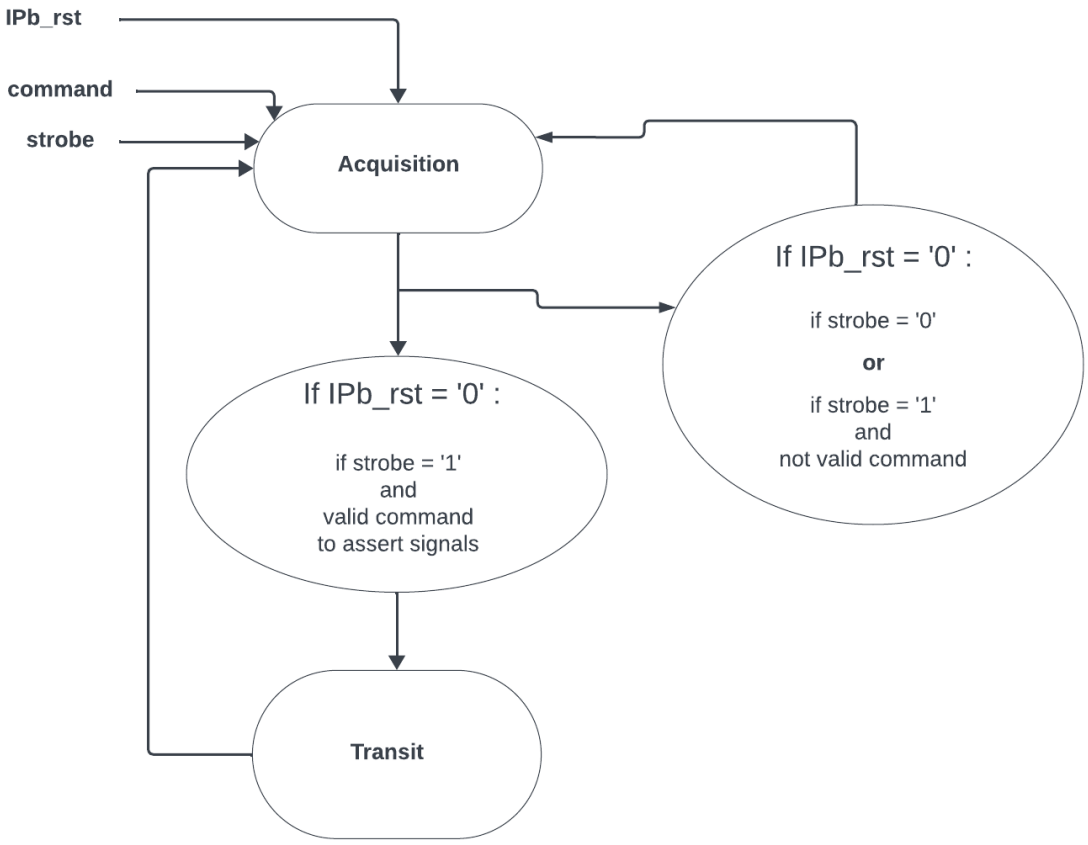


Figure 3.19: FSM used for both TDCs to generate the 3 external reset signals and a trigger, which are software driven.

As shown in Figure 3.20 the 24 bits for a command are divided into two sections. The first one takes from bit 23 to 8 and considers the specific options that can be set for a command. In particular, from bit-position 23 to 20, it is possible to choose the wanted signals, while bits 18-19 assign generated signals to one or both TDCs. The other bits are used respectively to set the soft mode for digital reset (17) and the continuous mode for the trigger (16). If the trigger continuous mode is set, the least significant bits of the first section configure the pulse distance. This value features 8 bits and considers a maximum time distance of $\sim$6 μs as the 40 MHz clock cycle is the minimum time step. The second section is used to send specific signals towards the TDCs, preserving the other ones that are currently asserted. This condition implies setting the corresponding bit to control

Function	Different options								
Signal	Trigger	Dreset	Ereset	Breset	TDCA	TDCB	Soft_Dr	Continuos_Tr	Period for continuos trigger
Bits	23	22	21	20	19	18	17	16	15 to 8

Function	TDCB_selector				TDCA_selector			
Signal	Trigger	Dreset	Ereset	Breset	Trigger	Dreset	Ereset	Breset
Bits	7	6	5	4	3	2	1	0

Figure 3.20: Scheme of the 24-bit command.

a specific signal, otherwise all the previous command options are not considered by the FSM. A set of 4 bits is assigned for each TDC, in which each bit is associated with a signal.

To sum up, the valid command, considered in the previous explanation for the FSM, follows this 24-bit pattern and the conditions explained.

3.3.2 The I2C_master

The PicoTDCs configuration works through an I2C communication bus that can run as fast as Fast Mode Plus (1Mb/s) and was tested up to $\sim$ 0.4 Mb/s speed for our purposes. The I2C is a serial synchronous protocol that implements a multiple master and slave structure using a bidirectional communication based on two lines: the serial clock (SCL) and the serial data (SDA). Our case considers the Figure 3.21 circuit of a single I2C master instantiated as an IPbus slave and connected with both the PicoTDCs, which act as I2C slaves.

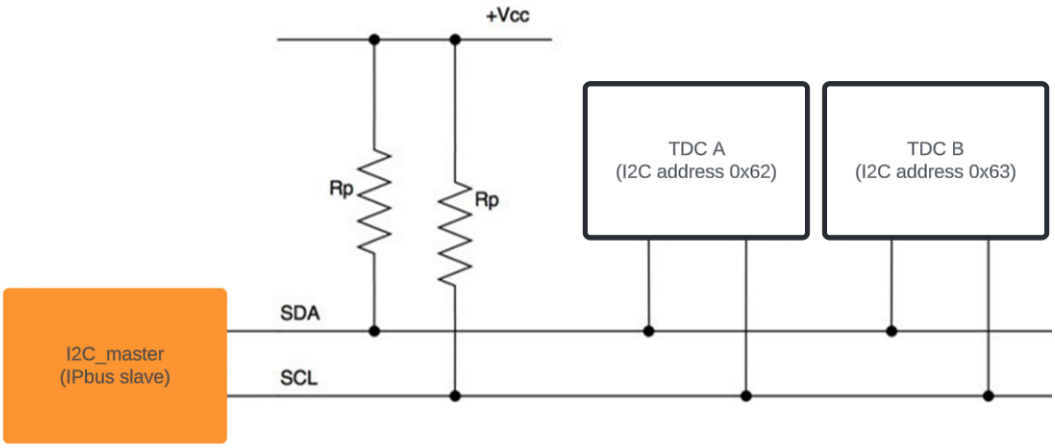


Figure 3.21: I2C network built for the 2 TDCs slave and the `I2C_master` IPbus slave, working as I2C master (V_{cc} = 1.2 V and both the R_p = 4.7 KΩ).

This protocol is half-duplex since only one device at a time can use the `SDA` line to transmit data, while the `SCL` clock is always driven by the master. The signals scheme, shown at the top of Figure 3.22, describes some specific states for the `SCL` and `SDA` signals used in I2C transactions:

- **The start**: the master marks the beginning of a transaction setting a high to low `SDA` transition, while the `SCL` is left high.

- **The stop**: the master stops a transaction setting a low to high SDA transition, while the SCL is left high.
- **The ack**: such a state stands for the acknowledgment from the receiver, indicating a successful data transfer. In detail, the transmitter sets SDA in high impedance and the receiver asserts SDA low for an entire SCL clock pulse. Otherwise, if the data transfer fails the SDA is set high, meaning for a NACK state (not acknowledgment).
- **The data**: SDA transitions take place when SCL is low, while the data is valid when the SCL is high.

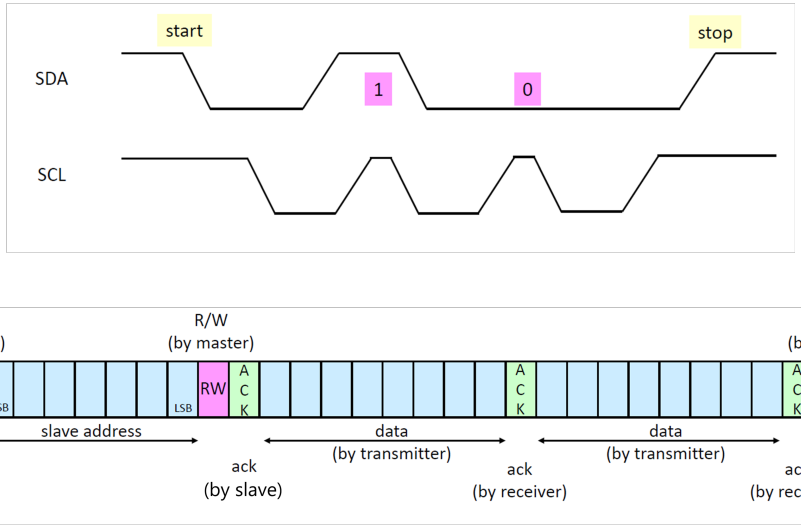


Figure 3.22: On the top we have an example of a SDA-SCL signals, while on the bottom the I2C transaction pattern is shown

The initial protocol IDLE state is defined by a logic high state of both SCL and SDA lines. The data transactions are always started and stopped by the master, which can act as a receiver or a transmitter. Each slave may transmit data after being addressed by the master, using a 7-bit address assigned to each connected slave. Therefore, the I2C transaction pattern, in the bottom of Figure 3.22, is used for read and write I2C cycles. The master always starts the cycle with a start SDA state and provides the SCL. Then it sends the slave address followed by the R/W bit, which is set to '0' for a write and '1' for a read. After this initial condition, the slave sends an ACK on the SDA and the specific cycle begins. Considering a write cycle as an example, the master acts as a transmitter and the addressed slave is the receiver, while the opposite happens for a read. So the transmitter sends one byte and waits for the receiver ACK until the master sets a stop SDA state, ending the cycle. Otherwise, if the receiver sends a NACK state, the communication is immediately stopped by the master.

The IPbus slave I2C_master is provided by the CERN Git repository and implements an FSM that generates the I2C read and write cycles. Such a module is directly connected to the SCL and SDA lines of the two TDCs and its structure is shown in Figure 3.23. Starting from left to right the IPbus_interface module is instantiated to control some internal registers, needed to drive the I2C_master_interface. This is

a sub-module that provides in its internal structure two single clock FIFOs, using the generated `IPb_clock`, and the `I2C_master_logic`, which mainly includes the FSM for the I2C implementation.

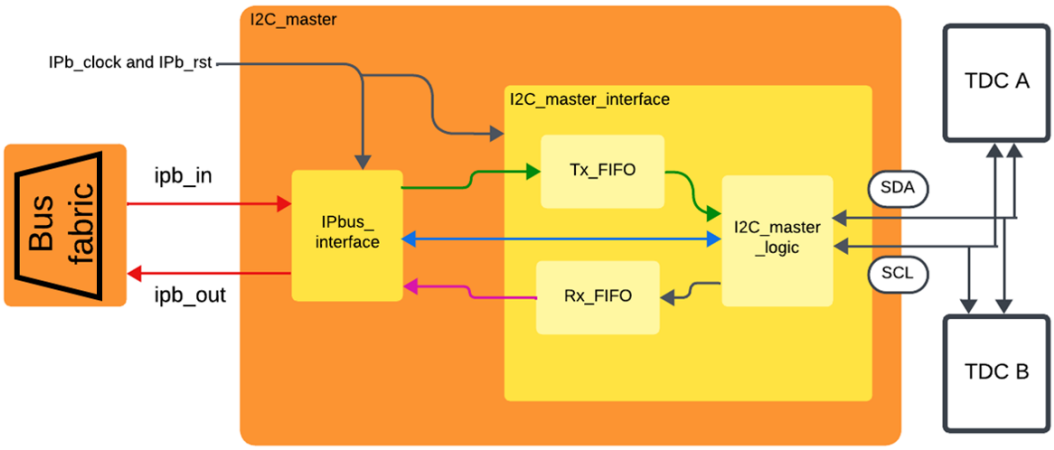


Figure 3.23: I2C Master internal structure where each colored line, linked to the `IPbus_interface`, indicates a set of internal registers, explained in Table 3.5, while the red color is used to indicate the IPbus data transfer.

The `Tx_FIFO` provides the byte data stream for an I2C write, while the `Rx_FIFO` gets the byte data stream obtained after an I2C read. The `IPbus_interface` uses the internal register shown in Table 3.5 to build the I2C transactions inside the `I2C_master_logic`, using IPbus software commands.

First of all, we use an IPbus write to set, within the `prescaler` register, the `SCL` pulse length and the `data_setup`, using the `IPb_clock` period as the step unit. In detail, the `data_setup` variable defines the time needed to set data on the `SDA` line after the `SCL` falling edge. Such a value must always be lower than the one set for the `SCL` pulse length.

Then, to start an I2C transaction, it is compulsory to write the I2C slave address inside the `device_addr` register. The `I2C_master_logic` FSM uses such a value to address the desired slave. After this preliminary setup, the FSM is ready to start an I2C transaction triggered by the values written in registers `rd` for a read and `wr` for a write.

Following the green line in Figure 3.23, the bytes, needed for a write, are written inside the `TX_FIFO` module using register `wr_data`. Then the write cycle starts whenever a random value is written inside `wr`, using an IPbus write cycle. All the bytes inside the `TX_FIFO` are read by the FSM and used for the I2C transaction pattern sent to the addressed slave.

On the other hand, for a read cycle, the number of desired read bytes must be written inside the `rd` register using an IPbus write. At this point, the FSM starts a read I2C cycle and all the bytes, received from the addressed slave, are written inside the `RX_FIFO`. Finally, the FIFO data can be read using IPbus read cycles and addressing the register `rd_data`, as shown by the purple line in Figure 3.23.

colour	name	Register_addr	bits [31:0]	function
Light blu	prescaler	0x0	[31:16] - [15:0]	It sets the length of the SCL pulse and the data_setup, using the IPbus clock cycle as step unit.
	device_addr	0x1	[6:0]	It sets the slave 7-bit address used in the transactions.
	status	0x6	[3:0]	These are status and control outputs signals for the I2C_master_interface.
	rd	0x2	[8:0]	It triggers an I2C read cycle providing the total number of desired bytes.
Green	wr	0x3	[31:0]	It triggers an I2C write cycle.
	wr_data	0x4	[7:0]	It writes a byte inside the Tx_FIFO that will be used for an I2C write.
Purple	rd_data	0x5	[7:0]	It reads a byte inside the Rx_FIFO, using an IPbus read.

Table 3.5: List of the internal registers of the IPbus_interface module, considering the three line colors used in Figure 3.23.

3.3.3 The TDC readout slave

As already mentioned, each TDC provides 32 differential readout lines organized in 4 ports of 8 lines each. This design reflects the internal arrangement of the 64 input channels divided into 4 groups of 16 channels each. In detail, the PicoTDC ASIC includes a 512-word deep FIFO for each group as the final data stage. This FIFO has a 32-bit input as the TDC data width defined by the constructor, while the output is connected to a parallel differential interface of 8 lines. In summary, each FIFO output corresponds to one of the 4 readout ports and is synchronized with another differential output line called Sync.

Figure 3.24 shows how the data readout works for each port, where the readout label is assigned for each output differential line and can assume an integer value from 0 to 3. The bottom of the figure includes the 32-bit data stream called Frame. The Frame pattern shows how the TDC data output is arranged using 4-byte words, defined by the

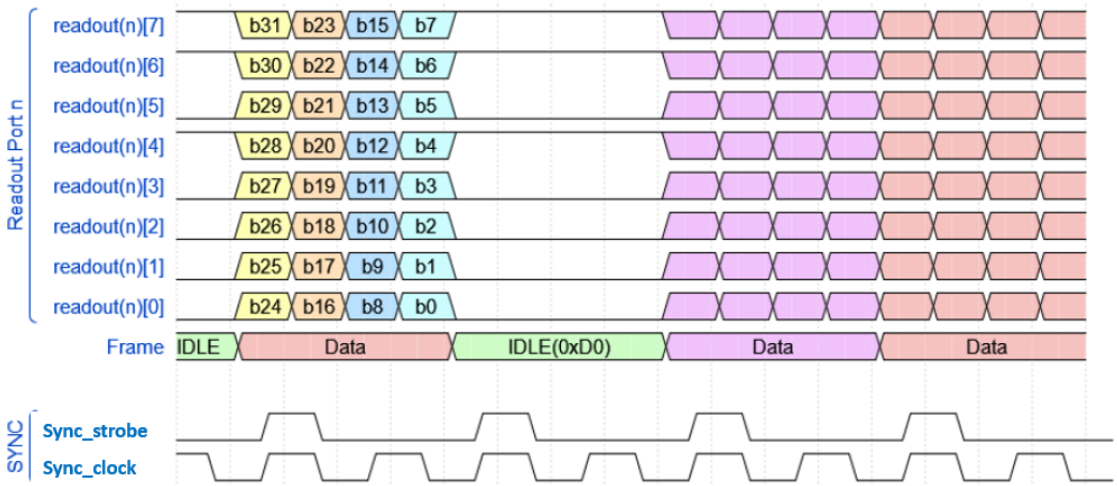


Figure 3.24: One TDC port readout considering both modes for the `Sync` differential signal.

8-line output after 4 consecutive readout cycles. When there is no data inside the FIFO, the parallel port is set to a constant value 0xD0, which builds the idle word 0xD0D0D0D0. Therefore, the final frame pattern provides TDC data interspersed with idle words when no data are found in the FIFO. To provide data output for other devices connected with the TDC, the `Sync` output signal can be configured via I2C for 2 different readout modes as shown in Figure 3.24:

- **Sync_strobe**: the first byte of a 32-bit word in the frame bus, is marked by a pulse of a readout clock period length. Then, the next three bytes are sent synchronized with the readout clock rising edge.
- **Sync_clock**: it defines a periodic signal with a frequency of half the data rate, to provide a DDR output in which each byte is synchronous with the rising or the falling edge of the `Sync` signal.

Furthermore, regardless of the `Sync` mode, the TDC data throughput can be configured by setting the readout rate at a frequency of 320/160/80/40 MHz. The user can choose among two modes that define the number of 4 ports used:

- **Single port readout**: all the data are sent out through port 0, reducing the total readout bandwidth and the number of pins needed to interface with the ASIC.
- **Four ports readout**: the data are sent out considering the internal TDC channels arrangement in 4 groups. In this case, each port is connected to its respective FIFO and all the pins are used to interface with the TDC.

To test both these two configurations, it is compulsory to instantiate one IPbus slave for each PicoTDC port. Therefore, the firmware provides, for each TDC, four `PicoTDCX_readoutY` slaves, where, as shown in Table 3.4, the `X` label defines the connected TDC (A or B) and the `Y` is a number from 2 to 4, indicating the TDC ports from 1 to 3, or takes no value for port 0. Table 3.4 also shows the `PicoTDCX_readoutY` corresponding addresses, considering values from 0x5 to 0xC. Configuring the TDC for a 160 MHz readout rate, a 8 bits $\times$ 160 MHz = 1.28 Gb/s bandwidth is obtained in the case of Single

port readout, while in the Four ports readout a higher 4 ports × 8 bits × 160 MHz = 5.12 Gb/s bandwidth is set. The `PicoTDCX_readoutY` design exploits the Sync_strobe mode and supports a data rate of 160 MHz, as it ensures a 1.28 Gb/s bandwidth for only one port that is a sustainable rate for the IPbus system implemented considering a small data buffer to store the TDC output.

The internal structure of this IPbus slave is explained in Figure 3.25, in which the `PicoTDCA_readout` is shown as an example. This module is mainly based on a dual clock FIFO, provided by the IP Libero catalog [56], and exploits the same 32-bit width for both input and output. The FIFO write operation is synchronous with the PLL-generated `TDC_readout_clock` clock, running at 160 MHz, while the FIFO read operation uses `IPb_clock`, running at 31.25 MHz. The reset network of the slave works asynchronously using the `IPb_rst` signal, which means each internal module resets its state whenever the `IPb_rst` is asserted.

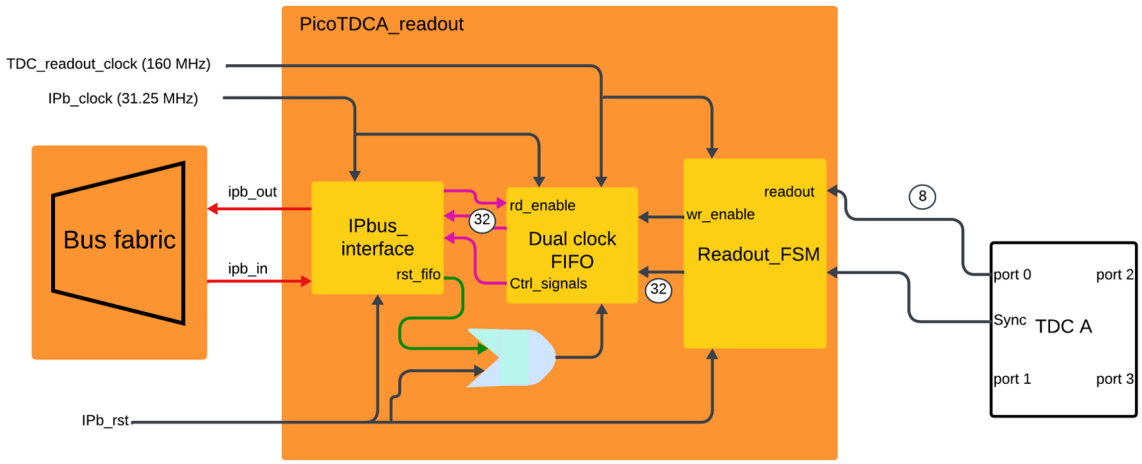


Figure 3.25: `PicoTDCA_readout` module internal structure.

The data written in the FIFO are generated by the `Readout_FSM` that features an internal FSM, running with the `TDC_readout_clock`, to sample the data coming from the TDC and build a 32 bits data stream towards the FIFO. As shown in Figure 3.25, the FSM also generates the FIFO `wr_enable` signal, marking the valid data written inside the FIFO memory.

The FSM is designed to sample the data from the TDC port, considering the `Sync` signal, used in our case as Sync_strobe. In detail, the FSM looks for the `Sync` = '1' condition, which indicates the first byte sent by the TDC on the readout lines. The TDC port output data feeds a 4-stage shift register, which samples the data using the `TDC_readout_clock`. Each of the 4 flip-flops implemented represents a 1-byte unit memory, so the FSM, after receiving the `Sync` pulse, waits for 4 clock cycles and packs the FFs memories in one 4-byte word. At the end of this process, a 32-bit bus is obtained similarly to the one, labeled as Frame, in Figure 3.24. Furthermore, the `wr_enable` signal is generated by the FSM and is asserted when the 32-bit generated data is different from the 0xD0D0D0D0 idle word.

This process is reliable since it works synchronously with the 160 MHz clock generated by the `Clock_and_reset` module, which is fed by the 40 MHz onboard clock used by the TDC to provide a 160 MHz readout rate. The PicoTDC readout clock and the IPbus slave clock (`TDC_readout_clock`) frequency match and are driven by two different PLLs fed

by the same source clock. In detail, the TDC PLL and the FPGA PLL, set within the Clock_and_reset module, are designed to provide no phase difference between the source and the output clocks, ensuring the reliability of the IPbus slave for sampling the output of the TDC port.

To manage the data throughput from the TDCs, the FIFO RAM size has been chosen to contain $\sim 16K$ words. The wr_enable signal controls the FIFO writing operation, while the FIFO read is managed by the IPbus_interface, using the internal registers explained in Table 3.6. Therefore, the FIFO write cycle is automatically provided by the TDCs, while the read cycle is controlled by user IPbus commands.

Colour	Name	Register_addr	Bits [31:0]	Function
Green	Reset_FIFO	0x2	[31:0]	Setting 0x1 it is possible to reset the FIFO.
Purple	Cnt	0x1	[14:0]	Ctrl_signals: it returns the number of available data for a read operation, inside the FIFO buffer.
	Empty	0x1	[30]	Ctrl_signals: it indicates if the read buffer is empty.
	RxR	0x3	[31:0]	It triggers the rd_enable signals to pull out, from the read buffer, only one 32-bit word at a time.

Table 3.6: List of the internal registers of IPbus_interface with different line colors, as shown in Figure 3.25.

Therefore, as shown in Figure 3.25, following the purple lines related to the rd_enable, we can use an IPbus read cycle, addressing the RxR register, to trigger the FIFO rd_enable signal and pull out a 32-bit word from the FIFO. The other registers linked to the Ctrl_signals lines can also be read via IPbus, to know how many available data are inside the FIFO read buffer. The Empty register checks the Empty flag assertion for the FIFO buffer and the Cnt register provides the number of available data for the FIFO read operation.

On the other hand, the green line in Figure 3.25 corresponds to the rst_fifo signal, used together with the IPb_rst signal as input of an OR port connected to the FIFO reset. This signal is controlled by the Reset_FIFO register, which can be addressed through an IPbus write cycle. By writing the 0x1 value within such a register, the FIFO reset is asserted for 3.2 μs removing all stored data in the RAM.

Executive summary

The firmware implemented supplies an on-chip IPbus that can be controlled through UDP transactions, sent from and to a host via an Ethernet connection. In detail, the architecture provides a hierarchical topology of VHDL logic modules, connecting the RGMII PHY interface to 12 IPbus slaves that control the board and perform readout. Therefore, within the firmware, Ethernet frames are managed at each UDP/IP model level up to the IPbus packet, then decoded to perform write and read IPbus cycles.

Almost all the 12 IPbus slaves are used for the configuration and readout of the PicoTDCs on board. For what concerns TDC configuration and control, a slave for the I2C communication with the TDCs and another one controlling the external signals to the TDCs were designed. Each ASIC readout was supported by 4 slaves, which were built to match each TDC port data throughput considering a configured readout rate of 160 MHz.

Chapter 4

Software organization for the PicoTDC board

The software project was developed to match the implemented firmware features and is based on the source files of the same Git repository used for the firmware development [57]. In particular, the software is designed to provide a final end-user interface that uses simple terminal commands for the configuration and readout of both the PicoTDC ASICs on board. Therefore, using C++ as the programming language, its structure was built as in Figure 4.1, in which the interactive interface is the top layer of a three-layer model of increasing abstraction level from bottom to top.

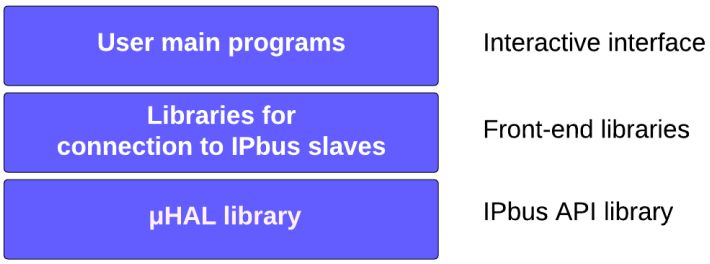


Figure 4.1: Software layers structure.

This chapter provides a software design overview, in which each section is dedicated to a specific layer of Figure 4.1. Starting from the bottom layer, the first section describes the μHAL (Hardware Access Library) library main objects used to build the applications, managed by the Control Hub software application. Then the following section describes the front-end libraries developed to control the operation of each implemented IPbus slave. The final section explains all the implemented commands of the interactive interface, focusing on the main ones used within the PicoTDC time resolution measurement.

The software developed can be found within the same Git repository created for the firmware project [58].

4.1 μHAL API library and Control Hub overview

As explained in section 2.3, the IPbus protocol relies on Ethernet communication to control the hardware features of a device network. Our case considers a simple scenario

of a host connected directly to the PicoTDC board, which uses the implemented firmware on the PolarFire FPGA to manage the Ethernet transactions.

The μHAL was developed to control the IPbus protocol on-chip through specific UDP/IP frames. At the software level, μHAL uses these frames to trigger read or write IPbus cycles inside the IPbus subsystem designed within the firmware. This mechanism is implemented through μHAL objects, identifying each IPbus slave and its internal memory.

4.1.1 The Control Hub application

The Control Hub is a software service developed by CERN to mediate the transactions between general client software processes and connected hardware devices in a network. Built specifically to support the IPbus transactions, such an application queues the different requests provided by developed-μHAL applications and manages each request transmission to an addressed IPbus slave, implemented inside a specific device corresponding to an Ethernet network node. Since IPbus works as a request/response transactional protocol, the Control Hub maintains the connection to the request originator until the slave response is received. In summary, as deeply explained in [59], Control Hub generates connections between a single device and a single μHAL application that sends a request, so no more connection is generated for the connected device until the response is sent to the original requester and the connection closes.

A Control Hub instance can work on the same host of the software processes or on separate remote machines, communicating across the network. Therefore, such an application allows the setup of large DAQ systems including different device crates in a wide xTCA architecture network, where the Control Hub instances work as switches for software process transitions. The communication within each working software client is performed through TCP protocol, while each connected device implements UDP transport protocol for connection with each Control Hub instance. Such a solution has been chosen to build a network of devices that supplies the following features:

- **Complexity only at software level**: the Control Hub solves UDP connection failure using the reliability of the TCP link with μHAL clients. This mechanism allows the implementation of a simpler firmware on the FPGA, based only on UDP.
- **Fast bit-rate**: UDP provides fast communication since it does not supply latency to control transaction failure.
- **Full bandwidth usage for data transmission**: UDP does not provide any control error mechanism, so the bandwidth used towards the external devices is fairly managed by the Control Hub and is exploited only for data transitions.

As mentioned in subsection 2.3.3, TCP is a connection-oriented protocol as it generates a connection between a transmitter and a receiver before starting any data transfer. So, until the connection stands, it uses handshaking algorithms to mitigate connection congestion during data transactions along the connection. Such a TCP functionality provides reliable communication, unlike UDP, which, as a connectionless protocol, cannot ensure reliability.

Figure 4.2 shows our case, which considers a single local host implementing a mono thread μHAL application and the Control Hub, connected directly to the PicoTDC board. As explained in chapter 3, the central FPGA, shown in the figure, provides the UDP

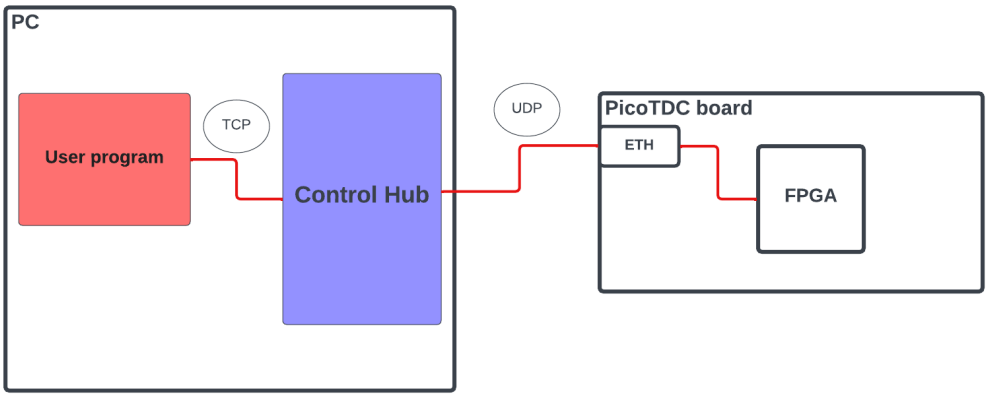


Figure 4.2: Single local-host Ethernet network built for communication with the PicoTDC board.

transactions management for the IPbus system. At the same time, the two developed user programs employ a TPC connection with the Control Hub once at a time, as they were designed as mono-thread μHAL applications. Such a simple Ethernet network was implemented to ensure a steady connection during all the board operations.

This system was implemented using a Linux OS PC, as the Control Hub was developed as a Linux service, and the default Control Hub configuration, shown in Table 4.1, was chosen. Since TCP is a transport protocol, it works by the addressing of specific ports, as shown in subsection 2.3.4 for the UDP. Considering TCP transactions with software clients, the first table parameter indicates the Control Hub port, set to 10203 by default. The 20 ms time represents the timeout during a UDP communication with an IPbus target device, so if a UDP transaction fails the Control Hub sets a timeout of 20 ms. The last timeout of 15000 ms represents the upper time limit in which Control Hub searches for an IPbus target communication. Therefore, the Control Hub terminates the connection started by a particular μHAL application if no response is obtained within 15000 ms of timeout. These parameters can be changed by creating a new configuration file called "sys.config" inside proper directories [60]. Then, in our simple network system, the Control Hub application is just switched on, before any μHAL software operation starts.

Configuration setting	**Default value**
TCP port on which Control Hub listens for connections from clients μHAL	10203
Maximum number of UDP packets transmitted per single transaction to each IPbus target	16
Timeout for communication with IPbus targets (typically FPGA-based devices)	20 ms
Time after which a process will shut down if not communicating with IPbus target	15000 ms

Table 4.1: Default Control Hub configuration [60].

4.1.2 The μHAL API library

As mentioned in subsection 2.3.3, the μHAL library considers 2 different operating modes regarding its connection with hardware devices:

- **The Local-client mode**: the μHAL library communicates directly with the device over UDP.
- **The Remote-client mode**: the μHAL library uses the Control Hub software application to communicate with the hardware.

Upon every μHAL application development, a file with `.xml` extension must be created, as it is used along the code to define the connection parameters for the client host and the target device communication. This file is called the XML connection file and mainly establishes which mode, among the two explained before, is used over the constructed network. Considering the XML connection file `TDCconnection_file.xml` used for our application, it supplies the main elements to describe the performed Ethernet connection [61]:

- **Id**: it is an identifier for the connection used along the application and in our case it considers the pattern `picoTDC_board.chtcp.0` (`board.protocol.project_version`).
- **Uri**: it defines the protocol and location to access a target device in URI format. Our case considers the URI `chtcp-2.0://localhost:10203?target=192.168.200.32:50001`, in which:
  - `chtcp-2.0` defines the usage of TCP channels between Control Hub and μHAL applications, providing the 2.0 version of the IPbus protocol.
  - `localhost:10203` defines the TCP communication between the localhost port for μHAL and the assigned 10203 port address of Control Hub.
  - `target=192.168.200.32:50001` defines the target board endpoint considering its IP address and assigned port 50001 for UDP communication.
- **Address_file**: it defines the location for the address XML file.

The address XML file must also be defined before any μHAL application, as it maps all the address space used for locating the IPbus slaves and their internal registers. At the software level, it represents a hierarchical table reproducing the topology of the IPbus slaves implemented on firmware. Such a file uses the <node> unit to have a one-to-one correspondence between an assigned 32-bit address and the firmware memory location. Figure 4.3 shows this mechanism, in which the IPbus slave with address 0x00000020

```
2  <node id="IPbus_slave" address="0x00000020" fwinfo="endpoint;width = 2">
3      <node id="reg0" address="0x0"/>
4      <node id="reg1" address="0x1"/>
5      <node id="reg2" address="0x2"/>
6      <node id="reg3" address="0x3"/>
7  </node>
```

Figure 4.3: Example of an endpoint node inside an XML address file, providing 4 internal registers.

corresponds to the <node> containing the option `fwinfo = endpoint`. Furthermore, the `fwinfo` option indicates the `width` attribute, which shows the LSbs-number, within the IPbus address, used by the slave to address its internal registers. Indeed, the endpoint node in the figure defines a width equal to 2, providing 4 different internal registers with addresses from 0x0 to 0x3.

Each node is characterized by the following attributes, some of which are not shown in Figure 4.3 for simplicity [61]:

- **Id**: the identifier name used inside the μHAL application.
- **Address**: the hexadecimal address assigned to each instantiated node.
- **Description**: a simple explanation of the node functionalities.
- **Permission**: it is the read or write permission on the correspondent register via IPbus. Specifically, the "r" and "w" stand respectively for read or write permission, while the "rw" allows both. If no permission attribute has been set, the permission is set to "rw" by default.
- **Mask**: it is a 32-bit value used internally as a mask for both the data received and transmitted.
- **Mode**: if the node identifies a memory block inside the firmware, the assigned mode defines its type:
  - **"incremental"**: it identifies a RAM block memory and it must always be supplied together with the "size" attribute that defines the depth of the buffer.
  - **"non-incremental"**: it identifies the FIFO buffer memory linked to this node.

Our system address file `address_table.xml` provides up to 12 endpoint nodes, given we implemented 12 IPbus slaves. Each of these nodes uses as address the relative IPbus address defined from 0x10 to 0xC0, leaving the 4 LSbs for the addressing of internal registers.

Once the connection and address XML files have been defined, the μHAL applications can be designed using specific objects identifying the connection and the defined <node>s of the address file. The object classes provided by the μHAL library are the following [61]:

- `ConnectionManager`: it identifies the connection file along the code and must be initialized using the directory for the connection XML file.
- `HwInterface`: it represents the connection with the Hardware and must be initialized using the `ConnectionManager` object.
- `Node`: it selects the desired node inside the address XML file and must be initialized using the `HwInterface` object.
- `ValWord<T>`: it is a template class that wraps a data word, received in an IPbus read. Since `T` stands for the data type used by the template class, our application considers `T` as a `uint32_t` type for a 32-bit data word.
- `ValVector<T>`: it is a template class that wraps a block of data, received in an IPbus read. As before, in our case, `T` is defined as `uint32_t` type.

Using all these classes, μHAL works as a delayed dispatch model, in which all the commands for read and write IPbus transactions are queued and concatenated until the dispatch is called or the maximum UDP packet size is reached.

```
1  #include "uhal/uhal.hpp"
2  #include <iostream>
3
4  using namespace std;
5  using namespace uhal;
6
7  ConnectionManager manager("file://path/to/connections.xml");
8  HwInterface hw = manager.getDevice("board.protocol.project_version");
9  const Node *register = &hw.getNode("IPbus_slave.reg1")
10
11 //write 0x2 in the address 0x00000021
12 register->write(0x2);
13
14 //read back
15 ValWord< uint32_t > reg = register->read();
16
17 //send the IPbus transactions
18 hw.dispatch();
19
20 cout << "reg1 value = " << reg.value() << endl;
```

Figure 4.4: Example of the delayed dispatch model used in a C++ program, implementing a μHAL application.

Figure 4.4 shows a simple example of the dispatch mechanism considering, the address XML file of Figure 4.3. As the first step, a `ConnectionManager` and `HwInterface` objects are initialized using respectively a string for the connection file directory and the `ConnectionManager::getDevice(const &string)` method, which returns an `HwInterface` object using the Id parameter of the XML connection file. Then, `register` is a pointer to a constant `Node` object and is initialized by exploiting the `HwInterface::getNode(const &string)` method, which uses the Id for a `<node>` within the address XML file as an internal parameter. The string "`IPbus_slave.reg1`" refers to an internal register of the `IPbus_slave` endpoint, shown in Figure 4.3. This string format provides the sum of the `IPbus_slave` and the `reg1` addresses, defining the address 0x00000021 used inside the firmware to access the corresponding memory location. Then, the initialized `register` object is used to start reading and writing IPbus transactions for the associated `<node>` using the methods:

- `Node::write(const uint32_t &Value)`: it generates the write IPbus transaction of the related `<node>`, considering an IPbus data payload of one 32-bit data.
- `Node::read()`: it generates the read IPbus transaction for the related `<node>`, returning a `ValWord<T>` object that wraps the received 32-bit word.

After setting the wanted transactions, the `HwInterface::dispatch()` method queues each generated UDP packet and sends it to address a specific memory location within the on-chip IPbus, for starting IPbus read or write cycles. As shown in Figure 4.3, a unique dispatch is used to write the 0x2 value inside the register memory location and read back the written value, which is assigned to a `ValWord<uint32_t>` object called

reg. Thus, after the dispatch, the reg wrapped value is defined as 0x2 and is printed on the screen using the value public attribute of the ValWord<T> class. Besides the methods implemented for simple IPbus reading and writing, the library also provides two methods to write and read buffer memory locations, defined across the address XML file using the Mode parameter. These functions are designed with only one parameter used to read or write a predefined number of memory locations:

- Node::readBlock(const uint32_t &aSize): it generates an IPbus packet including IPbus read requests for a maximum of 255 words, covering a total of aSize words request.
- Node::writeBlock(const std::vector<uint32_t> &aValues): it generates an IPbus packet including IPbus write transactions with a payload of 255 words maximum, until the whole size of the vector aValues is written inside the addressed memory buffer.

The 255-word count limit is constrained by the structure of the IPbus transaction packet header, as explained in subsection 2.3.4.

In summary, the μHAL API library is built to queue all the IPbus requests that address different memory locations within the on-chip IPbus. Then the HwInterface::dispatch() method manages the transmission of each request and scans each received response, alerting for possible errors at the UDP or IPbus level.

4.2 Libraries for connection to IPbus slaves

Considering our system, the μHAL library controls the IPbus protocol at the lowest abstraction level, considering the memory locations mapped by the address_table.xml file. Therefore, some front-end libraries were developed to directly manage the IPbus slaves operations for the board, exploiting functions that collect μHAL commands. As the implemented slaves are designed for the configuration and readout of both PicoTDC ASICs, the library dependency structure in Figure 4.5 shows an increasing abstraction level from top to bottom, starting from the front-end libraries of corresponding IPbus slaves up to the library used to control a single PicoTDC device.

The front-end libraries, of Figure 4.5, are explained in the chapter and are organized as follows to control the PicoTDCs:

- **External_signals**: it controls the TDCs_external_signals slave to generate the resets and trigger signals for the external pins of the PicoTDCs.
- **I2C_Master**: it manages the write and read I2C transactions for the I2C_master slave, to configure and control the internal registers of both PicoTDCs.
- **Readout**: it manages the readout of a TDC considering all its 4 ports. In detail, it controls the operations of the 4 PicoTDCX_readoutY, where X value selects the TDC A or B and the Y indicates the connected readout port.

At a higher abstraction level, the Configuration library is designed to provide Configuration objects identifying a list of values for each desired I2C internal register of the PicoTDCs. Therefore, such an object works as a register for all the configuration settings and uses the I2C_Master and External_signals dependencies to configure

the TDCs. At the bottom of the structure, the last `PicoTDC` library shows the highest abstraction level, as it works as a final control point for all the TDCs operations. This library must control the features of all the implemented IPbus slaves, so it depends on all the libraries at lower abstraction levels.

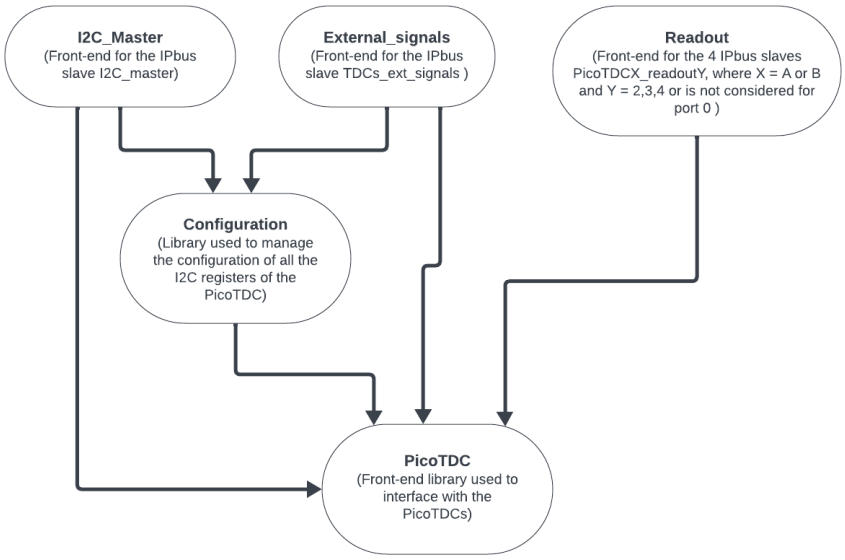


Figure 4.5: Front-end library dependency showing an increasing abstraction level from top to bottom.

4.2.1 The External_signals library

As already mentioned, the `External_signals` library is designed to manage the operations of the `TDCsext_signals` IPbus slave. This library only contains the `External_sig` class that defines objects with the following set of private members:

- `hw`: a pointer to an `HwInterface` object used in the class to dispatch the wanted IPbus transactions.
- `ext_signals`: a pointer to a constant μ`HAL::Node` object that identifies the `<node> EXTERNAL_SIGNALS` within the `address_table.xml` file.

```
12  class External_sig{
13
14  private:
15
16  uhal::HwInterface* hw;
17  const uhal::Node* ext_signals;
18
```

```
25  <node id="EXTERNAL_SIGNALS" description="TDC external signals" address="0x00000040" mask="0xFFFFFF" fwinfo="endpoint;width=0">
26  </node>
```

Figure 4.6: The `External_sig` object private members and the `EXTERNAL_SIGNALS <node>` within the `address_table.xml` file, which maps the `TDCs_ext_signals` IPbus slave.

The EXTERNAL_SIGNALS <node> maps the firmware memory corresponding to the TDCsext_signals slave since address = "0x00000040", as shown in Figure 4.6.

As explained in subsection 3.3.1, the TDCsext_signals slave scans each 32-bit received word searching for a valid command and activates an FSM that generates the corresponding external signal. Therefore, following the scheme of Figure 3.20, the External_sig public methods were designed as a one-word IPbus write transaction, containing a pre-defined command that triggers a specific signal. In particular, Figure 4.7 shows the External_sig::SetOneImpulseTrigger(int tdc_n) function which implements the one pulse software trigger signal for the TDCs. This function design is mainly repeated for all the other methods, providing the following steps:

1. The tdc_n parameter assumes an integer value from 1 to 3, identifying one or both the PicoTDCs.

2. Each valid option uses the ext_signals member to call the Node::write-(const uint32_t&) for the corresponding command.

3. If a valid tdc_n value is set, the HwInterface::dispatch() method dispatches an IPbus write transaction with a valid command.

```
192  void External_sig::SetOneImpulseTrigger(int tdc_n)//Enable one pulse trigger
193  {
194      if(tdc_n > 3 || tdc_n < 1)
195      {
196      std::cout<<"The inserted value is not valid to enable the external signals (1<tdc_n<3)!"<<std::endl;
197      }
198    else if(tdc_n == 1) //Drive the trigger to TDCA
199            {
200            uint32_t trigger_A;
201            trigger_A = 0x00850008; //One pulse trigger for TDCA
202            ext_signals->write(trigger_A);
203            }
204    else if(tdc_n == 2) //Drive the trigger to TDCB
205            {
206            uint32_t trigger_B;
207            trigger_B = 0x00890080; //One pulse trigger for TDCB
208            ext_signals->write(trigger_B);
209            }
210       else if(tdc_n == 3) //Drive the trigger to both the TDCs
211              {
212              uint32_t trigger_both = 0x008D0088;//One pulse trigger for both TDCs
213              ext_signals->write(trigger_both);
214              }
215           //Dispatch the selected trigger option
216           hw->dispatch();
217  }
```

Figure 4.7: External_sig::SetOneImpulseTrigger(int tdc_n) function implementation.

The other developed methods are listed below, showing a name that indicates the generated signals and their characteristics:

- void External_sig::SetLongReset(bool enable, int tdc_n):

it asserts or deasserts the digital reset signal setting the boolean enable to 1 or 0. The tdc_n value selects one or both the TDCs as before.

- void External_sig::SetSoftReset(bool enable, int tdc_n):

it considers the soft mode for the digital reset signal, meaning for a reset assertion of 1 ms maximum length. Both the enable and the tdc_n values work as before.

- `void External_sig::SetPeriodicTrigger(bool enable, uint8_t period, int tdc_n)`:

  it sets a trigger in continuous mode, as explained in subsection 3.3.1. The pulse period distance is set by the `period` variable, while the boolean `enable` switches on (1) or off (0) the trigger.

- `void External_sig::SetBunchCReset(int tdc_n)`:

  it asserts for one pulse length the reset signal of the TDC bunch counter. The `tdc_n` selects one or both the TDCs.

- `void External_sig::SetEventCReset(int tdc_n)`:

  it asserts for one pulse length the reset signal of the TDC event counter. The `tdc_n` selects one or both the TDCs.

4.2.2 The I2C_Master library

The PicoTDCs can be configured via an I2C connection with the `I2C_master` IPbus slave. Each TDC is associated with a 7-bit I2C address, while its configuration and status internal registers are based on 16-bit addresses with a depth of 1 byte. Therefore, the `I2C_Master` library provides a homonymous class that implements specific methods to control the I2C communication with both TDCs.

```
25  class I2C_Master
26  {
27      private:
28       uhal::HwInterface* hw;
29       const uhal::Node* prescaler;
30       const uhal::Node* device_addr;
31       const uhal::Node* rd;
32       const uhal::Node* wr;
33       const uhal::Node* wr_data;
34       const uhal::Node* rd_data;
35       const uhal::Node* status;
36       uint32_t sda_scl[2];
37
38       bool drive_sda = false;
39       bool drive_scl = false;
```

```
15  <node id="I2C_PICOTDC" description="I2C master controller for picoTDC" address="0x00000030" fwinfo="endpoint;width=3">
16      <node id="prescaler"     address="0x00" description="[31:16]: (scl pulse - setup time) / [15:0]: scl pulse" />
17      <node id="device_addr"   address="0x01" description="[6:0]: I2C addr [31] drive_sda [30] drive_scl" />
18      <node id="rd"            address="0x02" description="[8:0]: Number of byte to be read (trigs an I2c read)" />
19      <node id="wr"            address="0x03" description="[31:0]: N/A (trigs an I2C write)" />
20      <node id="wr_data"       address="0x04" description="[7:0]: Add byte in the Tx fifo" />
21  <node id="rd_data"       address="0x05"  description="[7:0]: Byte from Rx fifo" />
22      <node id="status"        address="0x06" mask="0xF" description="[3]: Fault detection / [2]: Rx fifo is empty / [1]: Busy / [0]: Ack_error" />
23  </node>
```

Figure 4.8: `I2C_Master` private members shown with the `I2C_PICOTDC` <node> structure of the `address_file.xml` file.

As shown at the bottom of Figure 4.8, the `I2C_master` slave memory map is identified, within the `address_table.xml` file, by the `I2C_PICOTDC` node and all its internal sub-nodes defined below. In detail, the `I2C_Master` object provides a list of private members, including a pointer to a constant μ`HAL::Node` object for each of the `I2C_PICOTDC` sub-nodes. Furthermore, a pointer to an `HwInterface` object is used as a private member, since all the public functions, designed to interface

with the I2C_master slave, use the HwInterface::dispatch() method. Therefore, each internal register of the I2C_master IPbus slave is identified within the software by a private member, and all the public implemented methods are then based on Node::write(const uint32_t) and Node::read() functions.

The first important public method of the class is the I2C_Master::Setup_Prescaler(), which exploits the prescaler class member to write the 32-bit memory location for the address 0x30, used inside the firmware to define the I2C SCL pulse length and the data_setup variable. This last defines the time length needed to set the SDA value after the SCL falling edge and it must always be lower than the SCL pulse length. The values assigned to these two parameters consider a maximum 16-bit length and the LSb stands for the period of the IPbus clock, running at 31.25 MHz. Since the TDC sustains an I2C communication up to 1 MHz, as explained in subsection 3.3.2, this function was designed to configure the I2C_master slave to support different I2C communication speeds. As shown in the bottom of Figure 4.8, the prescaler <node> is associated with a 32-bit value identifying the SCL pulse length within the bit-positions from 15 to 0, while the data_setup is initialized by the bits from 31 to 16.

```
101  //Function to set the scl and sda pulses length for the I2C_master slave
102  void I2C_Master::Setup_Prescaler()
103  {
104      try
105      {
106       std::string exception = "Setup larger than prescaler!";
107       if (sda_scl[0] >= sda_scl[1]) //The setup variable cannot be larger than the Prescaler
108       throw(exception);
109       else
110       {
111        //Set the data_setup (sda_scl[1]-sda_scl[0]) and the prescaler (sda_scl[1]) variables within the frimware
112         prescaler->write((((sda_scl[1]-sda_scl[0]) & 0xFFFF) << 16) | (sda_scl[1] & 0xFFFF));
113         hw->dispatch(); //dispatch
114         I2C_sleep(PAUSE); //pause used for allow a good I2C transfer
115       }
116      }
117       catch(std::string _exception) //managing of the setup exception
118      {
119       std::cout<<_exception<<std::endl;
120      }
121  }
```

Figure 4.9: I2C_Master::Setup_Prescaler() function design.

Figure 4.9 shows the Setup_Prescaler() function design that uses the sda_scl[2] class member to define the setup and the SCL pulse lengths, explained in Figure 4.10. Specifically, the setup defines the time length between the SDA value setting and the next SCL rising edge, so the expression (sda_scl[1] - sda_scl[0]) in Setup_Prescaler() returns indeed the data_setup value (d_setup in Figure 4.10). After the needed uint32_t value is built at line 112 of the code in Figure 4.9, the Setup_Prescaler() performs:

- the prescaler calling of the Node::write(const uint32_t &) method to generate the write transaction,
- the dispatch of the transaction.

Furthermore, at the function beginning a check for the sda_scl[0] $\geq$ sda_scl[1] is done, ensuring a SCL pulse larger than the data_setup length and avoiding metastability of the I2C connection. After several tests and optimizations, it was found that values of the sda_scl[0] and the sda_scl[1] equal to 20 and 40 respectively gives a stable

I2C communication at:

$$(\texttt{sda\_scl[1]} \times 2 \times T_{IPb})^{-1} \sim 0.4 \text{ MHz}, \tag{4.1}$$

in which the T_{IPb} is the IPbus clock period.

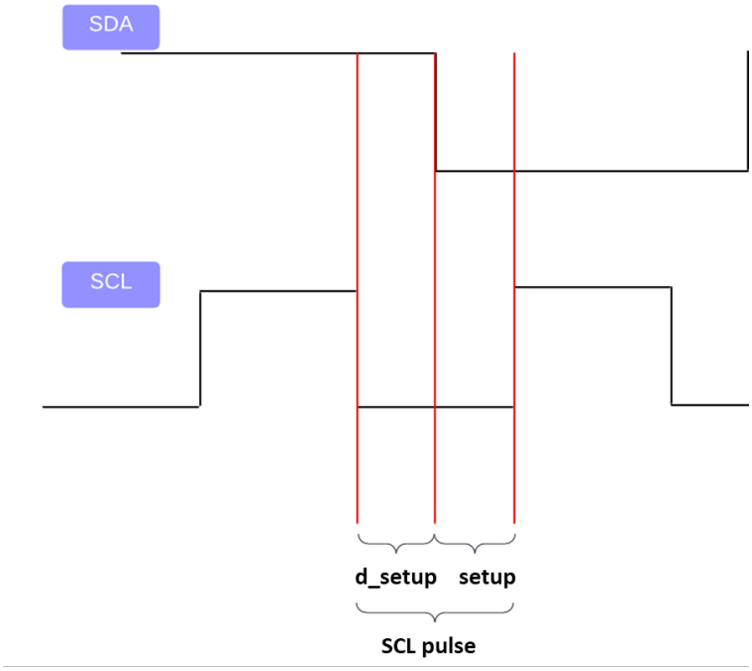
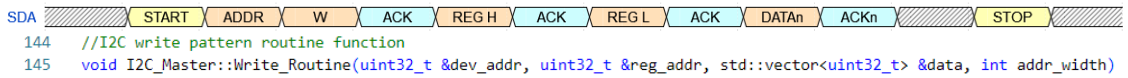


Figure 4.10: Meaning for the `uint32_t` values of `setup`, `setup_d` and the `SCL` pulse in an I2C transaction.

To start an I2C write cycle for the configuration of a PicoTDC register, some `I2C_Master` class members are used at different levels to build an IPbus packet that provides an I2C pattern transaction as the one on top of Figure 4.11. The data bytes must be preceded in the I2C pattern by the high and low bytes of the 16-bit TDC register address. Furthermore, the address is incremented internally for every byte received so it is possible to configure more than one register, addressing only the register with the lowest address. At the bottom of Figure 4.11, the main function that performs the I2C register writing is shown and it exploits the following parameters:

- `dev_addr`: is an `uint32_t` value identifying the 7-bit I2C address of the PicoTDC.
- `reg_addr` and `addr_width`: the int `addr_width` value defines the bytes width of the `reg_addr` parameter. Our case for a PicoTDC considers a 16-bit register address corresponding to `addr_width = 2`.
- `data`: it is a vector of `uint32_t` values that contains the bytes that must be written inside specific TDC registers, using the feature of the internal address increment.

The `I2C_Master::Write_routine` function writes these parameters inside addressed memory locations of the firmware, mapped by the private members of the `I2C_Master` class. In particular, this function uses each member to call an IPbus write




```
144  //I2C write pattern routine function
145  void I2C_Master::Write_Routine(uint32_t &dev_addr, uint32_t &reg_addr, std::vector<uint32_t> &data, int addr_width)
146  {
147     //mask operation to take only the 7 bit corresponding to the I2C device address
148     dev_addr &= 0x7F;
149
150    //debug item not used, as drive_sda and drive_scl are set false by default
151    if(drive_sda)
152       dev_addr |= DRIVE_SDA;
153
154    if(drive_scl)
155      dev_addr |= DRIVE_SCL;
156
157    // (1) write transaction for the I2C device address
158    device_addr->write(dev_addr);
159
160    // (2) write transaction for the register address sending bytes one by one
161    if(addr_width == 1) //other possible devices
162    {
163    wr_data->write(reg_addr & 0xFF);
164    }
165    else if(addr_width == 2) //TDC case
166    {
167     wr_data->write((reg_addr & 0xFF00) >> 8); //high byte
168     wr_data->write(reg_addr & 0xFF); //low byte
169    }
170
171    // (3) write transactions for all the bytes written in the selected register
172    for(uint32_t i : data)
173    {
174    wr_data->write(i & 0xFF);
175    }
176
177    // (4) write transaction to trigger the I2C write cycle in the I2C_master
178    wr->write(0x0);
179    //dipatch of the generated packet
180    hw->dispatch();
181  }
```

Figure 4.11: The I2C write transaction for configuring a TDC register (top) and the design of function `I2C_Master::Write_routine` (bottom).

transaction and store specific information within the registers of the `I2C_master` IPbus slave. Starting from line 157, Figure 4.11 shows the specific order of the IPbus write transactions that reproduce the mentioned I2C frame pattern within the `I2C_master` slave:

1. The `device_address` member calls the write method to write the `dev_addr` inside the IPbus register 0x31.

2. The `reg_addr` 16-bit value is split into 2 bytes: `REG H` and `REG L`. Therefore, the `wr_data` (IPbus address 0x34) calls the write method first for the `REG H` and then for the `REG L`.

3. To provide the next series of written bytes a loop over the `data` vector is set using the `wr_data` member, which calls the write method.

4. The `wr_data` transactions data are written inside a `Tx_FIFO`, as explained in subsection 3.3.2. When the IPbus register (address 0x33) corresponding to the `wr` member is written with any value, all the `Tx_FIFO` memory is read.

After these settings, the `hw` member calls the dispatch and all the information, sent to the `I2C_master` slave, is managed to trigger the `I2C_master_logic` FSM that builds the I2C frame pattern for the addressed TDC. The `I2C_Master::Write_routine` function works as a preliminary stage for:

```
int Write_Regs(uint32_t &dev_addr, uint32_t &reg_addr,
        vector<uint32_t> &data, int addr_width),
```

which also implements the control of the I2C transaction status. Therefore, after the I2C write cycle, the function checks for the error status of the `I2C_master` slave, reading the memory location corresponding to the `status` member (IPbus address 0x36). If an error is found the function returns a 0 and prints an error message on the screen, on the other hand for a successful transaction the function returns another value different from 0. Considering instead an I2C read cycle for a TDC register, the main developed function is:

```
vector <uint32_t> Read_Regs(uint32_t &dev_addr, uint32_t
     &reg_addr, uint32_t read_len, int addr_width),
```

which reproduces the transaction pattern of Figure 4.12 and returns the read data in a `vector<uint32_t>`, where each `uint32_t` value corresponds to a byte in the pattern. The I2C read pattern shows a read command preceded by an I2C write, containing the two address bytes of a TDC register. Furthermore, the register address is also incremented internally for an I2C read operation, so it is possible to read the byte content of multiple registers addressing the one with the lowest address.

The `Read_Regs` function is designed to exploit some of the parameters used for `Write_Regs`, but it also includes the `read_len` value that stands for the number of bytes read during the transaction. The code at the bottom of Figure 4.12 shows the function implementation, including other specific preliminary functions as:

- `void I2C_Master::Read_Wslave(uint32_t &dev_addr, uint32_t &reg_addr, uint32_t read_len, int addr_width):`

  it sets the `dev_addr` and `reg_addr` as before using the IPbus write transactions, and it triggers an I2C read of `read_len` byte payload using an IPbus write transaction for the `rd` member (0x32 IPbus address). The function then uses a dispatch to send the packet and trigger the read I2C transaction for the addressed TDC, storing the data within the texttttRx_FIFO implemented in the `I2C_master` slave.

- `std::vector<uhal::ValWord<uint32_t>> I2C_Master::Read_Routine(uint32_t read_len):`

  It builds a `std::vector<uhal::ValWord<uint32_t>>` object, reading one word from the `Rx_FIFO` at each call of the read method by the class member `rd_data` (IPbus address 0x35). The function stops whenever a payload of length `read_len` has been read.

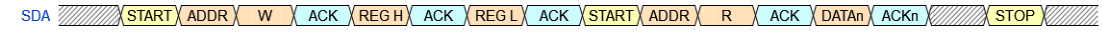
Therefore, the `Read_Regs` function returns the read payload in a `vector<uint32_t>` following the steps below:

1. The function calls the `Read_Wslave` method to trigger a read I2C transaction for the TDC.

2. A control of the transaction status is done using the `status` class member. If no error is found the process continues, otherwise the system provides an alert message for each specific error.

3. At the end, the function calls the `Read_Routine` function and return the `new_data_v vector<uint32_t>` object obtained through the `value` attribute of the μ`HAL::ValWord<uint32_t>` object.

These functions are used within the `Configuration` library to overwrite and control the single I2C register of the PicoTDC. The `Configuration` library uses specific objects, designed to group a variable number of TDC registers, and the `I2C_Master` dependency to configure the PicoTDCs.




```
275 //Function to trigger an I2C read transaction and read the obtained payload
276 std::vector<uint32_t> I2C_Master::Read_Regs(uint32_t &dev_addr, uint32_t &reg_addr, uint32_t read_len,int addr_width)
277 {
278    // (1) Trigger the I2C read transaction to the TDC
279    this->Read_Wslave(dev_addr, reg_addr, read_len, addr_width);
280 
281    // (2) Check the transaction status
282    while(true)
283    {
284       std::string my_exception;
285       uint32_t status_i = this->Get_Status(dev_addr);
286       try
287       {
288          if(status_i & 0x08)
289          {
290            my_exception= "I2C Transaction error: fault detection";
291            throw(my_exception);
292          }
293          else if(status_i & 0x01)
294          {
295            my_exception= "I2C Transaction error: ack error";
296            throw(my_exception);
297          }
298          else if(!(status_i & 0x02))
299          {
300            break;
301          }
302       }
303       catch(std::string except)
304       {
305          std::cout<<except<<std::endl;
306          break;
307       }
308    }
309    // (3) read a data payload of read_len length
310    std::vector<uhal::ValWord<uint32_t>>  data_v;
311    std::vector<uint32_t> new_data_v;
312    data_v = this->Read_Routine(read_len);
313    for( uhal::ValWord<uint32_t> dat : data_v)
314    {
315      new_data_v.push_back(dat.value());
316    }
317    return new_data_v;
318 }
```

Figure 4.12: The I2C read transaction of a TDC register (top) and the design of function `I2C_Master::Read_Regs` (bottom).

4.2.3 The Configuration library

To associate some important characteristics to a specific TDC register, the `Configuration` library provides as the first object the `reg_info` struct. As shown in figure 4.13, a `reg_info` object supplies specific members to describe the TDC register, including mainly two `uint32_t` values for the address (`address`) and stored value (`value`).

```
8   struct reg_info{
9   std::string name = "Default";
10  uint32_t address = 0x0;
11  uint32_t value = 0x0;
12  bool readonly = false;
```

Figure 4.13: Members of the `reg_info` struct.

Since the `value` member is a `uint32_t` variable and the I2C transaction to the TDC can be performed considering the internal increment of the address, `value` has been designed to collect in a 32-bit word the configuration of four consecutive registers setting each byte from LSB to MSB. All the other members of the related `reg_info` object provide instead the following features:

- `name`: name of the register with the lowest address.
- `address`: lowest 16-bit address among the four used to store the 32-bit `value` member.
- `readonly`: boolean variable that defines whether the four registers are read-only (true) or not (false).

This is done to reduce the number of IPbus packet dispatches during the TDCs configuration, as we constraint the I2C read and write transactions, managed by the `I2C_master` IPbus slave, to a maximum of 4-byte data payload instead of only 1 byte.

The `reg_info` struct exploits the `I2C_Master` library dependency to provide the `I2C_master::Read_Regs` and `I2C_master::Write_Regs` re-implementations, performing respectively I2C write and read transactions to the TDC considering a `reg_info` object. Figure 4.14 shows both the function implemented, in which an `I2C_Master` object is used as a parameter and a limit of 4 bytes payload for the transaction is set:

- `void reg_info::Config_Reg(I2C_Master master, uint32_t &device, int reg_n):`

  the `Write_Regs` function is called to perform an I2C write transaction to the TDC addressed by the `device` `uint32_t` value. The `address` attribute of the `reg_info` object identifies the selected TDC register address. Starting from the LSB, the function writes `n_reg` bytes of the `value` `reg_info` member, using a for loop to initialize the `reg_value` vector needed as parameter for the `Write_Regs`.

- `uint32_t reg_info::Control_Reg(I2C_Master master, uint32_t &device, int reg_n)`

  the `Read_Regs` function is called to perform an I2C read transaction to the TDC addressed by the `device` `uint32_t` value. The `address` attribute of the `reg_info` object identifies the selected TDC register address. Then, the `reg_value`

vector is initialized using all the bytes read during the transaction, constrained by the `reg_n` integer value up to a maximum of 4 bytes. Finally, the function returns the `uint32_t my_value`, which is initialized with all the `reg_value` stored bytes starting from the LSB.

Such a struct includes another method mainly used to modify internally a specific number of bits of the `value reg_info` member, at a specific `startbit` position :

```
void reg_info::Mod_Value(uint32_t startbit, uint32_t
        number_of_bits, uint32_t new_bits)
```

Calling this function re-initializes the `value` member with a modified 32-bit word, where specific bits were replaced considering the `new_bits` and `number_of_bits` parameters. The function allows the user to change the configuration of one out of the 4 specific TDC registers, identified by a `reg_info` object.

```
35  //Write up to 4 register from lower to high using as reference the lowest one
36  void reg_info::Config_Reg(I2C_Master master, uint32_t& device, int reg_n)
37  {
38    if(reg_n > 4) //Limit for a reg_info object
39    {
40      std::cout<<"You can write only 4 consecutive registers at a time!"<<std::endl;
41    }
42    else
43    {
44      if(!readonly) //Check for read/write registers
45      {
46        //vector which stores up to 4 uint32_t value identifying each byte of the I2C payload
47        std::vector<uint32_t> reg_value;
48        for(int i = 0; i < reg_n; ++i)
49        {
50         reg_value.push_back((value>>(i*8)) & 0xFF);
51        }
52        //Uses the Write_Reg function to perform the I2C write
53        master.Write_Regs(device, address, reg_value, 2);
54      }
55      else
56      {
57        std::cout<<"Read Only register cannot be written!"<<std::endl;
58      }
59    }
60  }

63  //Read up to 4 register from lower to high using as reference the lowest one
64  uint32_t reg_info::Control_Reg(I2C_Master master, uint32_t& device, int reg_n)
65  {
66    if(reg_n > 4) //Limit for a reg_info object
67    {
68      std::cout<<"You can read only 4 registers at a time!"<<std::endl;
69      return 0x0;
70    }
71    else
72    {
73      //vector which receives up to 4 uint32_t value identifying each byte of the I2C payload
74      std::vector<uint32_t> reg_value;
75      uint32_t my_value = 0x0;
76      //initialize the reg_value vector with the read bytes
77      reg_value = master.Read_Regs(device, address, reg_n, 2);
78      for(int i = 0; i < reg_n; i++)
79          {
80           //initialize the 32-bit value read for this register considering the number of bytes read
81           my_value |= reg_value[i]<<i*8;
82          }
83      return my_value;
84    }
85  }
```

Figure 4.14: The `reg_info::Config_Reg` function (top) and the `reg_info::Control_Reg` function (bottom).

As the `Configuration` library was designed to simplify the configuration of the PicoTDC via I2C connection, it contains a homonymous class that mainly identifies a list of `reg_info` objects resuming all the status and configuration registers of the PicoTDC.

```
27  class Configuration{
28
29  private:
30
31  std::fstream default_config;
32  std::string config_name;
33  std::vector<reg_info> list;
34  int number_of_registers = 0;
```

Figure 4.15: Private members of the `Configuration` class.

To manage a specific configuration text file with "CSV"[1] extension, the `Configuration` class supplies the private members shown in Figure 4.15:

- `default_config`: it is an `fstream` object used to indicate the configuration along the code.
- `config_name`: it is a `string` for the "CSV" directory.
- `list`: it is a `vector<reg_info>` containing the list of the `reg_info` objects for the PicoTDC configuration.
- `number_of_registers`: it is an integer value initialized by default at 0, which keeps track of the `list` vector size.

Therefore, the parameterized constructor of the class was designed to initialize the `default_config fstream` member, using only as input parameter the `config_name` string. Then, the constructor implements the parsing of the `fstream` object, filling the `list` member and updating the `number_of_registers` value.

An example of a "CSV" configuration file can be found in [58], in which a 7-column structure is defined using the semi-colon as the values separator. Each line considers a set of bits used to set a TDC feature and shows the following fields for `list` initialization:

- **Name**: it considers the name for the TDC feature.
- **Bits**: it defines the total number of employed bits.
- **Address**: it provides the TDC register assigned to the `list` elements.
- **Startbit**: it provides the start bit position within a 32-bit word, such as the `value` member of the `reg_info` struct.
- **Default_value**: it defines the bits used to implement a specific TDC feature
- **Access**: it defines if the register is read-only ("r") or not ("rw"). Within the list initialization, this parameter defines the `readonly` member of the `reg_info` objects.
- **Tags**: it considers some tags that describe the TDC features. Such a column is not useful within `Configuration` class and is mainly used to mark all the rows with a similar purpose.

[1] Comma-Separated Values

The `Configuration` parameterized constructor parses the "CSV" file and groups the rows with the same Address field, to initialize a specific `reg_info` for the `list` member of the `Configuration` object. At the end of the process, each `list` element supplies a TDC register address and a `value` member containing 4 configuration bytes, related to the addressed register and the three next registers specified in the PicoTDC manual [27][2].

The methods designed for the Configuration class mainly support modifications for the `list` private member or implement the TDC configuration and control via I2C connection. Therefore some of the most important methods, used mostly along the user interface application, are shown:

- `void Configuration::Modify_Reg(uint32_t address_i, uint32_t value_i)`:

  it searches inside the `list` member for a `reg_info` corresponding to the `address_i` address. Then the `value_i` overwrites the `value` member of the selected `reg_info` object.

- `void Configuration::Modify_Val(uint32_t address_i, uint32_t startbit, uint32_t number_of_bits, uint32_t new_bits)` :

  it searches inside the `list` member for a `reg_info` corresponding to the `address_i` address. Then, the function re-implements the `reg_info::Mod_Value` method, using the other parameter to modify the `value` member of the selected `reg_info` object.

- `void Configuration::Load_Config(uint32_t device, I2C_Master my_master)`:

  using an `I2C_Master` object and addressing the TDC through the `device` parameter, the function runs over the elements of the `list` member and exploits the `reg_info::Config_reg` method to overwrite the I2C internal registers of the addressed PicoTDC, loading the wanted configuration.

- `void Configuration::Control_Config(uint32_t device, I2C_Master my_master)` :

  using an `I2C_Master` object and addressing the TDC through the `device` parameter, the function runs over the elements of the `list` member and exploits the `reg_info::Control_reg` function to compare each `reg_info value` with the TDC register values, read during the I2C transactions. If some mismatches are found the function alerts the user with a message on screen.

Furthermore, both the `reg_info::Config_reg` and `reg_info::Control_reg` were re-implemented for the `Configuration` class, using the same names and adding the feature of searching a specific `reg_info` within the `list` of a `Configuration` object.

The `Configuration` class also provides some methods to control the PicoTDC behavior. Specifically, the PicoTDC needs a power-up routine since when power is applied to the PicoTDC all the registers are in an undefined state. Therefore, to prevent some critical situations, the constructor provides the `magic_word` register (0x0004), which

[2]In the "CSV" configuration file the 0xFFFC register, used for the TDC taps adjustment, is not considered.

must be configured with a specific value to activate all the TDC features. In the power-down mode, the TDC digital section is in a reset state and the ASIC has a low power consumption, as the PLL, the DLL and the hit receivers are disabled. After TDC configuration, the power-up and initialization must follow the scheme shown in Figure 4.16, which defines the following steps:

1. **PLL activation**: the register 0x0134 is used to reset the AFC (Automatic Frequency Calibration) module of the PLL and then restart the AFC, asserting the `pll_afcstart` signal. The PLL needs at least 10 ms to lock its state, then the `pll_afcstart` is deasserted.

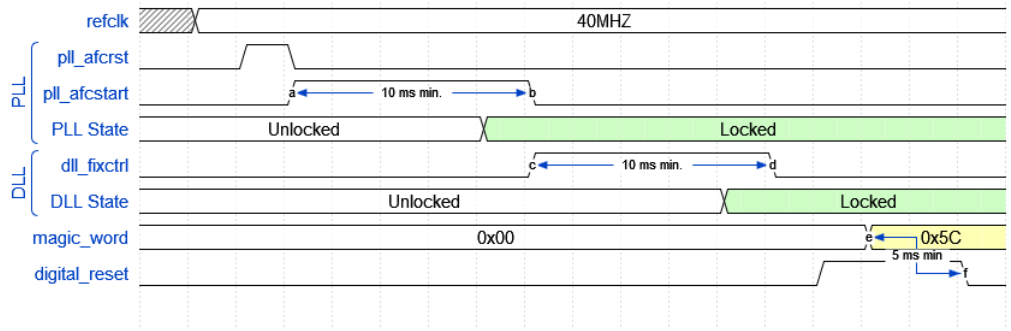
2. **DLL activation**: the register 0x0128 is used to assert the `dll_fixctrl`. The DLL locks its state after a minimum of 10 ms, so the `dll_fixctrl` is deasserted.

3. **Writing magic_word**: the digital reset signal is asserted and the 0x5C `magic_word` is written inside the 0x0004 register, to power-up the TDC. Furthermore, it keeps the digital reset asserted for at least 5 ms, to fully reset the TDC digital section.

All these three steps are accomplished by the `Configuration::Ext_startup` method as shown at the bottom of Figure 4.16. The function uses both the dependency of `I2C_Master` and `External_signals` library, providing an `I2C_Master` and `External_sig` objects as parameters. Furthermore, `Ext_startup` considers other two parameters to select the TDC:

- `device`: it is `uint32_t` variable, which identifies the 7-bit TDC address.

- `tdc_n`: it is used to select the direction of the digital reset signal generated by the related IPbus slave. Therefore, it can assume 2 values: 1 for the TDC A and 2 for the TDC B.




```
824  //Function to start up the TDC
825  void Configuration::Ext_Startup(int n_tdc, uint32_t device, I2C_Master my_master, External_sig my_ext)
826  {
827    reg_info* magic_word = this->Get_Reg(0x0004); //magic_word reg_info object
828    reg_info* PLL = this->Get_Reg(0x0134); //PLL reg_info object
829    reg_info* DLL = this->Get_Reg(0x0128); //DLL reg_info object
830
831    //Low Power mode and configuration of all the TDC input channels
832    magic_word->Mod_Value(0, 8, 0x0); //magic word
833    magic_word->Config_Reg(my_master, device, 1);
834    for(int i = 0; i < 64; i++)
835    {
836      this->Config_Channel(i, device, my_master);
837    }
838
839    //(1) PLL activation (with AFC)
840    PLL->Mod_Value(14, 1, 0x1); //afcrst
841    PLL->Config_Reg(my_master, device, 2);
842    PLL->Mod_Value(14, 1, 0x0); //afcrst
843    PLL->Mod_Value(13, 1, 0x1); //afstart
844    PLL->Config_Reg(my_master, device, 2);
845    std::this_thread::sleep_for (std::chrono::milliseconds(10)); //10 ms
846    PLL->Mod_Value(13, 1, 0x0); //afstart
847    PLL->Config_Reg(my_master, device, 2);
848
849    //(2) DLL activation
850    DLL->Mod_Value(19, 1, 0x1); //DLL fixctrl
851    DLL->Config_Reg(my_master, device, 3);
852    std::this_thread::sleep_for (std::chrono::milliseconds(10)); //10 ms
853    DLL->Mod_Value(19, 1, 0x0); //DLL fixctrl
854    DLL->Config_Reg(my_master, device, 3);
855
856    //(3) Writing magic_word
857    my_ext.SetLongReset(true, n_tdc);     //External digital reset assert
858    magic_word->Mod_Value(0, 8, 0x5C);    //magic word
859    magic_word->Config_Reg(my_master, device, 1);
860    std::this_thread::sleep_for (std::chrono::milliseconds(5)); //5 ms
861    my_ext.SetLongReset(false, n_tdc);  //External digital reset deassert
862  }
```

Figure 4.16: Initialization and power-up routine for the TDC, performed by the `Configuration::Ext_startup` method shown on the code at the bottom.

4.2.4 The Readout library

The `Readout` library was developed to control the readout of each TDC, by managing the behavior of the IPbus slaves connected to each of the 4 readout ports featured by the chip. The `address_table.xml` supplies 4 endpoint nodes for each TDC, designed as the one at the bottom of Figure 4.17. Each endpoint changes only in the address and in the id, matching the parameters of Table 3.4 in section 3.3.

To provide an object that identifies the single readout IPbus slave and its internal registers, the `Readout` library provides the `fifo` class. Figure 4.17 shows on the top the `fifo` class private members, including one pointer to a constant μ`HAL::Node` object for each `<node>` in the `address_table.xml` section shown on the figure bottom. These

```
21  class fifo{
22
23  private:
24
25  std::string fifo_name;
26  uhal::HwInterface* hw;
27  const uhal::Node* control;
28  const uhal::Node* count;
29  const uhal::Node* empty;
30  const uhal::Node* reset;
31  const uhal::Node* read_fifo;
```

```
<node id="PICOTDCA_READOUT" description="PicoTDCA Readout interface for group 1" address="0x00000050" fwinfo="endpoint;width=2">
    <node id="CTR"          address="0x0" description="Control"/>
    <node id="CNT"          address="0x1" mask="0xFFFF" description="Available data words"/>
    <node id="EMPTY"        address="0x1" mask="0x40000000" description="Fifo empty"/>
    <node id="RESET_FIFO"   address="0x2" description="Reset Fifo"/>
    <node id="RXR"          address="0x3" mode="non-incremental" permission="r" description="Readout data"/>
</node>
```

Figure 4.17: Fifo private members and the related node structure of a readout slave, instantiated inside the `address_table.xml` file.

pointers control different features of the readout IPbus slave, while other private members are implemented to manage the `fifo` class functionalities:

- `fifo_name`: it is the `fifo` identifier and is initialized with the endpoint id of the `address_table.xml` file.
- `hw`: pointer to an μ`HAL::HwInterface` object, to perform the IPbus packets dispatch.
- `control`: it identifies the `CTR` `<node>` (not used).
- `count`: it identifies the `CNT` `<node>`, used to know the number of data stored inside the dual clock FIFO of the readout IPbus slave.
- `empty`: it identifies the `EMPTY` `<node>`, used to alert for an empty buffer of the dual clock FIFO.
- `reset`: it identifies the `RESET_FIFO` `<node>`, used to reset the dual clock FIFO memory.
- `read_fifo`: it identifies the `RXR` `<node>` and it has the "non-incremental" attribute, as it is used to read the dual clock FIFO buffer for data readout.

In detail, the `fifo` class is designed to read and control the dual clock FIFO implemented within each readout IPbus slave, as it stores the data coming from one of the four PicoTDC ports. Therefore, two main methods, shown in Figure 4.18, were developed to control the main FIFO operations:

- `std::vector<uint32_t> fifo::Read_Fifomem()`:

  an IPbus read transaction is called by the `count` member, to know the number of data in the read FIFO buffer in a preliminary dispatch. Then, `count_i` variable is initialized with the word count and the `read_fifo` member calls the `Node::readBlock` function, to read all the data within the FIFO if `count_i` > 0. Therefore, this function uses two dispatches to perform the readout and the read

data are returned in a `vector<uint32_t>` object since the TDCs provide 32-bit output data.

- `void fifo::Reset_Fifo()`:

the `reset` member calls the `Node::write` method using as a parameter the value 0x1, to reset the dual clock FIFO in a single dispatch.

```
20  //Function to read the dual clock FIFO memory
21  std::vector<uint32_t> fifo::Read_Fifomem()
22  {
23      uint32_t count_i = 0;
24      //Vector for received data
25      uhal::ValVector<uint32_t> u_data;
26      //Word count inside the read buffer of the FIFO
27      uhal::ValWord<uint32_t> u_count;
28      std::vector<uint32_t> data;
29
30      //(1) Get the number of word inside the read buffer
31      u_count = count->read();
32      hw -> dispatch();
33      count_i = u_count.value();
34
35      //(2) Read all the buffer and return a vector with all the data
36      if(count_i > 0)
37      {
38       u_data = read_fifo -> readBlock(count_i);
39       hw -> dispatch();
40       data = u_data.value();
41      }
42      else
43      {
44      std::printf("Count = 0\n");
45      }
46
47      return data;
48  }
```

```
68  //Function to reset the Dual clock FIFO
69  void fifo::Reset_Fifo()
70  {
71      //assert the reset for 3.2 us
72      reset->write(1);
73      hw -> dispatch();
74  }
```

Figure 4.18: `fifo::Read_Fifomem()` and `void fifo::Reset_Fifo()` functions implementations.

To identify the entire readout of a PicoTDC within a software application, the `Readout` library includes the `TDC_readout` class providing one `fifo` private member for each of the four ports featured by the PicoTDC. The `TDC_readout` members are shown in Figure 4.19 and, besides the four `fifo` members, this class also features the `hw` pointer to an `HwInterface` object, for IPbus packet dispatches, and the `TDC_ID string`, which identifies the PicoTDC. The class parameterized constructor is designed to initialize each `fifo` object considering the related endpoint `<node>` within the `address_table.xml` and the TDC selected by the `TDC_ID` member: "A" for TDC A or "B" for TDC B.

Finally, exploiting the `fifo` members and the `hw` pointer, the functions to control the TDCs readout were designed re-implementing the `fifo::Read_Fifomem` and `fifo::Reset_Fifo` methods:

- `vector< uint32_t> TDC_readout::Read_All_FIFOS()`

each `fifo` member calls the `Read_Fifomem` method and the received data are collected in only one `vector<uint32_t>` object.

- `vector<uint32_t> TDC_readout::Read_One_FIFO(int n_fifo)`:

the `n_fifo` assumes an integer value from 0 to 3 to select the wanted `fifo` member and related TDC port. Then, the selected `fifo` object calls the `Read_Fifomem` method and returns the read data in a `vector<uint32_t>` object.

- `void TDC_readout::Reset_FIFOS(int n_fifo)`

the `n_fifo` selects the wanted `fifo` member that calls the `Reset_Fifo` method to reset the related dual clock FIFO.

```
46  class TDC_readout{
47
48  private:
49
50  fifo fifo0;
51  fifo fifo1;
52  fifo fifo2;
53  fifo fifo3;
54  uhal::HwInterface* hw;
55  std::string TDC_ID;
```

Figure 4.19: `TDC_readout` private members.

4.2.5 The PicoTDC library

The `PicoTDC` is the highest abstraction level library of the front-end dependency structure, shown at the section beginning. This library provides a homonymous class that identifies a PicoTDC ASIC and all its functionalities within a software application. Figure 4.20 shows its private members including an object from each of the main classes explained before for configuration and readout:

- `my_master`: an `I2C_Master` object to implement I2C transactions.
- `my_conf`: a `Configuration` object to manage all the TDC registers configuration at the software level.
- `my_readout`: a `TDC_readout` object to control the TDC readout.
- `device`: it is the I2C 7-bit register assigned to the TDC.

`Externa_sig` objects were not considered members of this class, as the user may want to exploit only the I2C register interface to control the trigger and resets of the PicoTDCs. All the `PicoTDC` methods designed represent the final re-implementation of the ones defined within each library of the dependency structure, including the `External_signals` library. Therefore, the `PicoTDC` methods used mainly in the interactive interface are explained below:

```
11  class PicoTDC{
12
13  private:
14
15  I2C_Master my_master;
16  TDC_readout my_readout;
17  Configuration* my_conf;
18  uint32_t device;
```

Figure 4.20: `PicoTDC` class private members.

- `void PicoTDC::Initialize_Ext(int n_TDC, External_sig my_external_block):`

it assumes a 1 or 2 value for `n_TDC` to select TDC A or B. Then, it triggers the TDC configuration loading and initialization routine, using the `PicoTDC` private members and the object `my_external_block`.

- `void PicoTDC::Write_TDC_Reg(uint32_t address):`

it uses the `address` variable to search the correspondent `reg_info` element within the `list` member of `my_conf`. Then, the `value` member of the selected element is written via I2C inside the TDC correspondent register address. Such a mechanism is thought to overwrite only registers stored in the `my_conf` private member. A similar method is used for the I2C read operation: `Control_TDC_Reg(uint32_t address)`

- `void PicoTDC::Config_TDC_Channel(uint32_t n_channel):`

it works similarly to `Write_TDC_Reg`, but the `reg_info` is selected considering the channel number n_channel, which identifies one of the 64 input channels supplied by the PicoTDC.

- `void PicoTDC::Power_TDC_Down(bool enable):`

if the `enable` variable is set to 1 the TDC turns to power-down mode overwriting the 0x004 register with the value 0x00. Otherwise, if enable is set to 0, the magic word 0x5C is written inside the 0x004 register, setting the TDC to power-up mode.

4.3 User main programs

To manage the features of the PicoTDC board via a single connection with a local host, the first step is to set the XML files for the connection and the addresses. Therefore, the `TDCconnection_file.xml` and `address_table.xml` files were built as explained.

Then, the user interactive interface is designed considering two main user programs to manage the TDCs configuration and readout. In particular, the interactive interface is based on the control of the program settings, using some predefined options and commands performed on the terminal prompt. The settings and statements inside the user programs are designed using the front-end libraries, while the options management is built using the `program_options` library of the C++ `boost` packet. This library is based on some objects and methods able to decode predefined command line options, which triggers some specific code portions at computational time.

4.3.1 User program for TDCs configuration (PicoTOF)

The `PicoTOF` user program is designed for the two PicoTDCs configuration, in which each ASIC is labeled A or B and has a specific I2C address (0x62 or 0x63). As explained in subsection 4.1, a μHAL application needs first a μ`HAL::ConnectionManager` object and a μ`HAL::HwInterface` object, initialized using respectively the `TDCconnection_file.xml` directory and the id `picoTDC_board.chtcp.0` provided in the connection file. To set the different commands and manage the TDC configuration, it is compulsory to declare and initialize the following objects defined in the front-end libraries:

- `My_master`: an `I2C_Master` object that provides the I2C communication between the IPbus slave and the PicoTDCs.
- `TDC_conf`: a `Configuration` object to manage all the TDC internal registers values for a common final configuration of both the TDCs.
- `My_block`: an `External_sig` object that triggers the external signals generated by the related IPbus slave towards one TDC or both.
- `My_readA` and `My_readB`: two different `TDC_Readout` objects used to initialize each `PicoTDC` object and provide a reset for all the dual clock FIFOs implemented in the readout IPbus slaves.
- `TDCA` and `TDCB`: the `PicoTDC` objects that identify PicoTDC A and B

The `PicoTOF` user program includes a first default initialization for the front-end objects used:

1. `my_master` object is initialized setting an `sda_scl[2]` member equal to $\{20, 40\}$, to provide an `SCL` frequency on board of ~ 0.4 MHz (calling the `I2C_Master::Setup_Prescaler()` function).
2. a default "CSV" file is used for the `TDC_conf` initialization. It includes the configuration of specific registers to match the developed firmware. In particular, the file considers a 160 MHz readout rate and the `sync_strobe` mode for the TDCs, as all the readout IPbus slaves were designed to support only this configuration.

Then, all the options listed in Table 4.2 are declared and implemented using the `program_options` dependency. The different options along the code are implemented as if-statements, considered at computational time if the related command has been set on prompt line as indicated in Table 4.2. The `PicoTOF` source code can be found in [58], while below only the options set in Figure 4.21 prompt line, used within the time resolution measurement, are explained.

```
:~$
:~$ ./PicoTOF --triggered --lw 400 360 --falling_edge n  --ch_en fine 62 63 --init A
```

Figure 4.21: Prompt command line including all the options used for the time resolution measurement.

"triggered" and "lw":

At the top of Figure 4.22 the "triggered" if-statement is shown, providing the `TDC_conf` object modification for the `list` member elements related to addresses 0x8 and 0xC, for TDC triggered mode. At the bottom of the statement, the `add_reg` function is implemented to store the modified `reg_info` objects within the `stored_register` vector, as the "set" option can be used to overwrite the TDC registers corresponding to the `stored_register` elements.

At the bottom of Figure 4.22, the "lw" if-statement is shown and it requires two integers to set the latency and window length for the trigger matching function of the TDC. If only 2 integers are provided, the `list` of the `TDC_conf` object is modified only for the `reg_info` element related to the 0x10 address. The same mechanism explained before for the `add_reg` function is implemented at the end. In the example in Figure 4.21, the "lw" option considers 400 and 360 values, identifying 10 μs latency and 9 μs window using the 40 MHz clock period as a time unit.

```
319     if (vm.count("triggered"))
320     {
321         std::cout << "Triggered mode selected: " << "\n";
322         // Change the TDC_conf object for a triggered mode
323         TDC_conf.Modify_Reg(0x8, 0x7A01E6);
324         TDC_conf.Modify_Reg(0xC, 0x2114);
325         // Add the modified reg_info objects to the stored_registers vector (if you want to use the set option)
326         add_reg(stored_registers, intTostr(0x8, 'H'));
327         add_reg(stored_registers, intTostr(0xC, 'H'));
328     }
340     if (vm.count("lw"))
341     {
342         // checking if more than 2 values are provided
343         if (vm["lw"].as<std::vector<string>>().size() > 2)
344         {
345             std::cout << "More than 2 values provided for latency and window lenghts, skipping\n";
346         }
347         else if (vm["lw"].as<std::vector<string>>().size() == 1) // if only one value is provided
348         {
349             std::cout << "Only one value provided for latency and window lenghts, skipping\n";
350         }
351         else //if 2 value are provided
352         {
353             uint16_t lat = 0, win = 0;
354             //conversion from string to integer
355             lat = strToint(vm["lw"].as<std::vector<string>>()[0]);
356             win = strToint(vm["lw"].as<std::vector<string>>()[1]);
357             if (lat > 0xFFF | win > 0xFFF) //check for windows and latency
358                 std::cout << "Latency and window lenghts must be 12 bits maximum, skipping\n";
359             else{
360                 std::cout << "Loading latency and window lenghts: " << lat << " " << win << "\n";
361                 lat_win = win << 16 | lat; // Latency and window lenghts combined
362                 TDC_conf.Modify_Reg(0x10, lat_win); // Change the TDC_conf object for different lat e win
363 
364                 // Add the modified reg_info objects to the stored_registers vector (if you want to use the set option)
365                 add_reg(stored_registers, intTostr(0x10, 'H'));
366             }
367         }
368 
369     }
```

Figure 4.22: If-statements implementing the "triggered" and "lw" options.

"falling_edge" and "ch_en":

The default `TDC_conf` object provides a `list` for a specific configuration, in which all the TDC input channels are disabled and the sampling of a signal for both rising and falling edges is considered. Figure 4.23 shows how the "falling_edge" if-statement is implemented. As before, the `TDC_conf` is modified considering two different modes:

the "y" value for setting the falling edge sampling or the "n" to avoid it. In the TDC resolution measurement, only the arrival time of the signal rising edge was measured, so the "n" string value was set to disable the falling edge sampling.

```
375  if (vm.count("falling_edge"))
376  {
377      char enable = vm["falling_edge"].as<char>();
378      //Set the falling edge sampling
379      if (enable == 'y' || enable == 'Y'){
380          TDC_conf.Modify_Val(0x8, 1, 1, 0x1);
381          // Add the modified reg_info objects to the stored_registers vector (if you want to use the set option)
382          add_reg(stored_registers, intTostr(0x8, 'H'));
383         }//Avoid the falling edge sampling
384      else if(enable == 'n' || enable == 'N'){
385          TDC_conf.Modify_Val(0x8, 1, 1, 0x0);
386          // Add the modified reg_info objects to the stored_registers vector (if you want to use the set option)
387          add_reg(stored_registers, intTostr(0x8, 'H'));
388         }
389      else
390          cout<<"The possible options are only y/Y or n/N"<<endl;
391  }
```

Figure 4.23: If-statements implementing the "falling_edge" option.

To enable different channels with different modes, the "ch_en" option is used. As shown at the top of Figure 4.24, the first decoded parameter is a `string` that identifies a specific mode to enable a TDC channel:

- `-none`: it sets a value for the `list` member of the `TDC_Conf`, to enable a channel for coarse time measurement and to sample a signal on the rising edge.
- `-fine`: it sets a value for the `list` member of the `TDC_Conf`, to enable a channel for fine time measurement and to sample a signal on the rising edge.
- `-fall`: it sets a value for the `list` member of the `TDC_Conf`, to enable a channel for coarse time measurement and to sample a signal on both rising and falling edges.
- `-fineandfall`: it sets a value for the `list` member of the `TDC_Conf`, to enable a channel for fine time measurement and to sample a signal on both rising and falling edges.

Then, the following option parameters identify the desired input channels and are used in the bottom section of Figure 4.24, to modify the `TDC_conf` configuration object. During the TDC resolution measurement only channels 62 and 63, configured for "fine" mode, were used.

As before the `add_reg` function mechanism is implemented for both "falling_edge" and "ch_en" options, as shown in Figures 4.23 and 4.24.

```
253     if (vm.count("ch_en"))
254     {
255         vector<string> channels = vm["ch_en"].as<vector<string>>();
256         int ch = 0;
257         string second_option;
258         bool error = false;
259 
260         //Check for the mode to enable specific channels
261         int pos = channels[0].find("-none");
262         if( pos != std::string::npos)
263             second_option = channels[0].substr(pos, 5);
264 
265         pos = channels[0].find("-fall");
266         if( pos != std::string::npos)
267            second_option = channels[0].substr(pos, 5);
268 
269         pos = channels[0].find("-fine");
270         if( pos != std::string::npos)
271            second_option = channels[0].substr(pos, 5);
272 
273         pos = channels[0].find("-fine&fall");
274         if( pos != std::string::npos)
275            second_option = channels[0].substr(pos, 10);
277     //Enable each channel, set within the option and stored in channels vector, considering the second_option
278     for (int i = 1; i < channels.size(); i++){
279         ch = strToint(channels[i]);
280            if(second_option == "-none"){
281                         std::cout << "Enabling channel: " << ch << "\n";
282                         TDC_conf.Enable_Channel(ch, 1); //Enable the ch for rising_edge and coarse measure
283                         }
284            else if(second_option == "-fine"){
285                         std::cout << "Enabling channel: " << ch << "\n";
286                         TDC_conf.Enable_Channel(ch, 2); //Enable the ch for rising_edge and fine measure
287                         }
288            else if(second_option == "-fall"){
289                         std::cout << "Enabling channel: " << ch << "\n";
290                         TDC_conf.Enable_Channel(ch, 3); //Enable the ch for both rising and falling edges (coarse measure)
291                         }
292            else if(second_option == "-fineandfall"){
293                         std::cout << "Enabling channel: " << ch << "\n";
294                         TDC_conf.Enable_Channel(ch, 4); //Enable the ch for both rising and falling edges (fine measure)
295                         }
296            else{
297                         error = true;
298                         break;
299                      }
300 
301            if (!error)
302               // Add the modified reg_info objects to the stored_registers vector (if you want to use the set option)
303               add_reg(stored_registers, channels[i]);
304     }
305 
306     if (error)
307        cout<<"A wrong option for ch_en command has been chosen, skipping \n";
308 }
```

Figure 4.24: If-statements for the "ch_en" option, showing the two main sections for decoding the channel mode and setting the channel enable values within the `TDC_conf` object.

"init":

After all the command options, used to modify the `TDC_conf` object, have been processed at the computational time, the last option used to trigger the I2C configuration of the TDC registers must be set: "set" or "init". The "set" mainly runs over the `stored_register vector<uint32_t>`, containing all the addresses of modified elements within the `TDC_conf`, to overwrite only the modified TDC registers. Such an option was not used during TDC resolution measurement, but would be useful in the future to overwrite specific TDC registers without fully re-initializing the TDC.

Figure 4.25 shows the if-statement related to the "init" option, which uses a single parameter to identify the PicoTDC that must be re-initialized. Therefore, it uses the `TDCA` and `TDCB` objects to load the new configuration stored in `TDC_conf` and perform the power-up routine.

```
429   if (vm.count("init")){
430
431     const string chip = vm["init"].as<string>();
432     //Selection of the chip that must be initialized
433     if(chip == "A"){
434                 TDCA.Initialize_Ext(1, my_block); //init
435                 my_readA.Reset_FIFOS(4);//reset of all the readout fifos
436                 }
437     else if(chip == "B"){
438                 TDCB.Initialize_Ext(2, my_block); //init
439                 my_readB.Reset_FIFOS(4);//reset of all the readout fifos
440                 }
441     else if(chip == "AandB"){
442                 TDCA.Initialize_Ext(1, my_block);//init
443                 my_readA.Reset_FIFOS(4);   //reset of all the readout fifos
444                 TDCB.Initialize_Ext(2, my_block);//init
445                 my_readB.Reset_FIFOS(4);//reset of all the readout fifos
446                 }
447     else
448         cout<<"The only options allowed are A,B and A&B, skipping\n";
449
450   }
```

Figure 4.25: If-statements implementing the "init" options.

Option name	Value provided	Option description
help	No value provided	It produces the help message with the list of the options.
load	string ("CSV" directory)	It loads a "CSV" configuration file.
load_conn	string (XML directory)	It loads an XML connection file.
piconame	string (id)	It sets id used in the XML connection file.
I2C_addrA	string (TDCA address)	I2C address of picoTDC A
I2C_addrB	string (TDCB address)	I2C address of picoTDC B
sda_scl	vector<string> (setup SCL_pulse_length)	It sets the values for `setup` and `SCL pulse` lengths.
triggered	No value provided	Triggered mode
free-running	No value provided	Untriggered mode
lw	vector<string> (l w)	Trigger latency and window settings
falling_edge	char (y/n)	Falling edge enable
ch_en	vector<string> (mode channels)	It enables the desired channels with [-none, -fine, -fall, -fineandfall] modes for fine resolution and falling edge detection.
en_all	string (mode)	It enables all the channels with [-none, -fine, -fall, -fineandfall] modes for fine resolution and falling edge detection.
dis_all	-	It disables all the channels of a TDC.
startup	string (chip)	Power-up routine for TDC A, B or AandB (also included in the init command)
powerdown	vector<string> (chip 1/0)	Powerdown for TDCA, TDCB or both A,B or AandB (1/0)
init	string (chip)	It configures and initializes the chip (A,B or AandB).
set	string (chip)	It configures the TDC registers modified by previously used options, for a selected chip (A,B or AandB).
rst_fifo	string (chip)	It resets the dual clocks FIFOs, provided by the IPbus slaves implemented for the readout of TDCA and TDCB.

Table 4.2: Options implemented in the `PicoTOF` user program.

4.3.2 TDCs readout user program (PicoRead)

As mentioned in section 2.1, the PicoTDCs can be configured to support a free-running or triggered mode readout. In detail, the TDC data output is provided considering a continuous stream or supplied whenever a trigger signal is sent to the PicoTDC. The triggered mechanism is designed inside the TDC to process TDC measurements only when a trigger signal marks an event, otherwise all the not selected measurements, of each input channel, are discarded automatically. The free-running mode instead skips the trigger matching function of the digital TDC section, pulling out all the TDC measurements of each input channel. As mentioned in subsection 3.3.3, the PicoTDC can also be configured to use one or four differential readout ports, supplying different output data sequences at the software level:

- **Single port readout**: the data output is performed using only chip port 0, keeping the same 32-bit data format as four-port mode. An additional 32-bit separator identifies the 16-channel group, which provides data. Within free streaming mode, the separator is only generated if data are available, while in triggered mode is generated if any trigger is sampled.
- **Four ports readout**: the data are read considering the 4 groups division of the channels and each port takes the output of one 16-channel group.

The `PicoRead` user main program was developed to perform the readout for TDC configured in triggered and single readout port mode. As for the `PicoTOF` user program, this readout interface needs the μ`HAL::ConnectionManager` and the μ`HAL::HwInterface` objects, initialized as explained. This interface was used for the TDC resolution measurement, in which a software trigger was implemented to have a final DAQ system able to count the number of acquired events. Therefore, the `PicoRead` user program is mainly based on these two front-end objects:

- `Ptread`: it is a `TDC_readout` object initialized in our case to perform the TDC readout of TDC A or B.
- `sigMgr`: it is an `External_sig` object, to generate a software trigger for a selected PicoTDC.

To implement the different prompt commands along the code, the `program_options_t` struct was designed as shown in Figure 4.26. In particular, a `program_options_t` object, called `opt`, is instantiated within `PicoRead` program and, using the options listed in Table 4.3, each `opt` member can be initialized to manage the readout for a specific TDC.

```
11  struct program_options_t {
12    std::string connection_filename;
13    std::string chip;
14    int events;
15    bool dump;
16    std::string output;
17  };
```

Figure 4.26: `program_options_t` struct members.

Option name	Value provided	Option description
help	No value provided	It prints help message with the options list.
connection	string (XML directory)	It loads the XML connection file.
chip	string (A, B)	It starts the readout for picoTDC chip A or B.
events	int	It sets the desired number of events when TDC is configured for triggered mode.
dump	bool (1/0)	It prints decoded TDC data on the screen.
output	string (file name)	It generates an output file with "ptdat" extension.

Table 4.3: Options implemented in the `PicoRead` user program.

The full `PicoRead` source code can be found in [58], while the code at the top of Figure 4.27 considers only the main loop implementing the PicoTDC readout. The prompt command line, at the bottom of the figure, represents the main options set for the TDC resolution measurement, generating a binary file with the "ptdat" extension.

```
83  while (iev < opt.events) { //Number of requested events (opt.events)
84    sigMgr.SetOneImpulseTrigger(trigger_lane);   // trigger lane defined by opt.chip
85                                                 // Send a software trigger to the selected chip
86    buff = ptread.Read_One_FIFO(0);   // Read  data
87
88    outb[0]=0x12345678;               // event header
89    outb[1]=iev;                      // event id
90    outb[2]=0x0;                      // separator
91    outb[3]=buff.size()*4;            // data buffer syze
92
93    if ((buff.size()+4)>MAX_OUT_BUF) {
94      std::cout<<"Too many data from FIFO per event, increase output buffer"<<std::endl;
95      exit(EXIT_FAILURE);
96    } else {
97      // in future here we may want to insert a chip header....
98      int i=0;
99      for (uint32_t word : buff) {outb[4+i]=word; i++;}
100   }
101   fout.write((char*)&outb,outb[3]+HEADER_WORDS*4); //write the data within an fout ofstream
102                                                    // the fout directory is defined by opt.output
$ ./PicoRead --chip A --events 10000000 --output file.ptdat
```

Figure 4.27: Loop performing the readout within the `PicoRead` user program and an example of a prompt command line for the `PicoRead` interface.

The "event" option sets a specific number of events in the while statement, which provides a software trigger at each cycle, using the `External_sig::SetOneImpulseTrigger`, for a TDC selected by the "chip" option. Then, the data are read by exploiting the `Ptread` object that calls the method `TDC_readout::Read_One_FIFO`, to read the FIFO connected to the TDC port 0 and store the payload in a `vector<uint32_t>`. In the end, the obtained data for each triggered event are packed, supplying a specific final header that includes four 32-bit words:

- **Event header**: it is a fixed 0x12345678 value that identifies the start of a new event.

- **Event Id**: it specifies the event id among the ones stored during the loop. Inside the loop in Figure 4.27, it is initialized through the `iev` `uint32_t` value.
- **Separator**: it is a 32-bit length separator initialized as 0x00000000.
- **Data buffer size**: it indicates the total number of bytes, stored within a triggered event.

These data packets are printed on the screen using the "dump" option or saved in a "ptdat" file considering the "output" option.

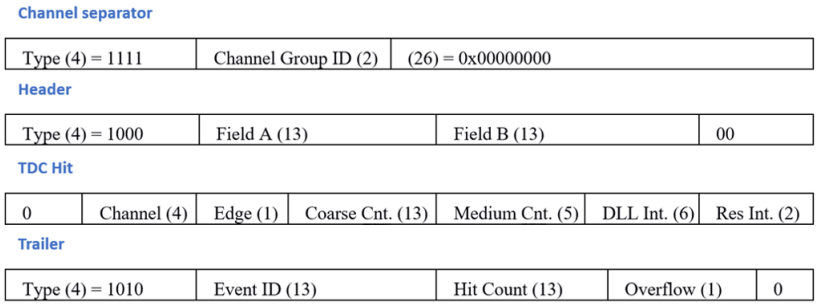


Figure 4.28: Event words sequence sets for our configuration.

Looking at the TDC output, each triggered event is described by a specific sequence of 32-bit words. This sequence can be modified through TDC configuration and the `PicoRead` user program was developed to support the sequence pattern shown in Figure 4.28. Each 32-bit word has a defined division into different fields, so, referring to the figure, the word sequence has the following order:

- **Channels group separator**: it is used to switch between one 16-channel group and the following one. The Type field considers 0xF as an identifier, while the bits 28 and 27 assume the value:
  - 00 for the channels from 0 to 15,
  - 01 for the channels from 16 to 31,
  - 10 for the channels from 32 to 47,
  - 11 for the channels from 48 to 63.
- **Header**: it is provided after each separator and the type field is characterized by a 0x8 value. It considers two 13-bit sections called Fields A and B, which identify the bunch counter and event counter values.
- **TDC Hit**: it represents the measured hit with the following bit division:
  - **Id**: the bit 31, set to 0, identifies the TDC hit word.
  - **Channel**: bit positions from 30 to 27 identify the input channel within the 16-channel group.
  - **Edge**: bit 26 shows if the measurement was performed on the rising edge (1) or the falling edge (0) of a pulse.

  - **Value**: bits from 25 to 0 are used for the full-time measurement of TDC, in which each specific measurement performed by the TDC is shown.

- **Trailer**: it is the event summary for every separator and is identified by the type field 0xA. It shows the event counter value and the number of hits found for each separator. The Overflow bit is not used for our default configuration so it is always set to 0.

Executive summary

The software design was developed to finally provide the user with a simple interactive interface for controlling the hardware feature of the PicoTDC card. Starting from the μHAL back-end library, some front-end libraries were designed to have higher abstraction level commands for the implemented IPbus slaves. The connection between a host and the board is supported by the Control Hub software application, which also manages the internal connection with a mono-thread μHAL application. As a final interactive interface, two user programs were designed for the configuration and readout of the PicoTDCs. These two were implemented to allow the user to control the PicoTDC features by some prompt command lines, reducing the complexity of the DAQ system operations.

Chapter 5

Resolution measurements

Since the PicoTDC board has to be used in future test beams to test particle sensors, a setup for the PicoTDC resolution measurement was built to prove the reliability of the developed firmware and software. Such a measurement was also important to test the board performances and find the effective resolution capabilities of the PicoTDCs ASICs, configured in their fine resolution mode. Previously, the ALICE Bologna group tested the PicoTDC resolution only using a test card provided by CERN, estimating a final resolution of less than 5 ps [62].

In general, the resolution measurement of a TDC ASIC is performed considering repeated measurements of the same known and stable time interval. The final histogram reporting all the occurrences must show a Gaussian shape with a mean set around the known time interval value. Fitting the histogram, the standard deviation parameter is taken as the effective resolution of the ASIC, while the mean represents the time interval measured.

The first chapter section shows how the measurement setup was built, while the second one explains the analysis done for the datasets obtained for both PicoTDCs A and B. At the end of section 2 the final results are discussed, providing some conclusions on the PicoTDC resolutions and the reliability of the developed DAQ system.

5.1 Experimental setup

The setup shown in Figure 5.1 provides two differential signals with the same frequency on two input channels of one PicoTDC. One of these two signals is delayed using a time delay programmable line, to define a fixed time difference between the two. Therefore, by setting different delays, many stable time differences, in a specified time range, can be measured for resolution estimation. Our setup is built with the following devices:

1. TTi PLL-105P: it is a linear DC Power supply to power the PicoTDC board with a voltage of 11.0 V.

2. PicoTDC board: Figure 5.1 shows the test board, in which the TDCs are highlighted in red, while the PolarFire FPGA and the Ethernet connector are highlighted in purple. The Ethernet connector is wired to a local host with Linux OS, providing the libraries and the designed software (including the Control Hub).

3. The mezzanine adapter: in Figure 5.1, it is connected to TDC B through an FMC connector. This is an adapter board developed by the INFN electronics laboratory

of Bologna. In detail, it provides two MCX connectors on either side directly linked to the FMC connector pins assigned for TDC B differential input channels: 62 and 63 (the same applies for TDC A). The card also hosts an IDC34 connector and two VHDCI connectors (like the one mounted on the ALICE-TOF TRM).

4. Si5341-D evaluation board: this card generates precise clock signals, running at a configurable frequency. In our case, such a board is configured to supply two sub-LVDS18 signals.

5. Electromagnetic trombone: it is a programmable delay line with a range from 0 to 625 ps and a resolution of 0.5 ps. The micro terminal, placed on top of the device, is used to modify the delay supplied on the input signals.

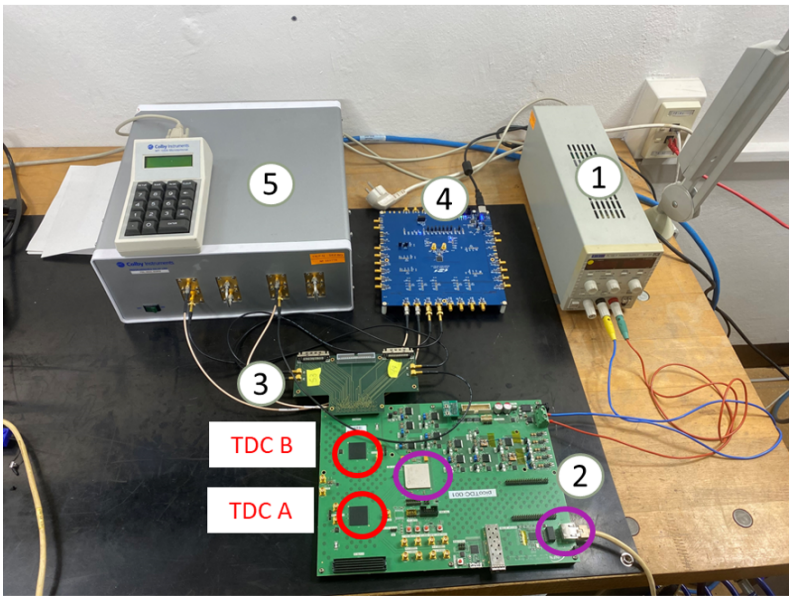


Figure 5.1: Experimental setup built for the resolution measurement of TDC B.

To explain better the setup workflow, Figure 5.2 shows a scheme of how the setup is organized for TDC B configuration and readout, matching Figure 5.1. As mentioned, the Si5341-D EVB board generates two sub-LVDS18 signals, with a common voltage of 0.9 mV compatible with the TDC manual indications. These two differential signals have different directions in the setup:

- `Signal1` (red) feeds the two inputs of the electromagnetic trombone using two SMA to SMA cables.

- `Signal2` (blue) is sent to the mezzanine adapter using two SMA to MCX cables. The generated clock signal is sent to the TDC B channel input 62, using the adapter plugged into the FMC connector.

Then, `Signal1` passes through the electromagnetic trombone acquiring the configured delay time plus an offset due to the trombone mechanism. The delayed `Signal1` is sent to the other two MCX connectors of the mezzanine (using two SMA to MCX cables), which are linked to the TDC B input channel 63. The explained signal-generation mechanism is also valid for TDC A, considering always the differential input channels 62 and 63.

Therefore, the setup structure is designed to measure the arrival time of both `Signal1` and `Signal2` clocks. The difference between these two measurements defines the fixed time interval, to perform the resolution estimation. The configuration and readout of both the PicoTDCs are triggered by the developed user main programs, exploiting the Ethernet connection with the local host.

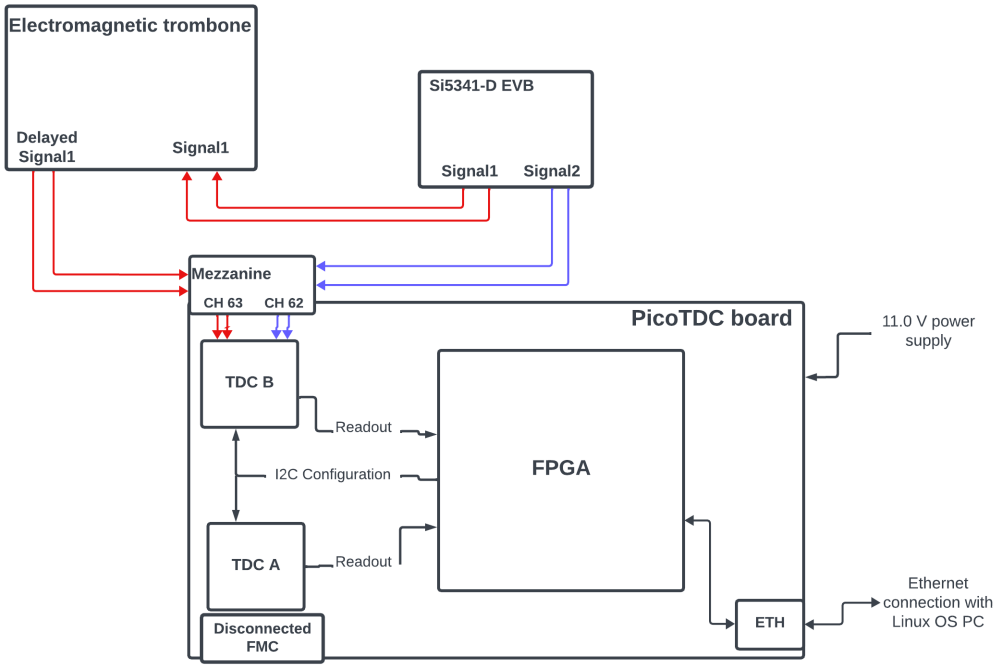


Figure 5.2: Test setup scheme for resolution test of the PicoTDCs mounted on PicoTDC board.

5.1.1 Si5341-D evaluation board

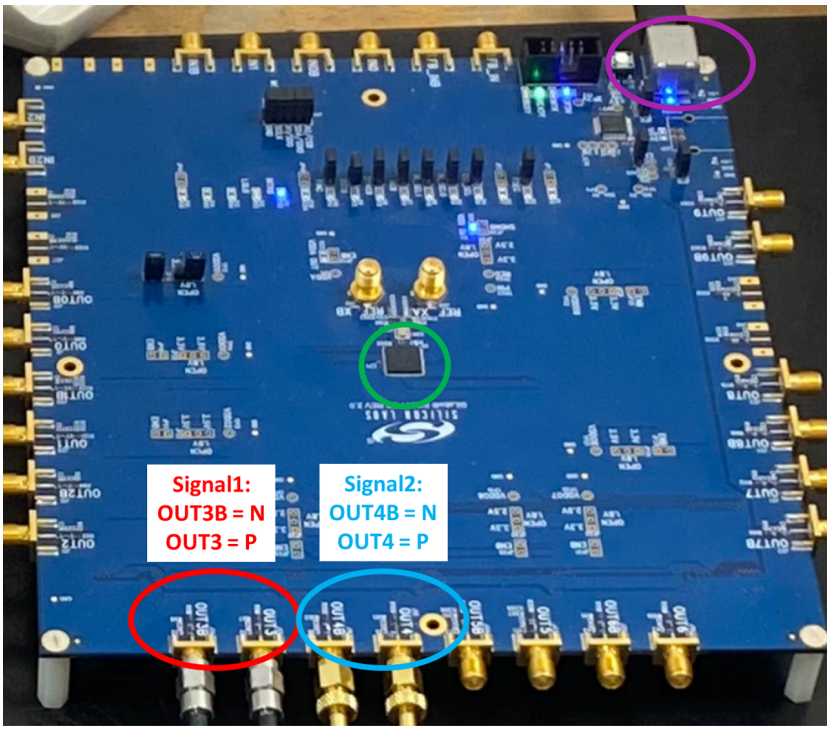


Figure 5.3: Evaluation board that features the Si5341-D frequency generator, highlighted in green. The USB connector and the connected outputs are highlighted with different colors.

The board, used for the experimental setup, is produced by Skyworks Solutions and hosts the Si5341-D frequency generator (highlighted in green in Figure 5.3). This chip can generate differential output clock frequencies from 100 Hz up to 1028 MHz, starting from the input of a crystal oscillator, supplied on board and running at a frequency of 48 MHz [63] [64]. The Si5341-D chip is designed to produce precise single-ended and differential clock outputs, ensuring a 90 fs jitter phase.

The evaluation board is powered up through a USB connector (highlighted in purple in Figure 5.3) and provides as output 10 possible SMA female connectors, directly routed to the chip. Furthermore, the USB connection is used for chip configuration, to set the desired frequency and voltage standard (LVCMOS, LVDS, sub-LVDS ecc.) for each enabled output. The ClockBuilder Pro software application works as a configuration user interface for the board and the chip. This software was designed for Windows OS, so a further Windows machine was employed to prepare the setup.

As shown in Figure 5.3, our setup uses the outputs OUT3B and OUT3 for the `Signal1`, sent to the electromagnetic trombone, while OUT4B and OUT4 are exploited for `Signal2`, sent directly to the mezzanine adapter. The label N and P stands for the negative and positive polarities of the outputs. Finally, the board configuration provides 2 sub-LVDS18 clock signals running with a frequency of 100 KHz.

5.1.2 Electromagnetic trombone

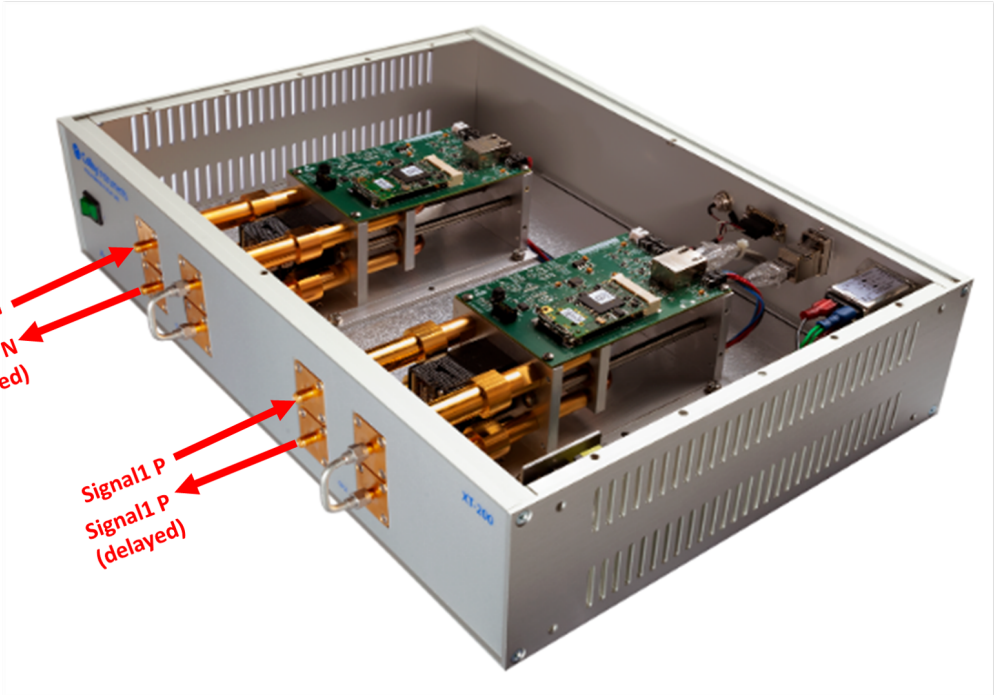


Figure 5.4: Electromagnetic trombone internal structure [65].

The electromagnetic trombone is the XT-200 programmable delay line, produced by Colby Instrument. This device is based on a single unit called "electromechanical trombone", which uses a stepper motor mechanism to set a longer or shorter path for an input electric signal. The added path length induces a delay in the input signal, which is then provided to the output connector.

The XT-200 can be configured for a specific delay value from 0 to 625 ps, considering a step resolution of 0.5 ps and a wideband signal frequency input from DC to 18 GHz [65]. In our experimental setup, the delay value can be modified through a further device, called MT-100A MicroTerminal and connected directly to the XT-200 through an RS-232 serial port. The signals instead are routed in and out of the device through female SMA connectors.

Since we needed to delay a differential signal, the XT-200 features two input channels instrumented with an "electromechanical trombone" unit. Therefore, as shown in Figure 5.4, both the negative (N) and positive (P) polarities signals, describing `Signal1`, pass through a delay line, getting out with a delay equal to an offset plus the configured delay.

5.2 Data acquisition and analysis

The data acquisition was performed using the user main programs developed for PicoTDCs configuration and readout: `PicoTOF` and `PicoRead`. Setting the signals generation system, explained in section 5.1, and configuring the electromagnetic trombone for different delays, a dataset of 10^7 events was collected for each specific delay value. A total of 9 datasets were collected for each TDC, covering the delay range of the electromagnetic trombone from 0 to 600 ps. A final analysis was performed to estimate the effective TDC resolution for a single-channel time measurement.

5.2.1 DAQ workflow

The data acquisition was based on 2-channel time measurements for a single PicoTDC. The Si5341-D evaluation board generated the first two differential clock signals, with a chosen frequency of 100 KHz. Then, the trombone set a given delay for both polarities of one among the two generated signals. Finally, the signals fed channels 63 and 62, as given by the adapter schematic plugged into the FMC connector.

As the first step, before starting the acquisition, the connected PicoTDC must have a specific configuration to match the firmware and the readout software requirements. Therefore, the developed user program provided a default configuration with the following instructions:

- Single Port Readout mode was set, as the readout user program was developed to read data only from port 0 of the connected TDC.
- A readout rate of 160 MHz and the Sync strobe mode were set to match the features of the IPbus slave used for the port 0 readout.
- The PicoTDC was also configured to provide both data measurements using fine (3.05 ps) or normal binning (12.2 ps) (depending on how the channels would be enabled using "ch_en" or "en_all" options)

Besides these default conditions, the TDC configuration was modified using the `PicoTOF` implemented options, shown in subsection 4.3.1. Therefore, the following commands were set on the terminal command line, before calling the "init" option for the initialization of the related TDC:

- `--triggered`: it was set as the `PicoRead` software, designed for the readout, implements internally a software trigger. Furthermore, the `PicoRead` was thought to receive frames data pattern for a triggered event, as the one shown in subsection 4.3.2.
- `--lw 400 360`: setting a trigger latency of $400 \times T_{clk\_board} = 10\,\mu s$ and a trigger window of $360 \times T_{clk\_board} = 9\,\mu s$, where $T_{clk\_board} = 25\,ns$ is the period of the 40 MHz clock that feeds the TDCs. Such an option matched the trigger latency with the period provided for the generated TDCs input.
- `--falling_edge n`: it was set to allow TDC time measurement only on the rising edge of the generated clock signals.
- `--ch_en -fine 62 63`: it enabled TDC input channels 62 and 63, considering the fine resolution mode and a time measurement performed on the rising edge of the generated clock signals.

Figure 5.5 shows an example of a TDC-triggered event in which `Signal1` and `Signal2` are the 100 KHz clock signals feeding TDC channels 62 and 63. The trigger is a 25 ns pulse signal pulled by the `PicoRead` software, and the TDC was configured to sample only the rising edge of the signals. Therefore, only the `Signal1` and `Signal2` rising edges are provided when `Trigger_window = '1'` and the delay, obtained by the difference between the two TDC measurements (as highlighted in red in Figure 5.5), is the final observable. By setting the trigger latency to 10 μs and the trigger window to 9 μs, an event can be considered valid if only one measurement per channel is performed, as both

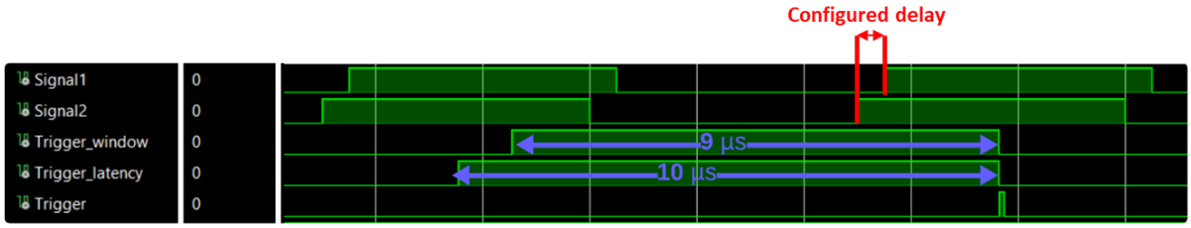


Figure 5.5: Signals waveforms within a valid TDC-triggered event for the defined configuration.

`Signal1` and `Signal2` have a frequency of 100 KHz and two measurements are needed to have a time difference.

After setting the TDC configuration, the setup was ready to acquire data using the `PicoRead` user program. The data acquisition conceived a dataset of 10^7 triggered events for each configured delay. To almost cover the dynamic range of the electromagnetic trombone, 9 different delays were configured within the 0-600 ps time range collecting a total of 9 datasets for each TDC. Starting from the 0 ps value and considering a 100 ps shift, 7 different delays were chosen up to 600 ps, while two additional delays of 10 ps and 20 ps were used to test the PicoTDC 2-channel measurement for smaller time differences. Therefore, as mentioned in section 4.3, the acquisition started considering the chip connected to the setup and repeating the following option commands line for each of the 9 delay values:

- `--chip A` or `--chip B`: selecting the connected chip for the readout.
- `--events 10000000`: it was set to send 10^7 software triggers to the selected TDC and collect each event, with or without TDC measurements.
- `--output file.ptdat`: it saved the collected acquired data of an event using the header format explained in subsection 4.3.2. The data were saved in a file with "ptdat" extension.

Each delay acquisition took ∼2 hours, showing reliable operations of the DAQ system both at software and firmware levels, without any communication interruption or timeout. In the end, 9 "ptdat" extension files were obtained for each TDC, corresponding to the 9 configured delay values.

5.2.2 Analysis and results

The TDC output for each channel separator corresponds to a sequence of 32-bit words as the one explained in subsection 4.3.2. Therefore, the analysis individually considers each of the 9 datasets acquired for each TDC and it is based on a selection mechanism for events showing a word sequence as the one in Figure 5.6 (obtained using the "dump" option of the `PicoRead` user program). Since, by TDC configuration and setup characteristics, all the valid triggered events provide only one measurement per TDC input channel, all the events in which one or both channels did not acquire data are considered invalid and must be dropped.

The analysis program is designed to scan all the 10^7 events stored in the "ptdat" file, searching for events providing only 2 hit time measurements in the frames sequence of

separator 0xFC000000 (highlighted in yellow in Figure 5.6). As a further condition, it selects the events that show only one hit for both channels 14 and 15, which correspond to the TDC input channels 62 and 63 used in the setup. As a final result, $\sim 10^6$ invalid events were dropped providing a final dataset for the TDC time resolution estimation of $\sim 9 \times 10^6$ valid events.

```
 Header: 12345678 BC2 0 38
00: 11110000000000000000000000000000  F0000000 SEPARATOR: 0 (group)
01: 10001110110100000010111100001000  8ED02F08 HEADER 1
02: 10100101111000010000000000000000  A5E10000 TRAILER: 962 (Event ID) | 0 (HIt count)
03: 11110100000000000000000000000000  F4000000 SEPARATOR: 1 (group)
04: 10001110110100000010111100001000  8ED02F08 HEADER 1
05: 10100101111000010000000000000000  A5E10000 TRAILER: 962 (Event ID) | 0 (HIt count)
06: 11111000000000000000000000000000  F8000000 SEPARATOR: 2 (group)
07: 10001110110100000010111100001000  8ED02F08 HEADER 1
08: 10100101111000010000000000000000  A5E10000 TRAILER: 962 (Event ID) | 0 (HIt count)
09: 11111100000000000000000000000000  FC000000 SEPARATOR: 3 (group)
10: 10001110110100000010111100001000  8ED02F08 HEADER 1
11: 01110100000011010000101101110101  740D0B75 HIT: 14 (channel) | 1 (edge) | 2607.45 ns (hit converted value)
12: 01111100000011010001001011011001  7C0D12D9 HIT: 15 (channel) | 1 (edge) | 2613.22 ns (hit converted value)
13: 10100101111000010000000000001000  A5E10008 TRAILER: 962 (Event ID) | 2 (HIt count)
```

Figure 5.6: TDC 32-bit words sequence provided for each valid triggered event.

Among all the 32-bit words describing an event, only the TDC time measurements were extracted by the analysis program. As the valid event definition considers a pair of time measurements acquired for channels 62 and 63, a time difference is obtained for each valid event identifying the time delay induced by the electromagnetic trombone. To estimate the final TDC resolution, the ROOT analysis tool is used to produce a histogram filled with the measured time differences for each configured delay. A Gaussian fit was applied to each histogram using the following PDF[1]:

$$f(x) = \frac{\text{Const}}{2\sigma\sqrt{\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{5.1}$$

in which the Const variable is the Gaussian area, while the mean μ and the standard deviation σ represent the measured time difference and the estimated resolution for a 2-channel time measurement.

To obtain the offset generated by the electromagnetic trombone, a first 0 ps delay measurement was performed. Figure 5.7 shows the 2 histograms found for TDC A and B, in which the 2-channel time differences are set on the x-axis, in TDC bin units, and the y-axis shows the entries per bin. The fit almost reproduces the shape of the histograms and the measurements obtained are shown in Table 5.1, not considering the related errors as they are negligible.

The time difference between the measurements of 2 different TDC channels provides the resolution:

$$\sigma_{time(2ch)} = \sqrt{\sigma_{ch1}^2 + \sigma_{ch2}^2}, \tag{5.2}$$

where the single TDC channel measurements are considered independent, so their resolutions are summed in quadrature. Since each TDC channel provides the same binning, the following assumption is considered:

$$\sigma_{ch1} = \sigma_{ch2} = \sigma_{time(1ch)} \tag{5.3}$$

and the final resolution $\sigma_{time(1ch)}$ for a single-channel time measurement is derived from equation 5.2 as:

$$\sigma_{time(1ch)} = \frac{\sigma_{time(2ch)}}{\sqrt{2}}, \tag{5.4}$$

[1]Probability Density Function

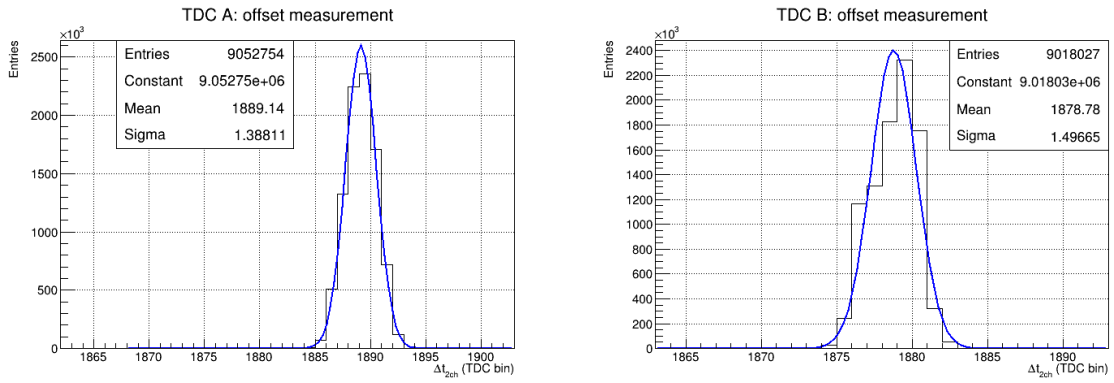


Figure 5.7: Trombone offset measurements for TDC A and TDC B.

in which the $\sigma_{time(2ch)}$ values are displayed in the third column of Table 5.1. The final offset and the $\sigma_{time(1ch)}$ resolution estimates, converted into ps units, are shown in the last two columns of the table. These conversions were made by multiplying the measured mean and the estimated $\sigma_{time(1ch)}$ values by the TDC binning factor of 3.05 ps. Although the same setup was used for offset measurement for both TDC A and TDC B, the measured offsets differ because the TDC channels were not calibrated to the same value before the data acquisition began.

TDC	Mean (TDC bin)	Resolution (TDC bin)	Set delay (ps)	Measured offset (ps)	Estimated resolution $\sigma_{time(1ch)}$ (ps)
A	1889.14	1.39	0	5761.88	2.99
B	1878.78	1.50	0	5730.28	3.23

Table 5.1: Fit results of the TDC A and B offset measurements (errors are negligible).

Taking the measured offsets as references, the same process was repeated for all the other 8 datasets, finding the histograms of Figure 5.8 for TDC A and Figure 5.9 for TDC B. Then all the estimated values found using the Gaussian fit are shown in Tables 5.2 and 5.3, where the related errors were not considered as negligible.

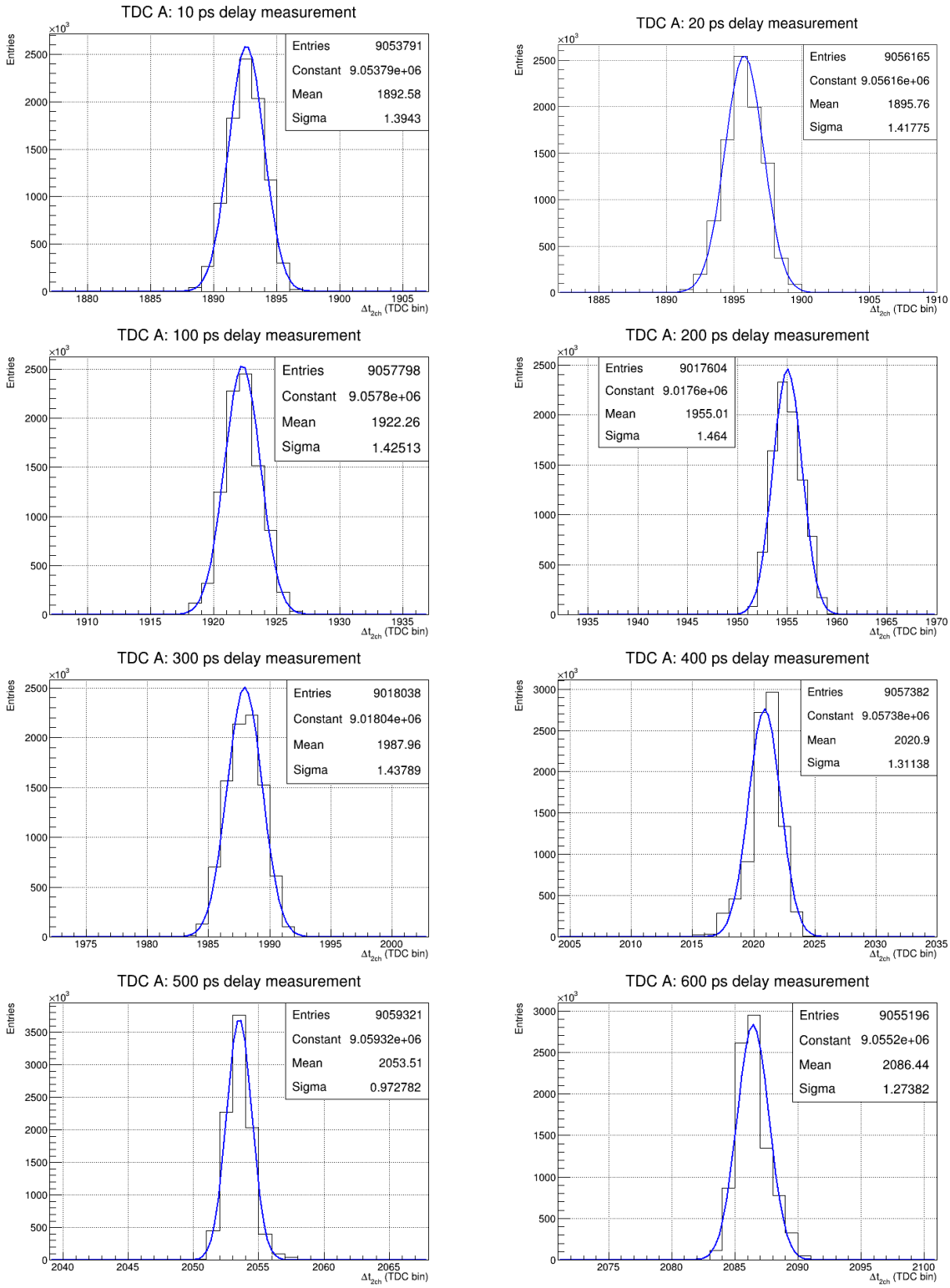


Figure 5.8: Histograms obtained for TDC A, considering different configured delays.

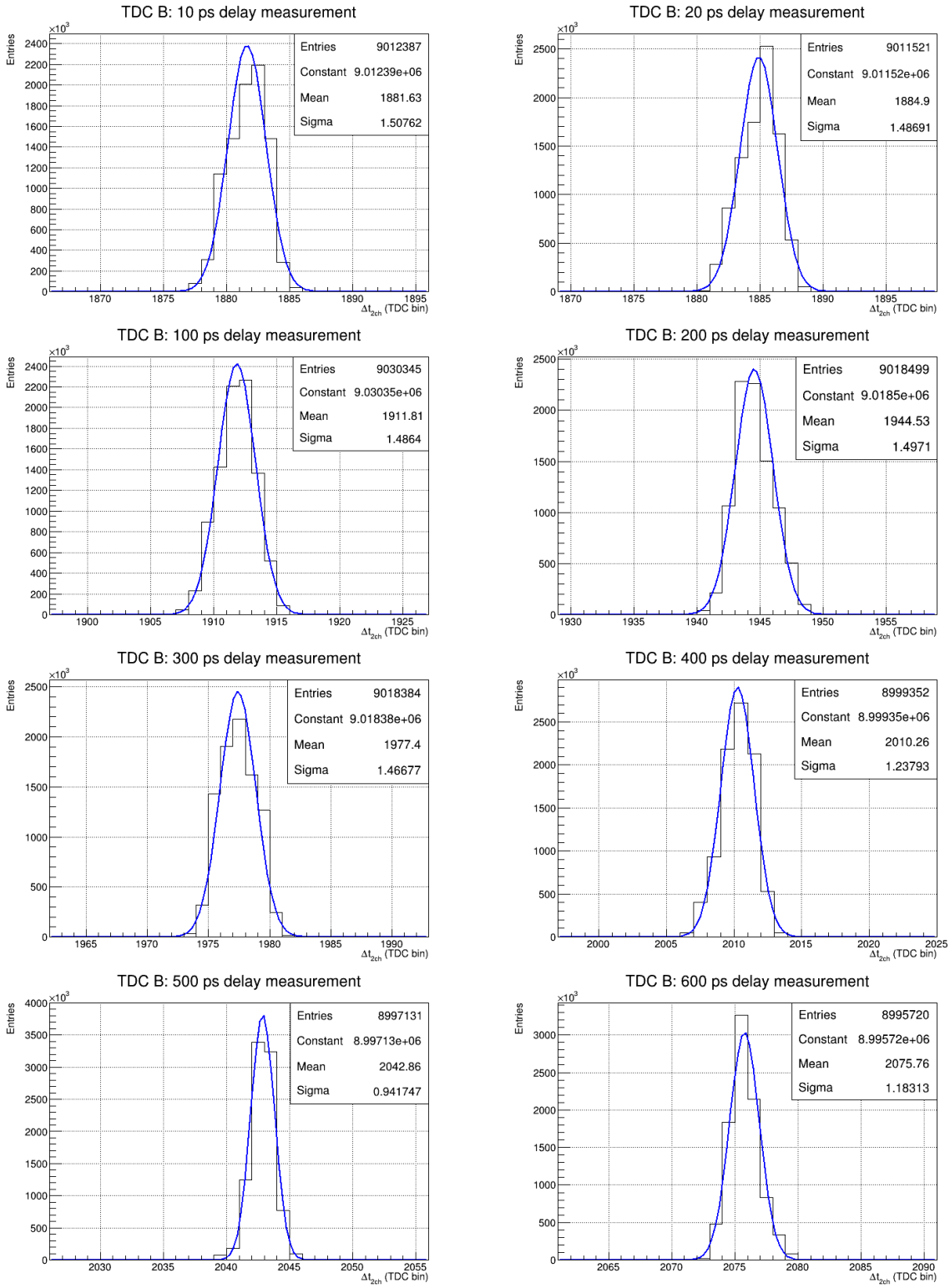


Figure 5.9: Histograms obtained for TDC B, considering different configured delays.

Mean (TDC bin)	Resolution (TDC bin)	Set delay (ps)	Measured delay (ps)	Estimated resolution $\sigma_{time(1ch)}$ (ps)
1892.58	1.40	10	10.49	3.01
1895.76	1.42	20	20.19	3.06
1922.26	1.43	100	101.02	3.07
1955.01	1.46	200	200.90	3.16
1987.96	1.44	300	301.40	3.10
2020.90	1.31	400	401.86	2.83
2053.51	0.97	500	501.33	2.10
2086.44	1.27	600	601.77	2.75

Table 5.2: Estimated values found by fitting the measurements done with TDCA (errors are negligible).

Mean (TDC bin)	Resolution (TDC bin)	Set delay (ps)	Measured delay (ps)	Estimated resolution $\sigma_{time(1ch)}$ (ps)
1881.63	1.51	10	8.69	3.25
1884.90	1.49	20	18.67	3.21
1911.81	1.49	100	100.74	3.21
1944.53	1.50	200	200.54	3.23
1977.40	1.47	300	300.79	3.16
2010.26	1.24	400	401.01	2.67
2042.86	0.94	500	500.44	2.03
2075.75	1.18	600	600.76	2.55

Table 5.3: Estimated values found by fitting the measurements done with TDCB (errors are negligible).

The fit performed almost reproduces the histogram found for each configured delay, estimating the 2 channel time differences shown in the "Mean" column of Tables 5.2 and 5.3. The "Measured delay" column, shown in both the tables, contains instead the estimated delay values obtained using the following equation:

$$\text{Measured delay} = (Mean_{delay} - Mean_{offset}) \times \text{LSb}, \tag{5.5}$$

in which $Mean_{delay}$ is the mean value for a specific delay measurement and $Mean_{offset}$ is the estimated value for the offset induced by the trombone (LSb = 3.05 ps). The measured delays, for both the PicoTDCs, are in excellent agreement with the expected ones, demonstrating the good performance of the PicoTDC board and the DAQ system. With the exception of the two lowest values of Table 5.3, they are in agreement within 2 ps with the expected values.

The estimated 1-channel resolution $\sigma_{time(1ch)}$ is always provided by the Equation 5.4, and the values, estimated for all the configured delays, are shown in the last column of the Tables 5.2 for TDC A and 5.3 for TDC B. To understand better the TDC A and B performances over the 0-600 ps delay range, the graphs, in Figure 5.10, plot the $\sigma_{time(1ch)}$ estimated values as a function of the measured 2-channel time differences, both

converted to ps units. Starting from the offset measurement, it is clear how the estimated resolutions float around a specific value, which can be inferred considering the average of all the resolutions and associated with its standard deviation:

- TDC A: $\sigma_{time(1ch)} = 2.90 \pm 0.33$ ps, where the worst estimated resolution is $\sigma_{time(1ch)} = 3.16$ ps.
- TDC B: $\sigma_{time(1ch)} = 2.95 \pm 0.43$ ps, where the worst estimated resolution is $\sigma_{time(1ch)} = 3.25$ ps.

These results confirmed the excellent performance of both PicoTDCs integrated on board, improving the previous results obtained by the ALICE group using an external test PicoTDC card plugged into an FPGA-based board.
In conclusion, the PicoTDC board meets the requirements as a test environment for the future development of the TRM2 card. At the same time, the designed DAQ system ensures stable and reliable operations at both software and firmware levels, making it a valuable resource for test beams or laboratory analysis implying generic sensors and related front-end electronics.

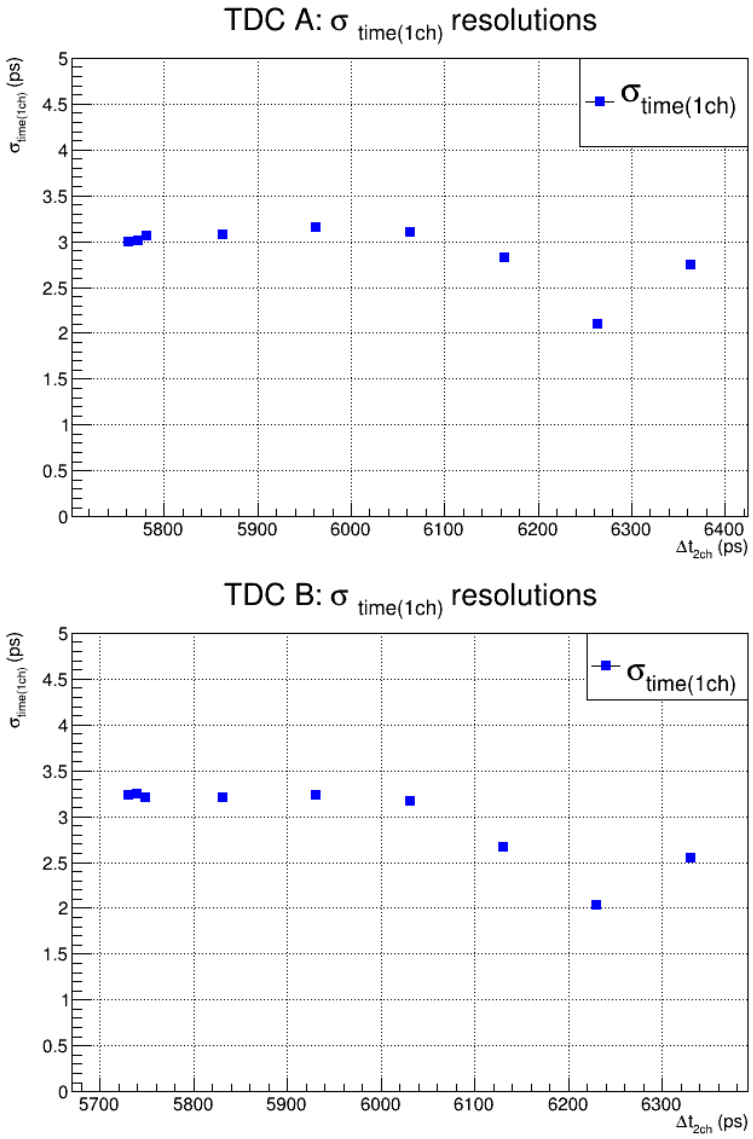


Figure 5.10: Plots for TDC A and TDC B, showing the estimated 1-channel TDC resolution as a function of the measured 2-channel time differences.

Conclusion

In this thesis, I presented an Ethernet-based DAQ system to evaluate a PicoTDC test board as a test environment for the development of the new TDC Readout Module (TRM2), which will be installed on the ALICE-TOF detector to replace damaged TRM cards during LHC Run 3 and Run 4. The INFN electronics laboratory and the ALICE group of Bologna designed the PicoTDC board with a specific hardware layout including two PicoTDCs and a PolarFire FPGA, as the main components selected for the TRM2 card design. The PicoTDCs are 64-channel TDCs with a configurable binning of 12.2 ps or 3.05 ps, and the PolarFire is a flash memory FPGA used to implement the PicoTDC board control and readout.

This project implemented the FPGA firmware architecture and related software, based on the IPbus protocol, to control the configuration and readout of both PicoTDCs. Since the IPbus communicates over Ethernet protocol, the firmware structure relies on a 1 Gb/s Ethernet solution and features specific logic VHDL modules to handle Ethernet frame transfer. Each frame contains specific IPbus packets that identify the final transactions for the on-chip IPbus, set at the bottom of the hierarchical firmware design. The on-chip IPbus implements a master-slave structure with 12 IPbus slaves that handle the board features and the PicoTDCs operations. Finally, C++ software front-end libraries were developed, through the μHAL IPbus back-end library, to control the operation of each IPbus slave and design a general-purpose software interactive interface controlled by the Control Hub software application. Both firmware and software support the UDP/IP layer model, which transfers information nested in the Ethernet frame structure over a wide network of devices.

To test the reliability of the designed DAQ system and to measure the board performances, a PicoTDC resolution measurement was performed employing a programmable delay line with 0.5 ps resolution. Two TDC channels were fed with 2 synchronous 100 KHz signals, shifted by a configurable time delay within the 0-600 ps range. The analysis showed excellent compatibility between the measured time differences and the configured time delays. Finally, a single channel resolution of (2.95 ± 0.43) ps was estimated for both the PicoTDC ASICs considering their finest binning. The developed firmware and software proved reliable during all 36 hours of measurement, providing 18 datasets of 10^7 events each.

In conclusion, the DAQ system ensures stable operation for the board, providing the expected PicoTDC performance and evaluating the system as a good tool for test beams and laboratory analysis with different sensors and front-end electronics. Furthermore, all the tests done with the board confirmed the choice of components and the learning from this thesis is now informing the design of the new TRM2 card.

Bibliography

[1] F. Noferini on behalf of the ALICE Collaboration, *ALICE results from Run-1 and Run-2 and perspectives for Run-3 and Run-4*, J. Phys.: Conf. Ser. 1014 012010 (2018), DOI 10.1088/1742-6596/1014/1/012010

[2] J. Liu on behalf of the ALICE collaboration, *Run 3 performance of new hardware in ALICE*, PoS (2024), DOI: https://doi.org/10.22323/1.450.0052

[3] R. Stock et al., *"Compression effects in relativistic nucleus nucleus collisions"*, Phys. Rev. Lett. 49 (1982), pages 1236–1239.

[4] U. Heinz, M. Jacob,*An Assessment of the Results form the CERN Lead Beam Programme*, arXiv (2000), [arXiv:nucl-th/0002042v1]

[5] P. Paganini, "Quantum Chromodynamics", *Fundamentals of Particle Physics: Understanding the Standard Model*, Cambridge University Press (2023), pages 249-326.

[6] D. H. Perkins, "Quark interactions and QCD", *Introduction to High Energy Physics*, 4th ed., Cambridge University Press (2000), pages 181-191.

[7] W. Nazarewicz, *QCD class*, Lesson at Michigan State University (2015).

[8] M. Fanì, *Proposal of a Continuous Read-Out Implementation in the ALICE - TOF Detector*, Master Thesis, University of Bologna (2015).

[9] M. A. Stephanov, *QCD phase diagram: An Overview*, arXiv (2006), [arXiv: hep-lat/0701002]

[10] S. Sarkar, H. Satz, and B. Sinha, "The Thermodynamics of Quarks and Gluons", *The Physics of the Quark-Gluon Plasma: Introductory Lectures*, ser. Lecture Notes in Physics. Springer Berlin Heidelberg (2009), pages 1-21, ISBN: 9783642022852

[11] H. Sazdjian, *Introduction to chiral symmetry in QCD*, published by EDP Science (2016).

[12] F. Karsch, *Lattice Results on QCD Thermodynamics*, arXiV (2001), [arXiv:hep-ph/0103314v1]

[13] Heng-Tong Ding, *Recent lattice QCD results and phase diagram of strongly interacting matter*, Nuclear Physics A, volume 931 (2014), pages 52-62, available: https://doi.org/10.1016/j.nuclphysa.2014.09.053

[14] ALICE collaboration, *The ALICE experiment, A journey through QCD*, arXiv (2022), pages 1-127, [arXiv:2211.04384]

[15] ALICE Collaboration, *Enhanced production of multi-strange hadrons in high-multiplicity proton-proton collisions*, arXiv (2017), [arXiv:1606.07424v2]

[16] ALICE Collaboration, *ALICE upgrades during the LHC Long Shutdown 2*, arXiv (2023), [arXiv:2302.01238v1]

[17] ALICE Collaboration et al, *The ALICE experiment at the CERN LHC*, Journal of I.: 3 S08002 (2008), DOI: 10.1088/1748-0221/3/08/S08002

[18] M. Slupecki, *Fast Interaction Trigger for ALICE upgrade*, Nuclear Inst. and Meth. A, vol. 1039 (2022), available: https://doi.org/10.1016/j.nima.2022.167021.

[19] N. Pancazio (for the ALICE Collaboration), *PID performance of the ALICE-TOF detector in Run 2*, arXiv (2018), [arXiv:1809.00574]

[20] R. Preghenella, *The Time-Of-Flight detector of ALICE at LHC: construction, test and commissioning with cosmic rays*, Doctoral Thesis, University of Bologna (2009).

[21] A. Alici, *Status and performance of the ALICE MRPC-based Time-Of-Flight detector*, Journal of I.: 7 P10024 (2012), DOI: 10.1088/1748-0221/7/10/P10024

[22] D. Falchieri for the ALICE Collaboration, *DRM2: the readout board for the ALICE TOF upgrade*, PoS (2018), TWEPP-17, 081. 6 p., available: https://cds.cern.ch/record/2312286

[23] D. Falchieri for the ALICE Collaboration, *DRM2: the readout board for the ALICE TOF upgrade*, TWEPP-17 Presentation (2017), available: https://indico.cern.ch/event/608587/contributions/2614132/attachments/1520714/2376966/twepp2017_falchieri.pdf

[24] A. Alici, P. Antonioli, A. Mati, S. Meneghini, M. Pieracci, M. Rizzi, C. Tintori, *Radiation tests of key components of the ALICE TOF TDC Readout Module*, CERN report (2004), available: https://cds.cern.ch/record/814086/files/p184.pdf

[25] J. Christiansen, *HPTDC High Performance Time to Digital Converter*, CERN Geneva (2004), Tech. Rep., available: https://cds.cern.ch/record/1067476

[26] ALICE Collaboration, "Time-of-Flight detector", *ALICE upgrades during the LHC Long Shutdown 2*, CERN Geneva (2023), pages 86-91, [arXiv:2302.01238v1]

[27] CERN EP-ESE-ME, *picoTDC - Picosecond Time to Digital Converter*, CERN Geneva (2024), Tech.Rep., available: https://picotdc.web.cern.ch/

[28] S. Altruda et al., *PicoTDC: a flexible 64 channel TDC with picosecond resolution*, IOP Publishing 18 n. 07 (2023), DOI: 10.1088/1748-0221/18/07/P07012

[29] A. Simon, A. Oliva, "Buck converter", *Power-Switching Converters*, Taylor Francis Group (2010), ProQuest Ebook Central, pages 16-21.

[30] Texas Instruments, *Fundamental Theory of PMOS Low-Dropout Voltage Regulators*, application report (2018).

[31] Microchip Technology Inc., *PolarFire FPGA Product Overview*, product report (2021).

[32] Microchip Technology Inc. and its subsidiares, *PolarFire FPGA Packaging and Pin Descriptions User Guide*, product report (2024).

[33] Texas Instruments, *TCA9416 Ultra-Low-Voltage I2C Translator with Rise Time Accelerators*, product data sheet (2021).

[34] Microchip Technology Inc., "Appendix 1: MAC Layers in the OSI Reference Model and Standard Ethernet Interfaces", *UG0687 User Guide PolarFire FPGA 1G Ethernet Solutions*, pages 17-20, guide revision 5.0.

[35] Cisco systems, *Serial-GMII Specification*, specification revision 1.8 (2005).

[36] AMD, Technical Information Portal, "RGMII Interface Protocols", *GMII to RGMII Product Guide (PG160)* (2022), available: `https://docs.amd.com/r/en-US/pg160-gmii-to-rgmii/RGMII-Interface-Protocols`

[37] Microchip Technology Inc., *VSC8541-02 and VSC8541-05 data sheet Single Port Gigabit Ethernet Copper PHY with GMII/RGMII/MII/RMII Interfaces*, product data sheet revision 4.2.

[38] C. Ghabrous Larrea, K. Harder, D. Newbold, D. Sankey, A. Rose, A. Thea and T. Williams, *IPbus: a flexible Ethernet-based control system for xTCA hardware* (2015), JINST 10 no.02, C02019., `DOI: 10.1088/1748-0221/10/02/C02019`

[39] OpenCores, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, specification revision B.3 (2002).

[40] CERN, IPbus user guide, *On-chip bus*, available: `https://ipbus.web.cern.ch/doc/user/html/firmware/bus.html`

[41] Microchip Technology Inc., Developer Help, *Introduction to TCP/IP (Part 2) - Five Layer Model and Applications* (2023), available: `https://developerhelp.microchip.com/xwiki/bin/view/applications/tcp-ip/five-layer-model-and-apps/`

[42] P. Brooks, *Ethernet/IP - Industrial Protocol*, Reasearch Gate (2001), vol.2, pages 505 - 514, DOI: `DOI:10.1109/ETFA.2001.997725`

[43] Microchip Technology Inc., "Appendix 2: Ethernet Frame Format", *UG0687 User Guide PolarFire FPGA 1G Ethernet Solutions*, pages 21-23, Guide revision 5.0.

[44] J. Postel, *RFC 791 - Internet Protocol*, Internet Engineering Task Force (1981), available: `https://tools.ietf.org/html/rfc791`

[45] R. Frazier et al., *The IPbus Protocol, an IP-based control protocol for ATCA/μTCA*, IPbus version 2.0. (2013).

[46] CERN , GitHub, *ipbus/ipbus-firmware*, available: `https://github.com/ipbus/ipbus-firmware`

[47] D. Porret, Gitlab, "firmware", *picoTDC/Picotdc Demo*, available: https://gitlab.cern.ch:8443/picoTDC/picotdc-demo/-/tree/master/firmware?ref_type=heads

[48] S. Geminiani, D. Falchieri, M. Giacalone, J. Succi, Baltig, "Firmware_PicoTDCboard", *geminian/PicoTDCboard*, available: https://baltig.infn.it/geminian/picotdcboard/-/tree/main/Firmware_PicoTDCBoard?ref_type=heads

[49] Microchip Technology Inc., *PolarFire Family Clocking Resources*, product guide (2023).

[50] Microchip Technology Inc., "Implementing 1G Ethernet Solutions", *UG0687 User Guide PolarFire FPGA 1G Ethernet Solutions*, pages 9-12, guide revision 5.0.

[51] Microchip Technology Inc., "RGMII to GMII Converter", *Microchip Technology*, available: https://onlinedocs.microchip.com/pr/GUID-53092BEF-DCEB-4741-9EFC-5843AA55C657-en-US-3/index.html?GUID-70978CEE-7307-4E21-AE4F-8A64B5E894C3

[52] Microchip Technology Inc., *CoreTSE v3.2*, product handbook (2022).

[53] Arm Ltd., *AMBA APB Protocol Specification*, protocol specification (2023), available: https://developer.arm.com/documentation/ihi0024/latest/

[54] Microchip Technology Inc., *CoreABC v4.1*, product handbook (2022).

[55] Microchip Technology Inc., *CoreAPB3 v3.8*, product handbook (2022).

[56] Microchip Technology Inc., *CoreFIFO v3.0*, product handbook (2021).

[57] D. Porret, Gitlab, "software", *picoTDC/Picotdc Demo*, available: https://gitlab.cern.ch/picoTDC/picotdc-demo/-/tree/master/software?ref_type=heads

[58] S. Geminiani, P. Antonioli, M. Giacalone, J. Succi, Baltig, "Software_PicoTDCboard", *geminian/PicoTDCboard*, available: https://baltig.infn.it/geminian/picotdcboard/-/tree/main/Software_PicoTDCboard?ref_type=heads

[59] R. Frazier et al., *Software and firmware for controlling CMS trigger and readout hardware via gigabit Ethernet*, Physics Procedia. 37 (2012), pages: 1892-1899, DOI: 10.1016/j.phpro.2012.02.516.

[60] CERN, IPbus user guide, *Control Hub*, available: https://ipbus.web.cern.ch/doc/user/html/software/ControlHub.html

[61] CERN, IPbus user guide, *IPbus software*, available: https://ipbus.web.cern.ch/doc/user/html/software/index.html

[62] G. Zanasi, *Preliminary characterization measurements of CERN picoTDC*, Bachelor Thesis, University of Bologna (2023).

[63] Skyworks Solutions Inc., *Si5341-D Evaluation Board User's Guide*, Product user guide (2021), available: https://www.skyworksinc.com/-/media/Skyworks/SL/documents/public/user-guides/Si5341-D-EVB.pdf

[64] Skyworks Solutions Inc., *Si5341/40 Rev D Data Sheet*, Product data sheet (2021), available: https://www.skyworksinc.com/en/products/timing/ultra-low-jitter-clock-generators/si5341d

[65] Colby Instruments, *XT-200-Operating and Programming Manual Version 1.0*, Product manual, available: https://www.colbyinstruments.com/xt-200

Ringraziamenti

A conclusione di questo elaborato desidero menzionare tutte le persone che hanno contribuito, in diverso modo, al raggiungimento di questo traguardo.

Innanzitutto vorrei ringraziare il Dottor Davide Falchieri, relatore di questa tesi, per avermi seguito durante tutto il mio percorso di tirocinio, presso il laboratorio di elettronica dell'INFN di Bologna, aiutandomi nello sviluppo del progetto firmware. Allo stesso modo vorrei ringraziare anche il Dottor Pietro Antonioli, correlatore di questa tesi, per avermi indirizzato e seguito nello sviluppo del relativo software per il controllo e il readout della PicoTDC board. Vorrei inoltre ringraziare entrambi per il lavoro svolto durante tutto il mio percorso di tesi, nel quale hanno saputo guidarmi con grande pazienza e disponibilità rispondendo ad ogni mio quesito e dubbio. Infine, vorrei ulteriormente ringraziarli per avermi introdotto nel mondo della ricerca scientifica, dove ho potuto apprezzare con mano ciò che ho studiato in questi due anni di laurea magistrale confermando la mia volontà di proseguire in questo percorso.

Vorrei ringraziare di cuore tutti i dipendenti del laboratorio di elettronica dell'INFN sezione di Bologna, per l'accoglienza riservatami durante tutto il periodo di tirocinio. Ringrazio ognuno di loro per i preziosi consigli e l'aiuto datomi per ultimare il progetto di tesi. In particolare, vorrei esprimere a tutti la mia gratitudine per la pazienza dimostrata nel rispondere a tutte le mie curiosità riguardanti l'elettronica di schede custom. Infine, vorrei ringraziare Casimiro Baldanza per avermi illustrato il design della PicoTDC board, da lui disegnata insieme al Dottor Davide Falchieri.

Vorrei ringraziare il Dottor Marco Giacalone per l'aiuto e il sostegno durante lo sviluppo del software di readout e le misure di risoluzione temporale. Vorrei ulteriormente esprimere la mia gratitudine a Lui e ai Dottori Luigi Pio Rignanese, Nicola Rubini e Bianca Sabiu per il supporto e l'aiuto dedicatomi, durante il periodo di tesi e le mie esperienze al CERN di Ginevra.

Vorrei esprimere una menzione particolare per Jacopo Succi, mio collega e compagno di tesi, che ha condiviso con me ogni attimo di questo percorso universitario. Lo ringrazio per i consigli e tutte le "discussioni" produttive, utili al fine di aiutare entrambi a proseguire nei nostri rispettivi progetti.
Vorrei inoltre ringraziare Giovanni Mastropasqua, tecnico dell'INFN, per l'aiuto e il sostegno durante il periodo di tesi, tra un caffé e l'altro.
Inoltre, vorrei ringraziare tutte le persone e i colleghi che ho incontrato in questi cinque anni di università, tra triennale e magistrale. Soprattutto vorrei menzionare la compagnia di "Baracca e Burattini", sparsa ormai in varie parti di Italia e del mondo. Tra questi

vorrei fare una menzione speciale per Matilda Panza, che ancora oggi, insieme a Jacopo e Giacomo Casali, è sempre pronta a sopportarmi e supportarmi.

Vorrei ringraziare i miei amici di sempre che hanno saputo accettarmi e aiutarmi durante il mio percorso universitario, nonostante i miei difetti. In particolare, vorrei menzionare i ragazzi e le ragazze del 98' che mi ricordano ogni giorno come sia bello avere al proprio fianco persone con caratteri estremamente diversi ma che si vogliono bene da sempre, a prescindere da tutto.

Vorrei ringraziare dal profondo del mio cuore quelle che ad oggi considero le mie tre famiglie.

In particolare, vorrei ringraziare Mirco e Claudia per avermi accolto sin da bambino in casa loro, trattandomi come un secondo figlio e avendo sempre un occhio di riguardo nei miei confronti.
Ringrazio Viola per la gentilezza e per il supporto, durante alcuni momenti davvero difficili del mio percorso universitario.
Infine, vorrei fare un ringraziamento speciale a Leonardo mio migliore amico e fratello. Lo ringrazio per essere ogni giorno la persona giusta al momento giusto per le parole, i modi di fare e la continua comprensione. Lo ringrazio per avermi mostrato come un vero amico sia capace di soffrire delle tue sconfitte e gioire dei tuoi traguardi, sempre insieme a te e spronandoti ogni giorno a fare del tuo meglio.

Vorrei ringraziare Sabrina e Giovanni per avermi accolto nella loro casa con gentilezza e ospitalità.
Vorrei ringraziare Tamara per condividere con me ogni giorno la vita, attraversando le tante difficoltà. La ringrazio per la sensibilità e l'amore che mi dimostra fin dal primo giorno, trovando sempre un minuto per stare insieme e porre rimedio ad una brutta giornata.

Ultima ma non meno importante, vorrei dedicare questa tesi alla famiglia che per me è casa.

Vorrei ringraziare i miei zii Lella, Mauro, Fiffi e Loris, per essere dal primo giorno i miei più grandi sostenitori. Li ringrazio per tutti quei piccoli gesti, come un abbraccio dopo qualche delusione o un semplice messaggio, che per me sono stati salvezza e dimostrazione di grande amore.
Vorrei ringraziare mio cugino Enrico per tutti i momenti passati in questi anni, come due fratelli. Lo ringrazio per la sensibilità con cui riesce a capire i miei problemi, tra una risata e l'altra.
Vorrei ringraziare mia mamma e mio babbo, i due pilastri che hanno reso possibile questo percorso. Li ringrazio dal profondo del cuore per credere nelle mie capacità ogni giorno, anche quando io non ne ho la forza. Spero che queste mio lavoro e le mie parole li rendano fieri della persona che sono diventato oggi, grazie ai loro insegnamenti e al loro amore.
Infine, vorrei ringraziare i miei due nonni Bruna e Gino che considero i miei personali eroi. Li ringrazio per avermi accudito da sempre, cercando ogni giorno di regalarmi tranquillità e spensieratezza in questo mondo così confusionario.