Alma Mater Studiorum · University of Bologna

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

Development of a PicoTDC based board: USB interface and test beams measurements

Supervisor:
Dr. Davide Falchieri

Co-supervisor:
Dr. Pietro Antonioli

Submitted by:
Jacopo Succi

Academic Year 2023/2024

Abstract

The TOF (Time-of-Flight) readout system of the ALICE experiment at the LHC is based on a TDC Readout Module (TRM) hosting a TDC produced by CERN in the early 2000s. This thesis is part of the TRM2 project, producing a new card based on the newly produced PicoTDC.

This thesis describes the firmware and software development for readout, control and configuration of a test board hosting a PolarFire FPGA and two PicoTDC ASICs and the configuration of two LIROC ASICs, integrated in two mezzanine cards plugged on the board, through an USB Super-Speed interface.

Both firmware and software were built considering the IPBus protocol already developed by CERN and adapted to the new USB interface. The IPBus implements a master-slave structure in which each slave is identified by a 32-bit address and communicates with the master through a 32-bit data bus.

The DAQ chain was then tested at CERN in June 2024 in a test beam to test its stability and the performance. During the operation at the test beam, the system proved to be fast and reliable over the long run. A set of six data acquisition runs were then used to get a preliminary result on the time resolution of an LGAD-LIROC-PicoTDC DAQ chain obtaining $\sigma = 42.83 \pm 0.18$ ps.

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 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, which 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 project is based on a newly developed TDC by CERN: the PicoTDC. As an intermediate step a PicoTDC Board was designed by the INFN electronics laboratory together with the ALICE-TOF collaboration of Bologna.

This thesis objective was the development of a new USB Super-Speed interface for the PicoTDC board to allow fast communication with the integrated PolarFire FPGA and data readout of the two PicoTDC ASICs hosted on board. My work consisted in the development of the firmware architecture for the USB interface through an FTDI FT601 chip, the development of the back-end libraries and software tools to adapt an already developed software suite for the configuration and readout of the two integrated ASICs. Furthermore, some modifications of the already existent FPGA SoC were made for the control of two LIROC ASICs that acted as front-end electronics for the sensors connected to the board. The following chapters describe the work done during the development of this new interface for the PicoTDC board. Specifically the first chapter provides an overview of the board features with a special focus on the PicoTDC ASICs characteristics, the different interface integrated on the board and an introduction on the IPBus SoC and its protocol. The second and third chapters focus on the firmware and software development for the new USB interface and the LIROC ASICs control. Finally, the last chapter provides a brief description of the test made during a test beam in June 2024 at CERN to evaluate the entire DAQ chain performance and the timing resolution achieved by the LGAD sensors tested using the PicoTDC Board and the LIROC front-end ASIC.

Chapter 1

The PicoTDC board, IPbus protocol and communication interfaces

The PicoTDC board is an evaluation board developed by the ALICE-TOF collaboration to test new interfaces and communication protocols for the control and readout of a PicoTDC ASIC which is the main element of the new TRM2. The board is equipped with a Microchip PolarFire FPGA to implement the logic to control two PicoTDC ASICs and multiple interfaces that can be used to communicate directly with the on-board FPGA.

In this chapter will be provided a detailed description of the PicoTDC ASIC together with the board design made by the electronics workshop of INFN Bologna and the main features of the multiple communication interfaces. Successively the IPBus System-on-Chip and its protocol will be described since it represents the backbone of the board system.

1.1 PicoTDC overview

The PicoTDC is an ASIC designed by CERN which provides 64-channels for high-resolution time measurements. Similar devices are now used, particularly in HEP experiments, to build PID systems, TOF detectors and tracking systems undergoing high particle rates[1].
Since it was designed to support multipurpose applications and R&D activities, it does not include any analog front-end nor discrimination circuits, given that they need to be optimized for each specific sensor type used. Furthermore, this ASIC features a 65 nm CMOS technology and an architecture similar to the HPTDC.

As a new generation chip, the PicoTDC provides a better resolution and higher channel multiplicity with respect to the older HPTDC that was developed in the early 2000s for LHC applications andhas been widely used in HEP experiments.

1.1.1 PicoTDC architecture

The PicoTDC ASIC can manage both leading and trailing edges of an input digital signal, providing the arrival time or the direct measure of the Time over Threshold (ToT) of the signal.

The architecture is divided in two distinct sections: the Timing Unit and the Data Processing Unit. The former takes the differential input signals coming from the sensors front-end circuit of each channel and perform the decoding of the hit time value, while the latter collects and stores them into a built-in FIFO memory, waiting to be read[2].

In order to reach the best resolution performance, the Timing Unit relies on a precise time reference given by an external differential 40 MHz clock which is fed to an on-chip PLL; such PLL performs the clock multiplication process generating a range of frequencies from 40 MHz up to 1.28 GHz. All clocks used by the TDC come from this PLL in order to reach synchronisation between the various clock domains and reducing the jitter[2].

As shown in Figure 1.1, the 64 differential input channels are divided in four groups with the same scheme which ends in the related readout FIFO and can be read by the DAQ electronics via four 8-bit differential parallel ports or via a single 8-bit differential port common to all four groups.

The time base for the TDC measurements is provided via a Delay Locked Loop (DLL) with 64 delay elements resulting in a 12.2 ps bin size, the PLL feedback divider and a synchronous clock counter. The 64 delay taps can be further divided into 256 time taps by an interpolator, reaching a 3.05 ps bin size.

The hit measurements is performed by sampling the input signal for each time tap every clock cycle and detecting a leading or trailing edge of the signal. The fine time value is then stored if a hit has been detected; this enables the architecture of the PicoTDC to detect one edge per channel per 1.28 GHz clock cycle. Because of

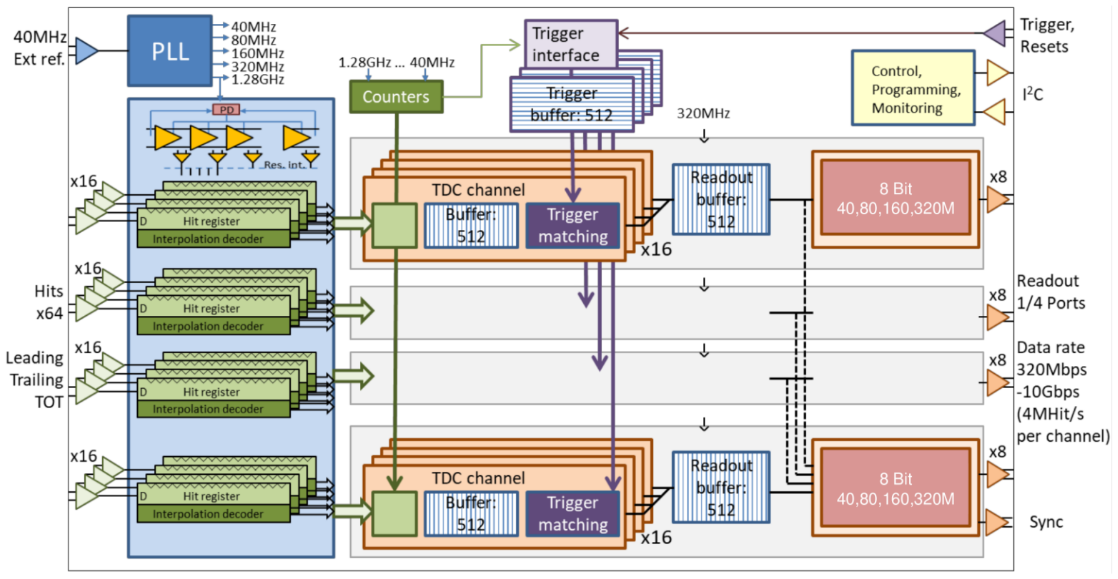


Figure 1.1: *PicoTDC internal architecture scheme with the Timing unit on the left (blue square) and the Data Processing unit on the right.*

this, a minimum separation of 781 ps is required between two edges or the resulting data might be corrupted[2].

Each channel can store up to 4 measurements in a local derandomizer until they are written into a 512-words deep memory buffer at a frequency of 320 MHz. The measurements can then be transferred into the 512-words deep FIFO shared by each group directly or by performing a trigger matching function to select only the events related to an external trigger signal. In the latter, the trigger time tag, the event ID and the bunchcounter ID can be stored temporarily into a 512-words deep trigger FIFO for each channel group and a window of programmable size is available to the trigger matching function to manage the spread of the hits related to the same event.

The PicoTDC is also provided with an I2C interface to access its internal registers with a 16-bit addressing. This interface can be used by an external I2C master to configure the functional parameters and control the TDC status.

1.1.2 Phase Locked Loop (PLL)

As introduced in section 1.1 the component responsible for the clock multiplication of the various parts of the architecture is the PLL. The block diagram of the circuit is shown in Figure 1.2 and is designed to compare the frequencies provided by the clk_in input and the adjustable feedback provided by the divider. When a match in phase and frequency is achieved, a steady stage has been reached and the PLL is locked onto the desired frequency and phase with respect to the reference input.

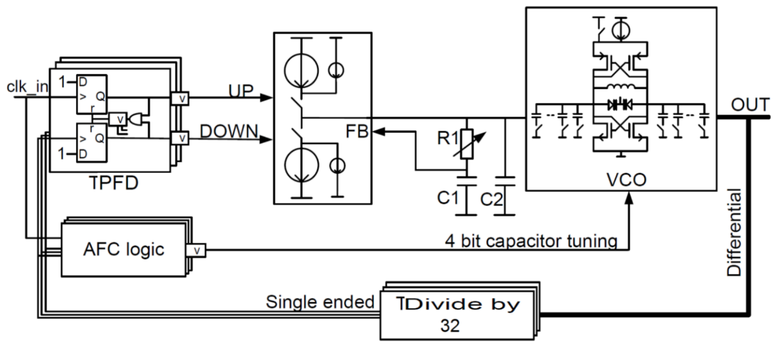


Figure 1.2: *Block diagram of the PicoTDC PLL.*

The Voltage Controlled Oscillator (VCO) generates a symmetrical clock signal, which is divided by the divider and then compared with the reference clock. If a discrepancy in phase or frequency is detected by the Phase-Frequency Detector (PFD), the control voltage of the VCO is adjusted via a charge pump and a filter circuit. Furthermore, the PLL is provided with an AFC[1] logic block in order to perform the calibration of the VCO switchable capacitor to ensure good performances of the PLL when sudden changes in temperature, voltage and/or power happen[1].

In particular, the PLL of the PicoTDC is designed using the Triple Modular Redundancy technique (TMR). This technique is often used for ASICs designed to work in critical conditions, such as in a high radiation environment, in order to reach high reliability of the electronic circuits[2].

[1]Automatic Frequency Calibration

1.1.3 Delay Locked Loop (DLL) and interpolation lines

The Delay Locked Loop is the first step of the hit decoding process done by the TDC's Timing Unit. The base functioning of a DLL is determined by a Phase Detector followed by a charge pump that controls the voltage of the delay gates of the loop. Specifically, this circuit constrains the total delay to be equal to one period of the external reference clock and allows the prevention of metastability of the delay line that can be caused by voltage or temperature variation.

In the case of the PicoTDC, shown in Figure 1.3, the DLL acting as the first step of the decoding works sinchronously with the 1.28GHz provided by the PLL and in its first stage reaches a resolution of 12.2 ps exploiting the 64-taps delay line.

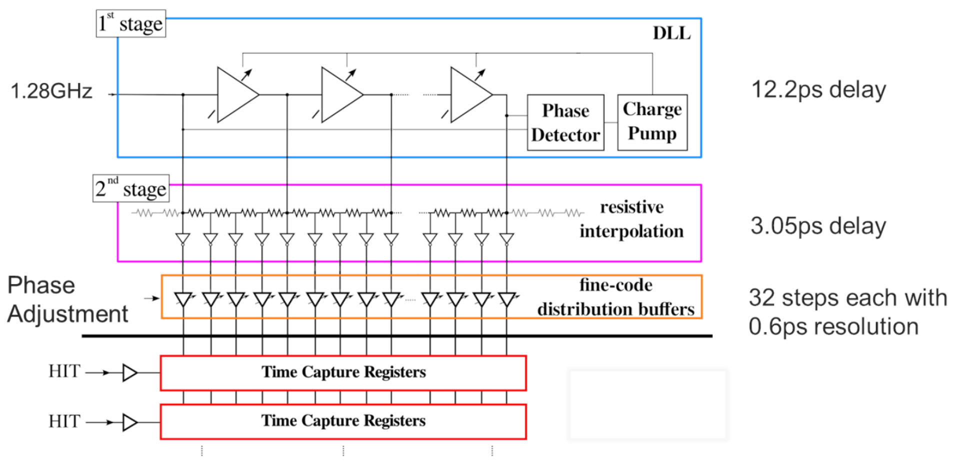


Figure 1.3: *Scheme of the DLL and fine interpolation stage for the PicoTDC hit decoding.*

Each delay line of the PicoTDC's DLL then feeds a second interpolation stage that arranges four resistive time taps to the next delay step. The total number of delay taps then becomes 256 with a resolution of 3.05 ps. In the final stage, each resulting delay line is fed to a built-in phase adjustment feature with a resolution of 0.6 ps to correct possible mismatches[1]. This adjustment is done directly on the resulting output of the two stages and used in the fine resolution mode (i.e. when the 3.05 ps binning is used), this requires an individual calibration for every TDC.

By dividing the channels in two, a 2570-bit register of the TDC can determine all the adjustments tap values for each half independently.

To get the DLL and the hit decoding stage to work properly, it must be initialized after the PLL has reached the steady state and it's locked onto the correct frequency and phase, which takes about ≈ 10 ms from the ASIC.

1.1.4 Data Processing

As mentioned in the previous subsections, the Data Processing Unit of the PicoTDC is built to implement data buffering and trigger matching for each channel group and works synchronously with the 320 MHz clock provided by the PLL. In particular, each channel buffer takes out the oldest hit from the related derandomizer and repeats the process until the 512-words deep FIFO fills up.

At this stage, depending on the TDC configuration, the data can be extracted in the following ways[2]:

- **Single Measurement Mode**: In this configuration, the TDC performs time measurements on one or both edges of the hit signal. The bit width of the time measurement is determined by the working mode of the TDC, so it will be 24 bits for the coarse mode and 26 bits for the fine mode. To convert the TDC data to the time value in seconds it's sufficient to multiply the 26 bit value by the smallest bin size (3.05 ps) and due to the bits alignment this works for both coarse and fine mode. Each detected edge is stored in the channel buffer using a 32-bits word in which are encoded the informations as follows:

  – word type (bit 31)

  – channel number (bits [30:27])

  – edge type (bit 26)

  – time measurement (bits [25:0])

- **Paired Measurement Mode**: In this configuration, also called TOT mode, the TDC performs TOT measurements. In this case, the leading and trailing edge informations are combined into a single 32-bit word. To convert the leading edge and TOT values into seconds, they must be multiplied each by their respective bin size, whose values are defined by the values of the tot_startbit and tot_leadingstartbit registers. The data word format is the following:

  – word type (bit 31)

  – channel number (bits [30:27])

  – leading edge time (bits [26:11] or [26:8])

  – pulse width (bits [10:0] or [7:0]).

1.1.5 Trigger interface

The trigger matching function, which is implemented for each group, reads the time measurements at random access from the related channel buffers. In particular, as shown in Figure 1.4, for each input trigger signal a time tag is associated and its value is determined by the difference between the arrival time and the configured latency width value. The trigger time tag is decoded using the 40 MHz coarse counter and can be subtracted from the hits time value to get only the time interval from an event of interest.

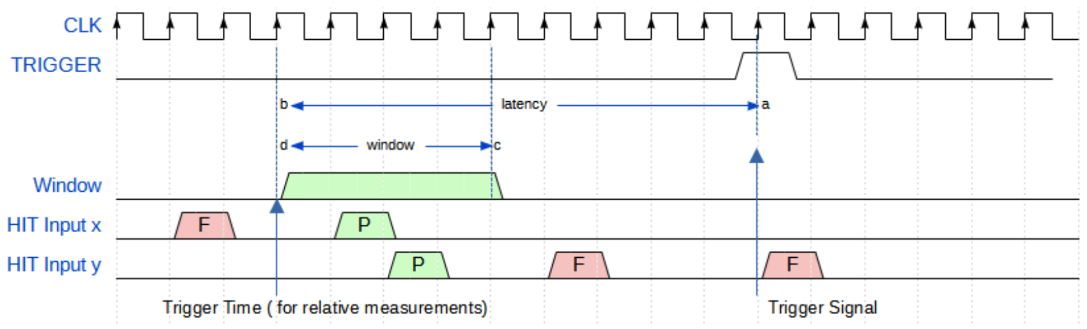


Figure 1.4: *Scheme of the trigger matching process. The function considers "Processed" (P) only the hits inside the matching window, otherwise they are tagged as "Failed" (F) and ignored.*

The buffered data is then matched to the relative trigger time tag using a configurable time window. Hits that fall inside this window are recognized as belonging to that specific trigger and written to the next FIFO together with one or two header words and a trailer word. The information encoded in these two 32-bit words can be modified by the internal TDC registers and can contain informations like the Event ID, the Bunch ID, the trigger time and some status bits showing the memory buffers status for the header word, and, for the trailer, the Event ID and the number of valid hits received by that specific group.

The length of the latency and matching window are defined as a finite number of 40 MHz clock cycles since the trigger time tag is decoded by the 40 MHz coarse counter. The maximum length of the latency is by construction defined as half of the maximum coarse count which is $2^{12} \times 25$ ns =102 µs; however, it is recommended to set this value to half of the maximum, so, in terms of time, the maximum recommended trigger latency equals to 51.2 µs[2].

In order for the matching function to work properly, the length of the trigger latency must be larger than the length of the matching window to ensure that all hits corresponding to a specific trigger signal are already present in the channel buffers. Hits are removed from the channels buffers only if they are older than the latest processed trigger (not within the matching window) or if they have been found rejectable by a special function such that, when no trigger is found in the trigger FIFO, hits that are older than the latency plus one clock cycle are automatically rejected to prevent buffer overflows and to speed up the search time. If the trigger FIFO is found full, then the successive triggers time tags are discarded, but the Event ID counter is kept running to maintain synchronisation.

The next data read-out phase is managed by each group using a round-robin read-out to ensure a fair bandwidth share between all the group channels. In this way the output data are organized in a 512-words deep FIFO ready to be read using an 8-bit parallel differential port running at frequencies up to 320 MHz. If a read-out FIFO runs full, then there are two main options on how to handle this situation[2]:

- **Back propagation**: In this case the trigger matching function will be suspended until new space is available. This will bring to the channel buffers to potentially store more data and if this situation is maintained for extended periods will bring the channel buffers to overflow. The total buffering capability of the TDC in this case amounts to 8704 measurements for each channel group.
- **Rejection**: In this configuration as the read-out FIFO gets full, then all event data will be rejected with the exception of header and trailer words. If this happens for an extended period of time then the bandwidth might not be shared fairly between the channels, but in some cases it is advantageous to reject hits when large amount of data starts to accumulate in the read-out buffers.

As mentioned before, data can then be extracted from the readout ports following the process shown in Figure 1.5. In order to synchronize to the data an additional differential sync signal is provided and can be configured to mark the first byte of a data word or to provide a clock signal with half the data rate. The first bit of the frame (bit 31 of the data word) determines if the frame is of data type (0) or management type (1), so the type of the frame can be determined after the reception of the first byte.

When no data is available in the read-out FIFO then the read-out port value is set to the constant value of 0xD0, which is a unique identifier for a management type frame, creating an idle frame of value 0xD0D0D0D0. Other possible management frame identifiers are: channel group separator (0xF), first header (0x8), second header (0x9) and trailer (0xA)[2].

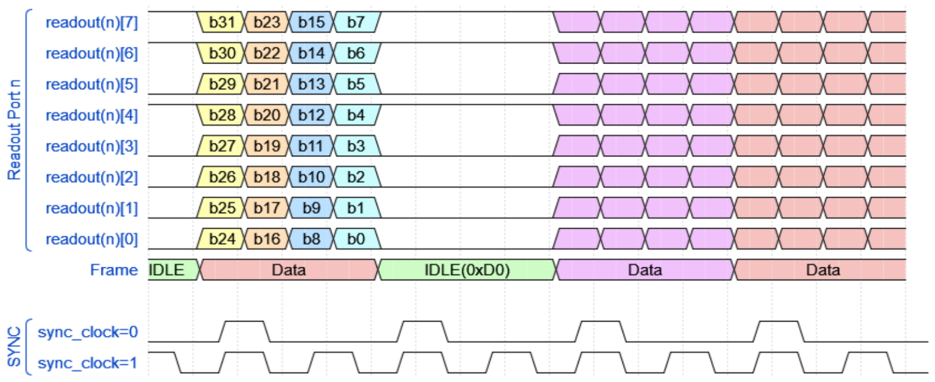


Figure 1.5: *PicoTDC data readout process via an 8-bit differential parallel port. When no data is available an idle frame (0xD0) is continuosly sent through the port.*

1.2 PicoTDC board features

The PicoTDC board, shown in Figure 1.6 is an evaluation board designed by the ALICE TOF collaboration at the Bologna section of INFN to test and evaluate the performance of the PicoTDC designed by CERN as a possible upgrade for the TDC module used in the ALICE-TOF. Furthermore, the board was also designed to test various communication interfaces, with a particular interest on high-speed, high bandwidth, communication protocols.

The board design can be divided in three main sections: the power supply, which takes the input voltage and distribute the power needed by the various components, the I/O section, which is equipped with various types of I/O interfaces, the core section which includes the two PicoTDC ASICs and the Polarfire FPGA that manages all the communications between the various components.

The 2 ASICs are connected to two FMC connectors using the sub-LVDS standard, where other boards of front-end electronics and sensors can be plugged. For the I/O section, the board has been equipped with different I/O solutions such as an Ethernet link managed by a PHY chip or by exploiting an optical link, two USB interfaces in the form of a USB 3.0 Micro-B SuperSpeed FTDI FT601Q bridge chip and a USB 2.0 Type-B Cypress EZ-USB FX3 mezzanine; furthermore a JTAG connector and other auxiliary I/O are provided for the FPGA programming and reset. A block diagram of the whole system is shown in Figure 1.7

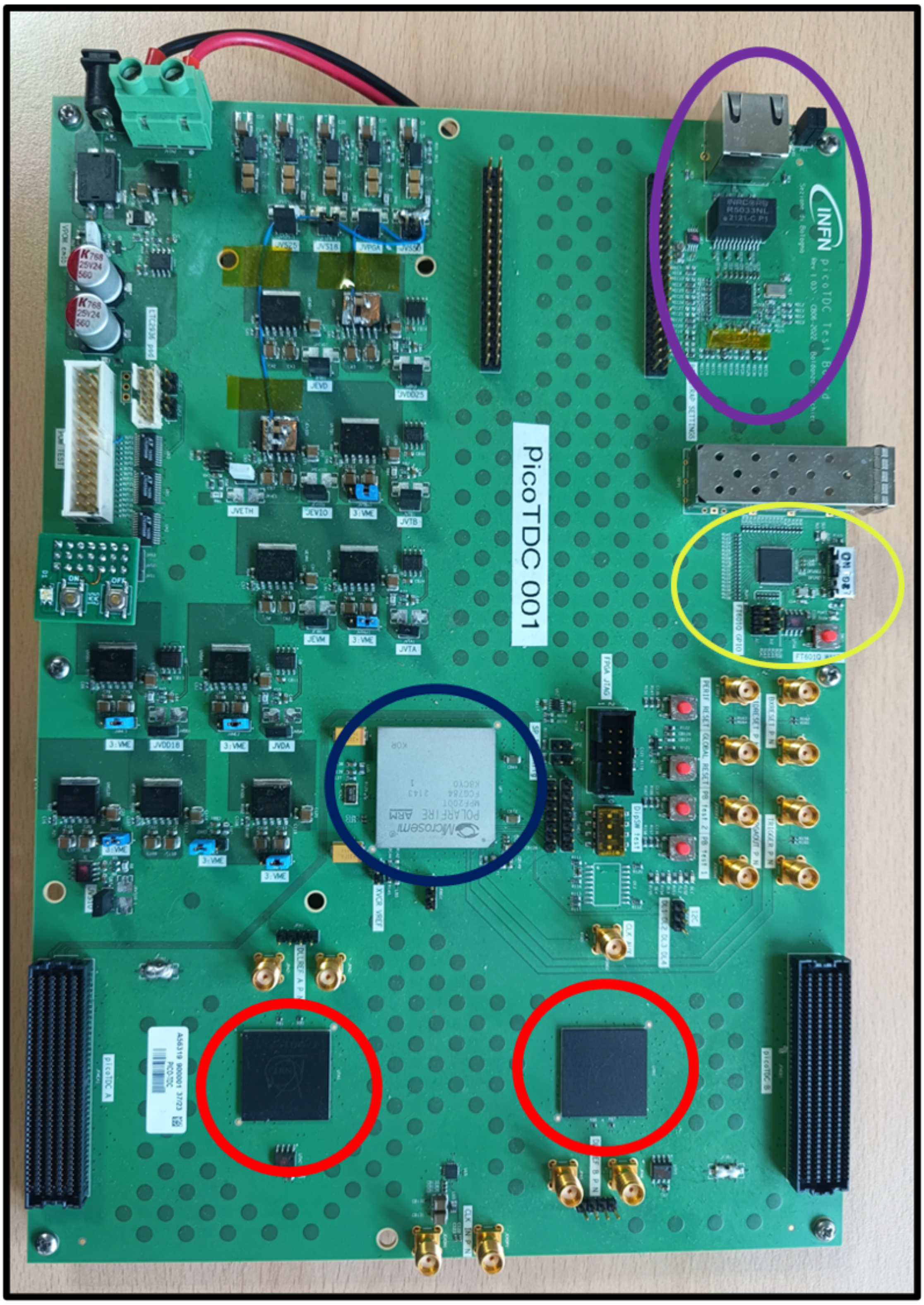


Figure 1.6: *Photo of the PicoTDC Board in which the PolarFire FPGA (blue), the two PicoTDCs (red), the Ethernet connector subsystem (purple) and the USB FTDI subsystem (yellow) are highlighted.*

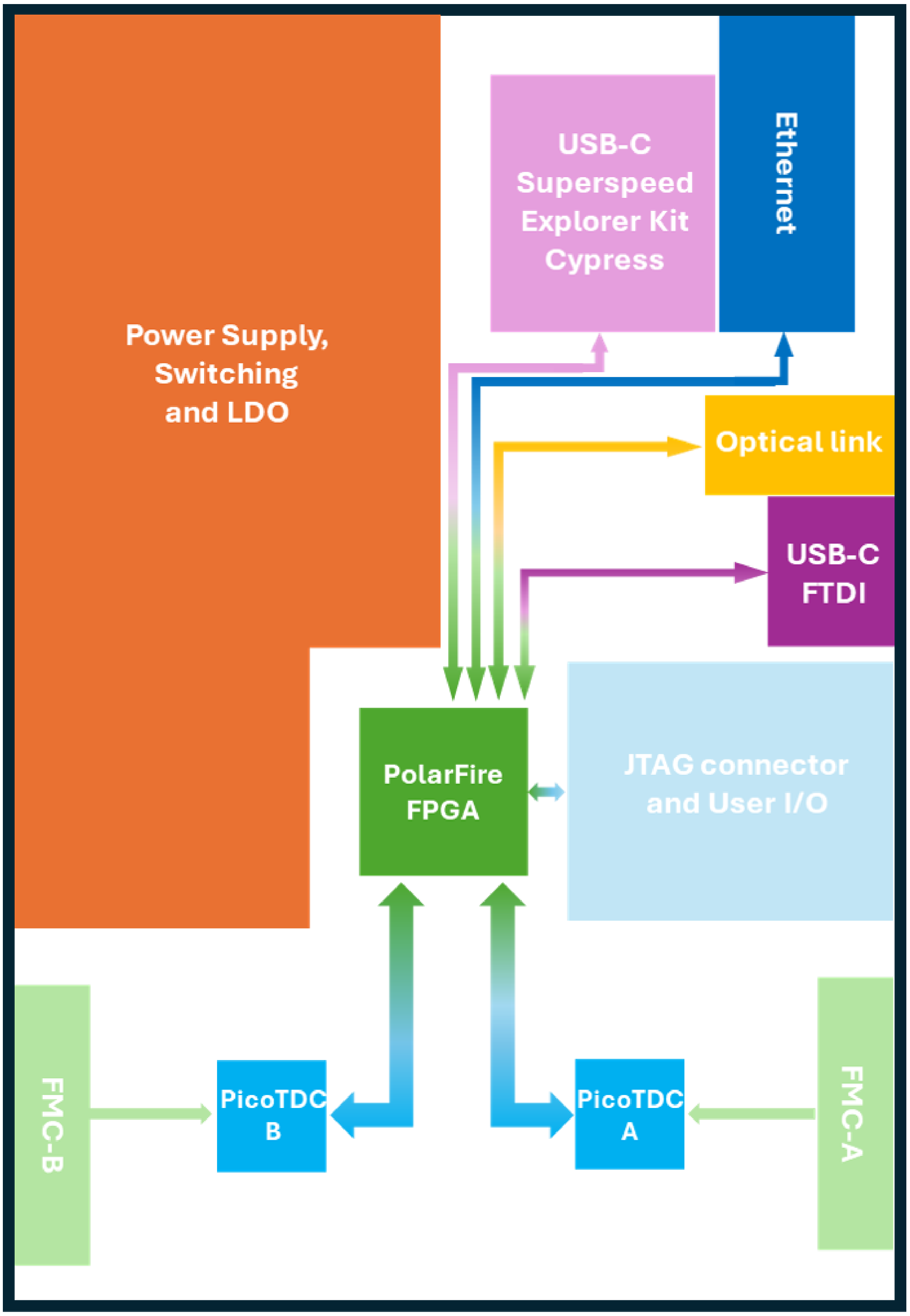

Power Supply, Switching and LDO
USB-C Superspeed Explorer Kit Cypress
Ethernet
Optical link
USB-C FTDI
PolarFire FPGA
JTAG connector and User I/O
FMC-B
PicoTDC B
PicoTDC A
FMC-A

Figure 1.7: *Block diagram of the PicoTDC board. The scheme shows its main components and their connections.*

1.2.1 Power distribution

In order to work properly, the board needs to be supplied with a voltage ranging between 9 V and 14 V and boots automatically when powered on. As a control interface, three voltage monitors are programmed to check the various voltage level and current drain and can cut the power to the whole board whenever a high current is detected.

After the power is provided, the current flows to switching converters generating the correct voltage levels needed by the five independent power rails. The power rails can then cascade into some LDOs that generate other voltage levels[3], needed by the various electronics device mounted, and to reduce the noise caused by the switching converters.

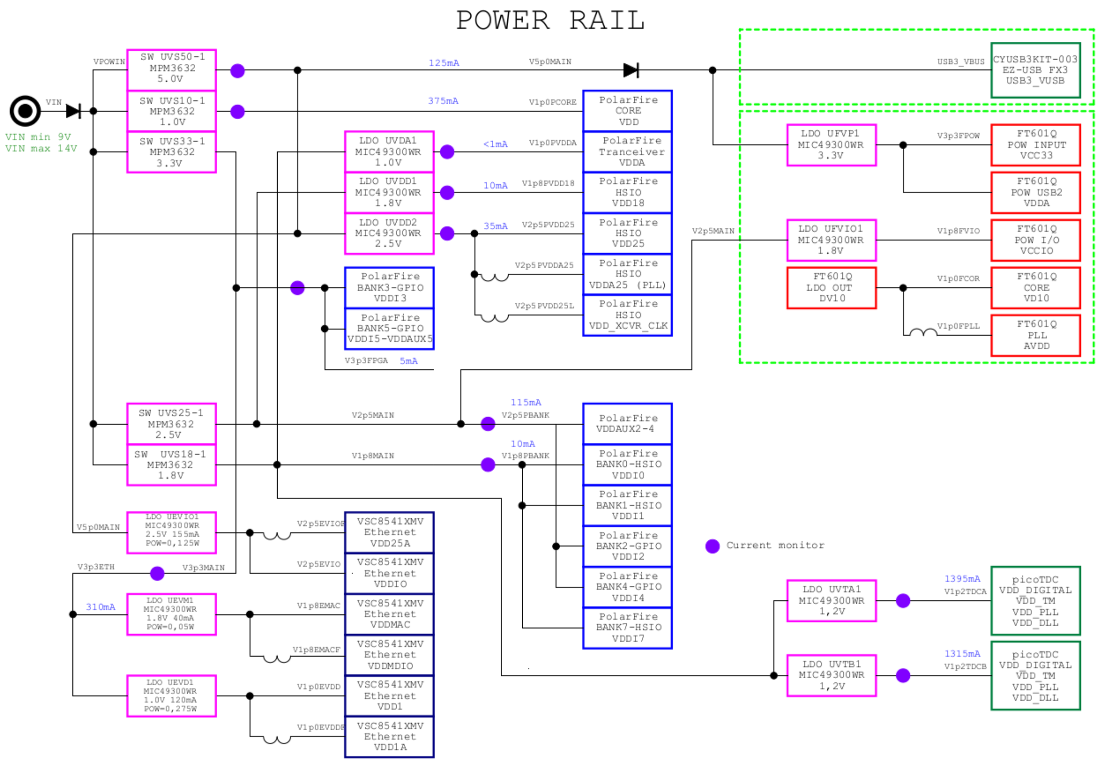

POWER RAIL
VIN
VIN min 9V
VIN max 14V
SW UVS50-1 MPM3632 5.0V
SW UVS10-1 MPM3632 1.0V
SW UVS33-1 MPM3632 3.3V
LDO UVDA1 MIC49300WR 1.0V
LDO UVDD1 MIC49300WR 1.8V
LDO UVDD2 MIC49300WR 2.5V
PolarFire CORE VDD
PolarFire Tranceiver VDDA
PolarFire HSIO VDD18
PolarFire HSIO VDD25
PolarFire HSIO VDDA25 (PLL)
PolarFire HSIO VDD_XCVR_CLK
PolarFire BANK3-GPIO VDDI3
PolarFire BANK5-GPIO VDDI5-VDDAUX5
CYUSB3KIT-003 EZ-USB FX3 USB3_VUSB
LDO UFVP1 MIC49300WR 3.3V
LDO UFVIO1 MIC49300WR 1.8V
FT601Q LDO OUT DV10
FT601Q POW INPUT VCC33
FT601Q POW USB2 VDDA
FT601Q POW I/O VCCIO
FT601Q CORE VD10
FT601Q PLL AVDD
SW UVS25-1 MPM3632 2.5V
SW UVS18-1 MPM3632 1.8V
PolarFire VDDAUX2-4
PolarFire BANK0-HSIO VDDI0
PolarFire BANK1-HSIO VDDI1
PolarFire BANK2-GPIO VDDI2
PolarFire BANK4-GPIO VDDI4
PolarFire BANK7-HSIO VDDI7
LDO UEVIO1 MIC49300WR 2.5V 155mA POW=0,125W
LDO UEVM1 MIC49300WR 1.8V 40mA POW=0,05W
LDO UEVD1 MIC49300WR 1.0V 120mA POW=0,275W
VSC8541XMV Ethernet VDD25A
VSC8541XMV Ethernet VDDIO
VSC8541XMV Ethernet VDDMAC
VSC8541XMV Ethernet VDDMDIO
VSC8541XMV Ethernet VDD1
VSC8541XMV Ethernet VDD1A
Current monitor
LDO UVTA1 MIC49300WR 1,2V
LDO UVTB1 MIC49300WR 1,2V
picoTDC VDD_DIGITAL VDD_TM VDD_PLL VDD_DLL
picoTDC VDD_DIGITAL VDD_TM VDD_PLL VDD_DLL
125mA
375mA
<1mA
10mA
35mA
5mA
115mA
10mA
310mA
1395mA
1315mA
VPOWIN
V5p0MAIN
V1p0PCORE
V1p0PVDDA
V1p8PVDD18
V2p5PVDD25
V2p5PVDDA25
V2p5PVDD25L
V3p3FPGA
USB3_VBUS
V3p3FPOW
V2p5MAIN
V1p8FVIO
V1p0FCOR
V1p0FPLL
V2p5PBANK
V1p8MAIN
V1p8PBANK
V5p0MAIN
V2p5EVIO
V3p3ETH
V3p3MAIN
V1p8EMAC
V1p8EMACF
V1p0EVDD
V1p0EVDD
V1p2TDCA
V1p2TDCB

Figure 1.8: *Schematic of the Power Rail circuit of the PicoTDC Board.*

The schematic shown in Figure 1.8 represents how the power rail is built and the components connected to the different voltages. In particular, taking a look at the various components:

- **PolarFire FPGA**: The core of the whole system is connected to various power rails ranging from 1.0 V to 3.3 V in order to power the banks, the transceivers, the power interfaces and the FPGA core with the correct operating voltages.

- **PicoTDC ASICs**: The power for these two important components of the board is provided by the 1.8 Vline through two LDOs set to the 1.2 V level.
- **Ethernet PHY chip**: This chip is connected to the 3.3 V and the 5.0 V power rails and uses the LDOs to generate 2.5 V,1.8 V and 1.0 V
- **Cypress EZ-USB FX3 mezzanine**: The power for the mezzanine, in this case, is provided directly by the 5.0 V power rail to power the whole mezzanine. The circuit to generate the working voltages for the mounted electronics is already integrated on the mezzanine itself.
- **FTDI FT601Q chip**: This chip is connected to the 5.0 V power switch through a 3.3 V LDO, to the 2.5 Vswitch through the 1.8 V LDO. Furthermore, the FTDI chip features an integrated 1.0 V LDO which provides the correct voltage to its core and PLL circuits.

One important observation regards the FPGA bank 7, in fact the PolarFire FPGA I/O architecture is composed by only six banks. However, the PolarFire power supply includes one more bank that needs to be powered with no I/O pins available.

1.2.2 PolarFire FPGA

The component chosen to implement the logic functions and the hardware control for this prototype is a Microchip Technology PolarFire MPF200T FCG784E FPGA[2]. This choice was made for the properties of this particular component, in fact, being the PolarFire a flash memory FPGA, it ensures low power consumption and a reliable behaviour during radiation exposure due to its SEU[3] immunity of the configuration memory.

This FPGA includes a system controller, security encryption features, many user-programmable I/O pins, a logic fabric and a 16-lane transceiver which communicates with it[4].

The logic fabric is composed by 192K logic elements, each composed by four LUTs[4] and one FFD, 588 mathematical units which features 18x18 MACC[5] to implement digital filters. The FPGA also integrates four kinds of memory blocks:

- **μPROM**: A non-volatile memory writable at programming time and readable at runtime. The construction of this memory block is designed to be SEU-immune and therefore useful to store data about parametric and initialization data.

[2]Field Progammable Gate Array
[3]Single Event Upset
[4]Look-Up Table
[5]Multiply-ACCumulate

- **LSRAM**: A volatile memory made of interleaving cells of 20 KB static RAM and provided with SECDED[6] features.
- μ**RAM**: A smaller RAM unit of 64x12 Bytes implemented using latches and with a SEU immunity feature.
- **sNVM**: 56 KB of non-volatile memory readable and writable at runtime by user service calls made through the csystem controller. Main utility of this memory block is to initialize **LSRAM** and μ**RAM** with secure data.

The FPGA provides a global network to route clocks and controls signals within large sections with low skew and regional networks which takes the clock signals only to limited domains of the FPGA hardware. Furthermore, it's provided with eight DLLs and eight PLLs for clock signals management excluding the PLLs of the transceiver lanes.

Taking a look at the I/O section of the FPGA, this is organized in six I/O banks which can host user defined I/Os, sharing the same VDD power and reference voltage. In total, up to 368 I/O pins support both single-ended and differential standards. For the digital logic I/O to the fabric, the FPGA is equipped with I/O delay chains, registers and control logic for the various modes[5].

The FPGA I/O banks, shown in Figure 1.9 are organized as follows:

- **Bank 0**: Provides connections to the Ethernet PHY chip and to the FTDI FT601Q chip through HSIO[7]. Furthermore, this bank is connected to the I2C lines for the configuration of both PicoTDCs through a voltage translator to reach the voltage level of 1.2 V needed by the ASICs.
- **Bank 1**: Provides HSIO pins to manage the connection with the Cypress EZ-USB FX3 mezzanine.
- **Bank 2**: Provides GPIO[8] pins for the connection of control and data read-out with the PicoTDC A together with some Ethernet signals.
- **Bank 3**: Provides pin connection for the FPGA programming via JTAG/SPI standard and for the global resets of the FPGA features.
- **Bank 4**: Similarly to Bank 2 provides pin connections to the PicoTDC B for control and data read-out, but it also connected to the on-board differential 40 MHz clock, some LEDs and switches.

[6]Single Error Correction and Double Error Detection
[7]High-Speed Input Output
[8]General Purpose Input Output

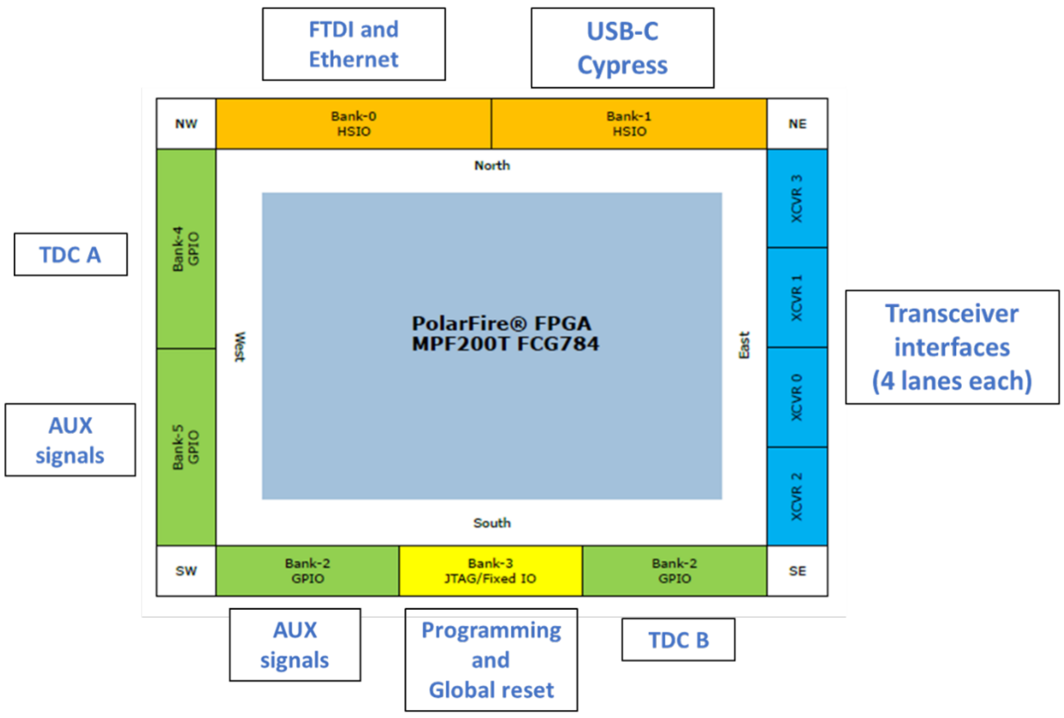


Figure 1.9: *Scheme of the banks locations of the PolarFire FPGA mounted on the PicoTDC board together with the relative connections with the various components.*

- **Bank 5**: Provides some GPIO pins for external clock and reset signals, some pin connections wich are routed to both the FMCs connectors and two JVS strips used for firmware debugging.

The FPGA system controller is based on the Cortex_M3 ARM processor to ensure the correct power-up procedure, functioning and correct responses to the system service calls. The system service allows the user to check the informations of the FPGA state and to call some specific system controller actions.

As for the programming features of the PolarFire FPGA, there are two possible working modes. The first, in which the FPGA acts as slave, the device flash memory is programmed via the JTAG connector or an external SPI master, while in the latter the FPGA acts as master and checks for an external SPI flash memory to update or reboot the firmware[4].

The decision to use this kind of FPGA over the SRAM type comes from various aspects like their immunity to SEUs in the configuration bits that can be caused by the radiation to which the electronics is exposed and due to the low power consumption performance using CMOS technology.

1.2.3 Ethernet interface

For the Ethernet communication interface the PicoTDC board is equipped with a VSC8541-05 PHY chip, providing a 1 Gbps Ethernet connection with the external PC and it's configured to allow a physical connection through a twisted pair Ethernet cable.

Within the framework of a node, the PHY chip communicates its information through a sub-layer called MAC[9] that manages the data transmission going in and out of the PHY chip. The MAC can be implemented inside the FPGA firmware with the help of some specific modules. A firmware containing the procedures for the MAC/PHY interface has been already developed and it features a full-duplex data 1 Gbps Ethernet connection[6].

The PHY chip mounted on the board implements an RGMII[10] interface, which provides DDR[11] data communication over four lines using a 125 MHz clock[7]. The VSC8541-05 is, on one end directly connected to the PolarFire FPGA, while on the other is connected to an RJ-45 connector through a transformer to modulate I/O voltage levels.

Furthermore, the PHY device can be configured by hardware strapping or by writing the internal registers via a SMI[12] interface. By using the first method some care must be taken in account since this method associates configurations instructions to some pull-up/pull-down logic values that get sampled on the deassertion of the NRESET signal, so any device must not be driving the related pins signals until the configuration is completed[8].

1.2.4 Cypress EZ-USB FX3 interface

The Cypress EZ-USB FX3 mezzanine card, developed by Infineon, provides one of the two possible SuperSpeed USB interfaces of the PicoTDC board. As shown in Figure 1.10 this mezzanine card features[9]:

- **USB integration**: the mezzanine card is provided with USB 3.2 Gen1 and USB 2.0 peripherals, both compliant with USB 3.2 Specification, a 5 Gbps SuperSpeed PHY chip compliant with the USB 3.2 Gen1 peripheral and 32 physical USB endpoints.

- **General Programmable Interface (GPIF II)**: A programmable interface working at a frequency of 100 MHz which enables connectivity with a wide range of external devices through an 8-,26-,24-,32-bit data bus and up to 16 control signals.

[9]Medium Access Control
[10]Reduced Gigabit Media Independent Interface
[11]Double Data Rate
[12]Serial Management Interface

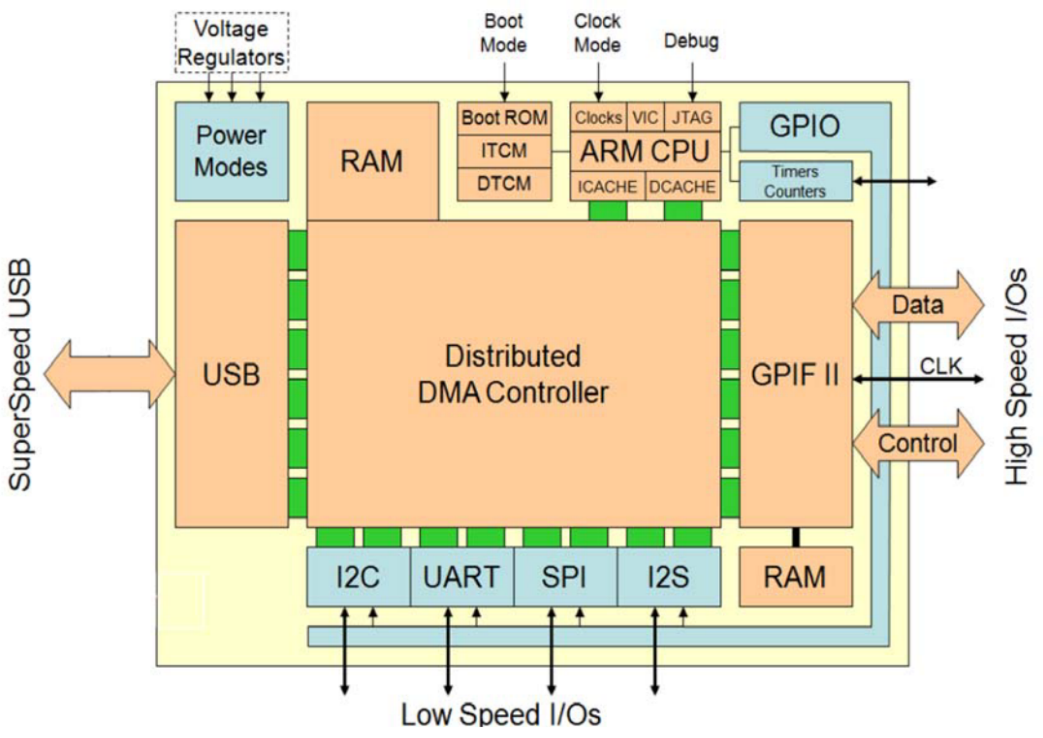


Figure 1.10: *Cypress EZ-USB FX3 mezzanine card logic block diagram.*

- **32-bit CPU**: An ARM based CPU with an ARM926EJ core with 200 MHz operations and 256 or 512 KB of embedded SRAM.
- **Selectable clock input**: The working clock can be provided by a 19.2 MHz crystal or provided externally. The supported frequencies are 19.2, 26, 38.4, and 52 MHz.
- **Additional connectivity**: Multiple interfaces for different standars are provided; in particular the card features an SPI master up to 33 MHz, UART support up to 4 Mbps, I^2C master controller at 1 MHz and an I^2S master transmitter at various frequencies.

One example of the firmware architecture that might have been implemented in order to create an USB interface for the PicoTDC board is shown in figure 1.11. In this case the PolarFire FPGA would have taken the role of the external master and managed the data exchange between the board and the mezzanine on-chip memory buffers which would have been read or written by the external USB Host.

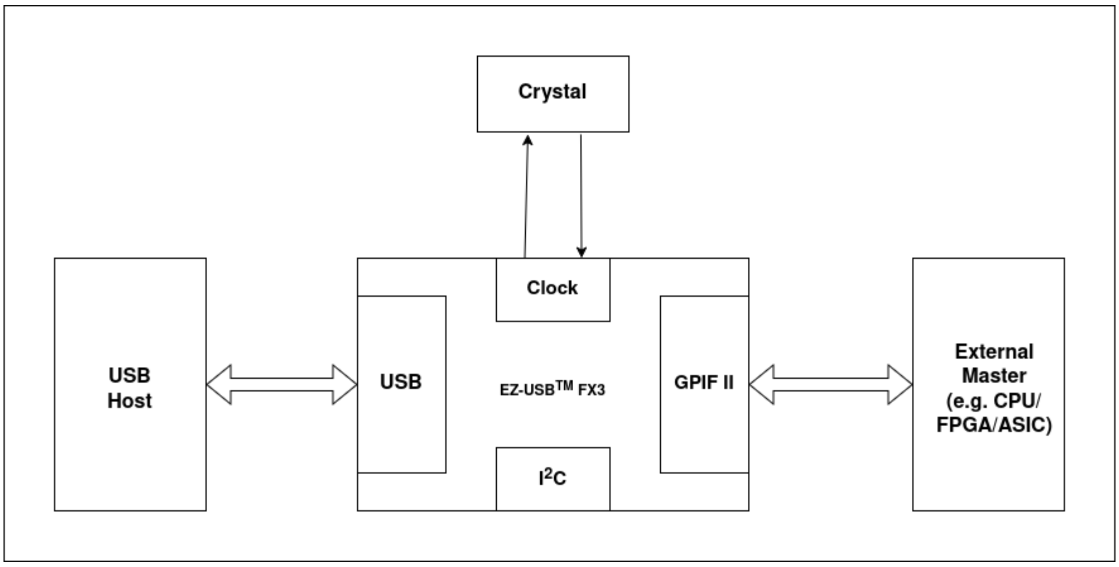


Figure 1.11: *Example of a possible firmware architecture to implement a USB interface through the Cypress card.*

1.2.5 FTDI FT601Q USB interface

The other possible USB interface is given by a FT601Q SuperSpeed USB to FIFO bridge produced by FTDI[13] Ltd. Figure 1.12 shows the block diagram of the chip.

Looking at the chip block diagram in Figure 1.12 it is equipped with the following features[10]:

- **USB 3.0 and USB 2.0 PHY chips**: Through those chipsets, the FT601Q can support USB 3.0 SuperSpeed (5 Gbps), USB 2.0 HighSpeed (480 Mbps) or FullSpeed (12 Mbps) transfer through a USB 3.0 Micro-B port. Furthermore, it can handle Control, Interrupt or Bulk USB transfer types through 8 configurable endpoints.

- **Two FIFO bus protocols**: Two different protocols for the 32-bit parallel interface with a burst data rate up to 3.2 Gbps.

- **Configurable GPIO and USB descriptors**: Two GPIO pins and a NVM can be used to set the working mode and the internal configuration of the FT601Q chip.

- **Selectable clock**: The clock input of the FTDI chip is fed by a 30 MHz crystal and through a PLL the working frequency of the FIFO bus can be

[13]Future Technology Devices International

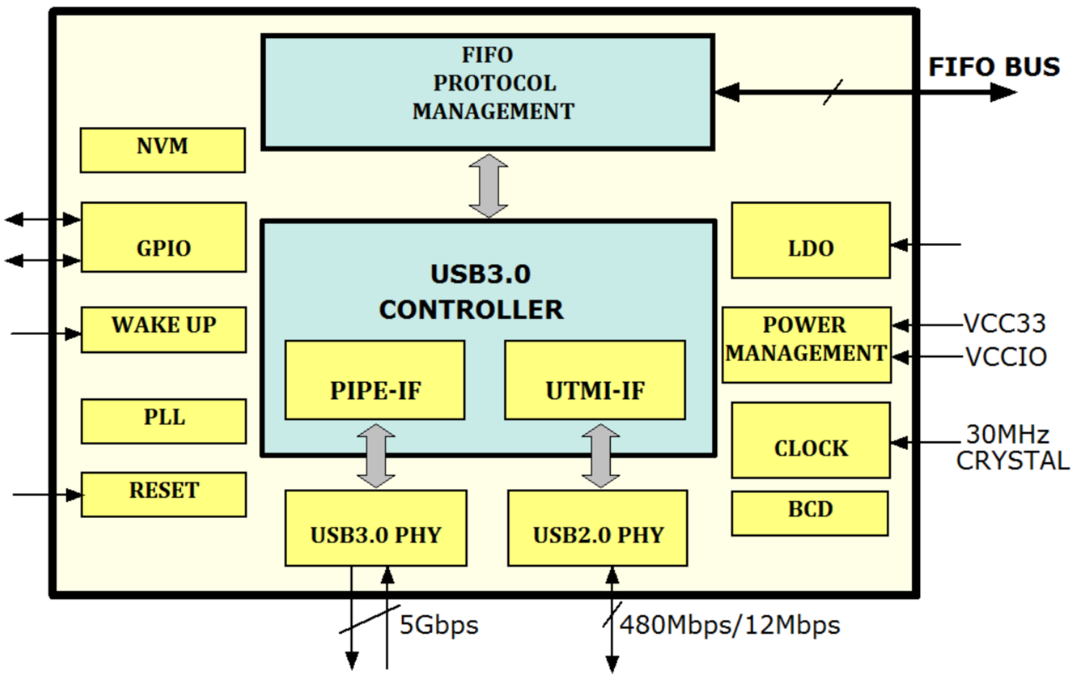


Figure 1.12: *Block diagram of the FT601Q USB 3.0 to FIFO bridge chip.*

selected. The supported frequencies are 33, 50, 66, or 100 MHz.

- **Built-in memory, wake-up and reset**: A 16 KB RAM is used as data buffer between the FIFO bus and the USB controller. Furthermore, the chip is provided with remote wake-up and reset capabilities.

1.3 IPBus project

1.3.1 μHAL and IPbus project

The IPbus is a communication protocol developed in 2009 by J. Mans et al. and used to control different hardware features of an electronic board by the exploitation of logic modules in the firmware of an FPGA. The original project features a complete suite of firmware and software to control a large network of devices connected to different endpoints through a UDP/IP layered model over the Ethernet protocol[11].

The UDP[14] is a connectionless protocol which provides basic information for the transport layer, such as the source and the destination port addresses. This particular choice was made to ensure a simple and fast communication but at the cost of less reliability; in fact, if an error occurs the data packet it's dropped or the connection might get closed in case of congestion.

At software level, μHAL provides a Hardware Access Library which acts as end-user *Python/C++* API to implement IPbus transactions between applications and target. In order to have multiple μHAL designed programs to communicate properly, the Control Hub application acts as arbiter between the various processes; in particular it communicates with each application using the TCP[15] protocol to ensure a reliable connection from and to the target device.

1.3.2 Bus on-chip architecture

The firmware implementation of the on-chip IPbus is based on the Wishbone SoC[16] bus, which provides a common interface between various IP cores[12]. This standard was developed to improve reusability and compatibility and to solve integration problems and limitation in the development of the SoC.

The architecture of this standard is designed using a master-slave architecture in which the master can address the slave for data transfer, in particular the width of both data and address buses are variable between 0 and 64 bits. The communication between master and slave is synchronous with the bus clock and controlled by an handshaking logic featured by specific and optional signals. In particular, two signals, strobe and ack, are respectively asserted by the master to signal a valid data transfer and by the slave to end the transaction[13].

As shown in Figure 1.13, the IPbus features a hierarchical topology design in which a single bus master is connected to multiple slaves via a bus fabric selector that manages the address decoding and selects the correct slave bus to connect to the master. The bus architecture is designed to have a point-to-point A32/D32 structure and features two separate buses for the data transfer protocol, a

[14]User Datagram Protocol
[15]Transmission Control Protocol
[16]System on Chip

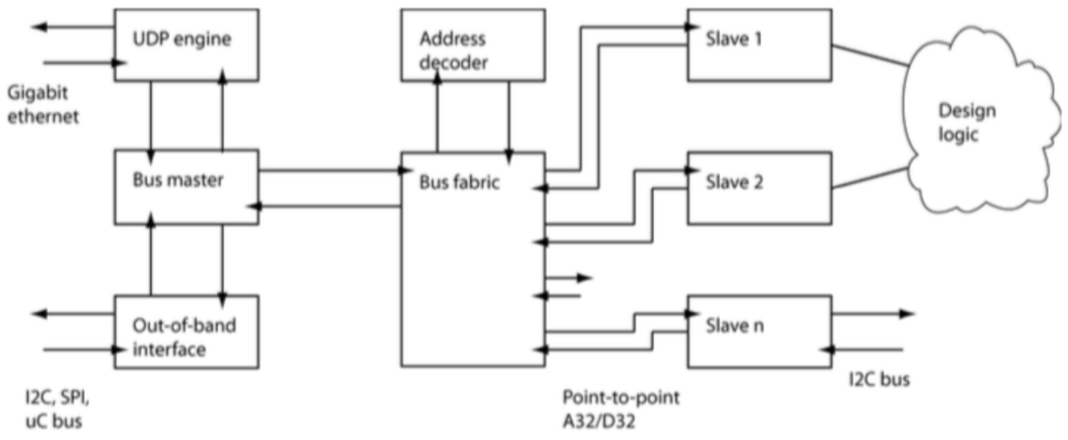


Figure 1.13: *Block diagram of an example IPbus on-chip firmware architecture.*

Bus Type	**Direction**	**Signal**	**Width** (bits)	**Description**
ipb_in	Master to Slave	ipb_addr	32	Address Bus
		ipb_wdata	32	Data to be written to the slave
		ipb_write	1	Takes value 1 for a write cycle and 0 for a read cycle
		ipb_strobe	1	Asserted when valid data and address are set on the respective lines and marks the start of cycle
ipb_out	Slave to Master	ipb_rdata	32	Data read from slave
		ipb_ack	1	Acknowledge signal asserted to end the transaction
		ipb_err	1	Error flag asserted to end transaction in case of error

Table 1.1: *IPbus signals description.*

breakdown of the bus structure is given in Table 1.1. On the ipb_in bus the master manages the transaction type, the slave address, the input data to the slaves and starts the transactions, while on the ipb_out bus the slave sends the response to the master.

The on-chip bus is fully synchronous with a single system clock signal, and there are theoretically no constraints on the relationship occuring between the transport layer interface clock and the bus clock. For slaves that do not require wait states, the 32-bit data bus width allows for a full usage of a Gigabit Ethernet interface as long as the bus clock is set to frequencies ≥ 31.25 MHz, which is one fourth of the Gigabit Ethernet standard clock (i.e. 125 MHz).

1.3.3 IPBus communication protocol

As described in the previous section, the bus cycle is initiated by the bus master when a valid command is received through the Ethernet interface; in particular there are two main transfer types: the write cycle and the read cycle. Keeping in mind the signal description in Table 1.1, the master initiates the transaction by asserting the ipb_strobe signal and then waits for the assertion of the ipb_ack or the ipb_err signals by the slave to end the transaction.

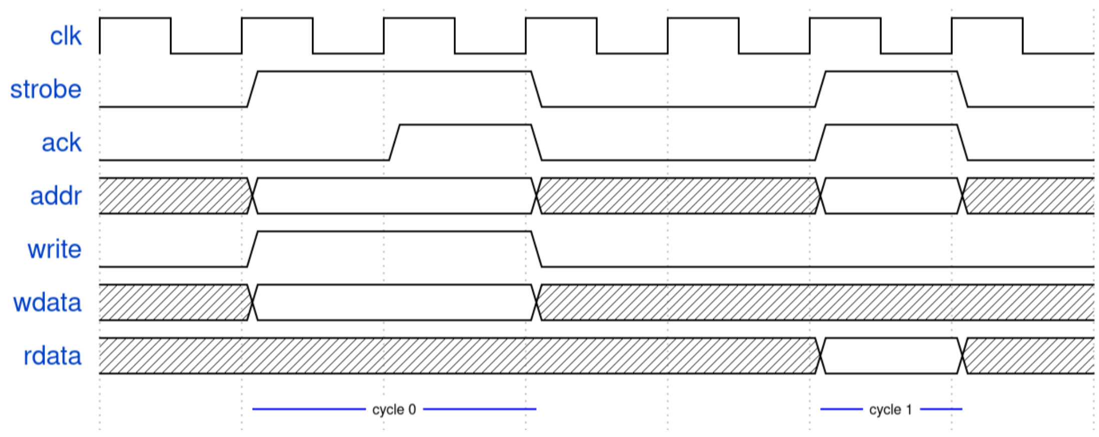


Figure 1.14: *Time diagram of an IPbus write cycle (cycle 0) followed by a read transaction (cycle 1).*

In Figure 1.14 a scheme of the IPbus protocol for a write operation followed by a read operation. A bus cycle starts with the master driving the address and the write bit in the bus, along with the possibile data to be written in case of a write transaction, and asserts the strobe signal. The cycle ends when the ack or err signal is asserted by the addressed slave; the assertion of the handshaking signals can happen on the same cycle of the assertion of the strobe in case of a zero-wait slave or can be delayed by one or more clock cycles with respect to the strobe signal being asserted. The master register the data upon ack signal assertion[13].

After the cycle has been terminated by the reception of the ack/err signal, the master can either deassert the strobe or load another address and/or data on the bus keeping the strobe signal high to signal a new cycle immediatly after the one that has just terminated.

Even though the IPbus protocol is based on the Wishbone SoC standard, there are two important differences given by the fact that the master is not required to deassert the strobe signals between different cycle, but it is, however, guaranteed to deassert the strobe or begin a new cycle on the clock cycle after the ack reception. Furthermore, the slaves are not allowed to tie the ack signal high and must deassert it on the same clock cycle in which the strobe signal is deasserted, but it is possible to tie the ack to the strobe if a zero-wait state is possible.

1.3.4 IPBus packet structure

Let us now take a look at the structure of an IPbus control packet going from a μHAL-based software client to a hardware target. Looking at the example shown in Figure 1.15 the IPbus-specific information is contained into the payload of a UDP packet which is itself wrapped inside an IP[17] packet inside an Ethernet packet[14].

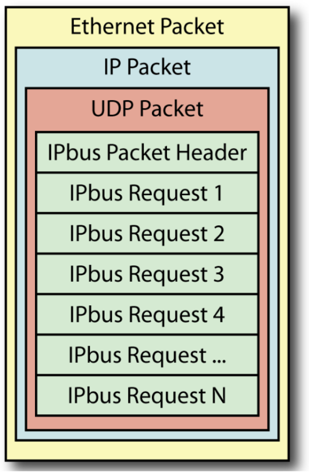


Figure 1.15: *Example of an Ethernet packet sent from the software client to the target hardware.*

Regarding the IPbus packet content: there is a first IPbus packet header followed by one or more IPbus transaction headers and their body containing the relevant informations for the specific transaction. The bit fields of the header words are shown in Figure 1.16.

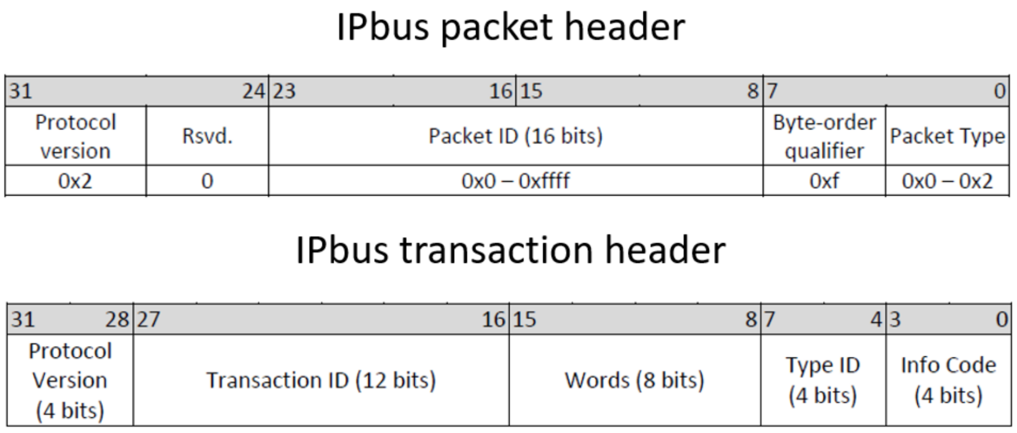
The request and the reply of an IPbus packet always begin with a 32-bit header. This header has been introduced in version 2.0 to support reliability features, in fact, if a IPbus target receives an invalid packet header it silently drops the packet.

[17]Internet Protocol

The combination of the Protocol Version and the Byte-order Qualifier fields provide a simple system to determine the byte ordering of the 32-bit word (i.e. its endianness) from the header word. The target firmware is capable of handling both big and little endian requests and replies with the same endianness; this feature allows the client software to communicate with the target hardware using its native endianness optimizing CPU usage of the control computers[14].

The Packet ID field has the purpose to provide a reliable communication while using an unreliable transport protocol like UDP.

For the study case the most important packet header is the Control Packet (defined by the Packet Type value of 0x0) which is the one followed by IPbus transactions.



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 1.16: *Bit fields of the 32-bit words of the IPbus packet header and transaction header.*

A Control Packet is the concatenation of its header and one or more IPbus transactions. Each transaction is composed by an IPbus transaction header and a body containing the address of the on-chip slave and the optional data to be written. The format of the transaction header is the following:

- **Protocol Version** (bits 31 to 28): Define the protocol version of the packet header, in this case it must be set to 0x2.

- **Transaction ID** (bits 27 to 16): This field contains the information of the transaction identifier to allow the client/target to track each transaction within a given packet.

- **Words** (bits 15 to 8): Contains the information of the number of words within the addressable memory of the bus to be processed. In case of block reads/writes defines the read/write size. Since this information is coded by an 8-bit word, transactions of size greater than 255 words must be split in multiple transactions.
- **Type ID** (bits 7 to 4): Define the type of the transaction (i.e. read(0x2) or write(0x3)).
- **Info Code** (bits 3 to 0): This field contains the information on the direction and error state of a transaction request or response. Every request made by the client software to the target must set this field to the value of 0xf. An Info code different from 0x0 in the transaction response header indicates a non-successful transaction by the target.

After a request transaction header the Control Packet continues with the 32-bit word containing the address of the intended slave and, in the case of a write transaction, the 32-bit word containing the data to be written.

The response control packet is then composed by the target and sent back to the requesting client. This packet is composed by the response header with the appropriate Info code (Table 1.2) and followed by one or more 32-bit words in the case of a read/read block transaction.

Packet Type Value	**Direction**	**Meaning**
0x0	Response	Request handled successfully
0x1	Response	Bad Header
0x4	Response	Bus error on Read Transaction
0x5	Response	Bus error on Write Transaction
0x6	Response	Bus timeout on Read Transaction
0x7	Response	Bus timeout on Write Transaction
0xf	Request	Outbound request

Table 1.2: *List of the possible Info Codes of the IPbus transaction headers and their meaning.*

Chapter 2

Firmware and Software adaptation for the USB interface

During the making of the work for this thesis, the firmware adaptation for the FT601Q USB FIFO bridge interface was developed to test the usage of a different interface from the original Ethernet one.

The change in the interface and in the protocol of the transport layer allowed the simplification of the protocol both at software and firmware level, providing a more direct communication between the on-board IPbus infrastructure and the external control computer.

Since the USB connection provides point-to-point connection between the different endpoints, the need of a complex packet structure was overcome sending only the transaction packets directly to the proper USB interface without the need to decode and encode the request and the response packets respectively.

The solutions adopted during the development of the firmware and software will be explained in this chapter.

2.1 FTDI FT601Q FIFO Mode Protocols

As mentioned in Section 1.2.5, the chip provides two different working modes to communicate with the bus master through the 32-bit wide data bus. By design the bus clock is provided by the slave to the master.

The first working mode, called FT600, is capable of multichannel connectivity up to 4 bidirectional channels corresponding each to 1 USB IN and 1 USB OUT endpoints with a bus clock frequency of 33 MHz or 50 MHz, while the second one, called FT245, provides high speed single channel communication with clock frequency of 66 or 100 MHz. It is important to notice that both protocols are synchronous to the falling edge of the bus clock[10].

When the chip is configured to work in the FT245 mode or in the 1 channel FT600 the FIFO buffer is configured as 4 KB * 2 (double buffered) each on the RX and TX channels. When the number of channels increase to 2 or 4 the FIFO buffer is split in the same way with 2 KB * 2 or 1 KB * 2 respectively.

The other signals used in the two different communication protocols are:

- **TXE_N**: Active low Slave to Master signal indicating Transmit FIFO Empty in the FT245 mode or Status Valid in the FT600 mode.
- **RXF_N**: Active low Slave to Master signal indicating Receive FIFO Full in the FT245 mode or Data Receive Acknowledge in the FT600 mode.
- **WR_N**: Active low Master to Slave signal acting as Write Enable in the FT245 mode or Data Transaction Request in the FT600 mode.
- **RD_N**: Active low Master to Slave signal acting as Read Enable in the FT245 mode.
- **OE_N**: Active low Master to Slave signal acting as Data Output Enable in the FT245 mode.
- **BE[3:0]**: Active high bidirectional signals acting as byte enable for the 4 byte wide parallel bus.
- **GPIO[1:0]**: General Purpose I/O signals used, in this case to set the working mode of the FTDI chip.
- **F_DBUS[31:0]**: 32-bit wide data bus for data transmission between the FPGA (Bus master) and the FTDI chip.

2.1.1 FT600 Multi-Channel Protocol

As mentioned before, the FT600 Multi-Channel mode provides connectivity with up to 4 channels each mapped to USB IN and OUT endpoints. For the sake of clarity in the protocol explanation we will refer to FIFO IN or FIFO OUT which correspond respectively to the USB OUT and USB IN endpoints. The FIFO OUT is for data transmitted from the USB Host to the device (USB IN) and the FIFO IN is for data transmitted from the device to the USB Host(USB OUT).

This protocol makes mainly usage of the WR_N, RXF_N, BE[3:0], DATA[31:0] data bus and optionally the TXE_N signals to manage the communication between the FPGA, that acts as master, and the FTDI chip. In particular when the bus is in the idle state DATA[31:16], DATA[7:0] and BE[3:0] are driven to logic ”1” by the bus master, and DATA[15:8] is driven by the bus slave to provide the FIFO status to the bus master. The upper nibble (DATA[15:12]) provides the 4 OUT channels FIFO status, while the lower nibble (DATA[11:8]) provides the 4 IN channels FIFO status. They are all active low.

The bus master starts a transfer cycle by asserting the WR_N signal based on the FIFO status followed by the command phase at the next clock cycle and the data phase when RXF_N is asserted. At command phase, the master will send the channel number which it intends to transfer data with on DATA[7:0] and a Read/Write command on BE[3:0]. The channels are mapped 1 to 4 with the corresponding hexadecimal values 0x01 to 0x04, while for the command value 0x0 corresponds to a Master Read transaction and 0x1 to a Master Write transaction. There may also be a required turn-around phase for DATA[31:0] and BE[3:0] after the command phase and at the end of the data phase[10].

Examples of a Master Read and Master Write transactions waveform are shown in Figure 2.1 and in Figure 2.2 respectively.

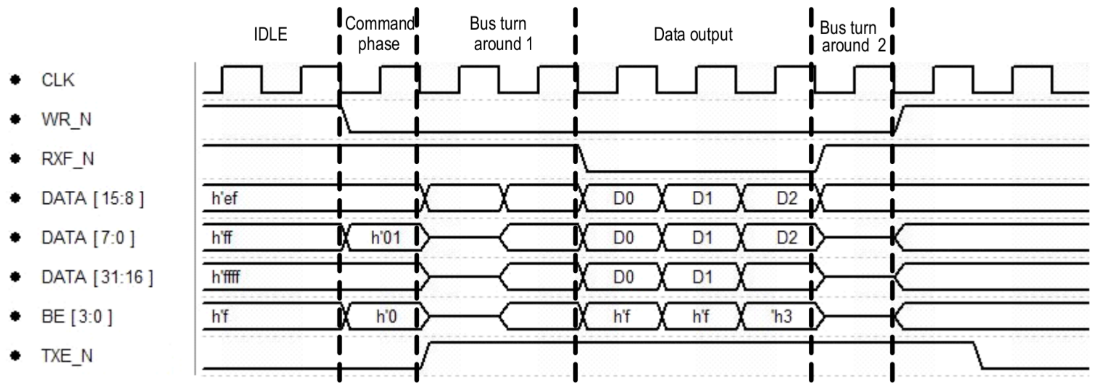


Figure 2.1: *Waveform of a Master Read transaction for the FT600 Multi-Channel mode.*

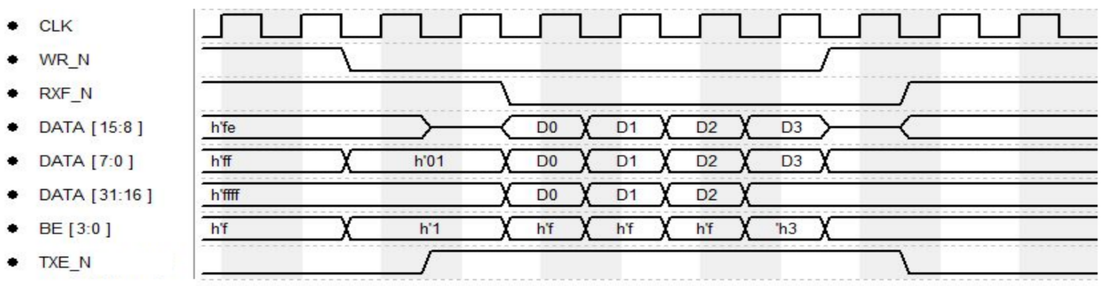


Figure 2.2: *Waveform of a Master Write transaction for the FT600 Multi-Channel mode.*

2.1.2 FT245 Synchronous Protocol

The FT245 Synchronous mode, contrary to the previous mode, uses only one IN and one OUT FIFO channel but supports higher clock frequencies, resulting in a theoretically higher bandwith.

The signals used in the protocol for this mode are TXE_N, RXF_N, OE_N, RD_N, WR_N together with the DATA[31:0] data bus and the BE[3:0] signals. In this mode, the slave (FTDI chip) signals its status to the master (FPGA) through the RXF_N and TXE_N signals, instead of using the data bus bits as in the previous mode. Since only one channel is available and these two signals are used to signal to the master to start a Master Read or Master Write transaction, the need for the command phase is removed making the protocol more intuitive.

When the USB Host writes data to the device, the FTDI chip asserts the RXF_N signal; the master then starts the read cycle by asserting first the OE_N signal and the RD_N in the next clock cycle. On the same cycle of the assertion of the RD_N signal, the slave drives the DATA bus and the BE valid byte to the correct values. On the other hand, when the USB Host request to read a certain number of bytes from the device, the TXE_N signal is instead asserted, signaling the master to start a Master Write transaction. In this case, the master asserts the WR_N signal while driving the DATA bus and the BE valid byte to the correct values. Normally all the 4 bytes should be valid in a bus transaction with the exception of the last word when the data transaction length is not aligned[10].

Examples of a Master Read and Master Write transactions waveform are shown in Figure 2.3 and in Figure 2.4 respectively. Both transactions are terminated by the slave with the deassertion of the TXE_N or RXF_N after the requested number of bytes have been transferred.

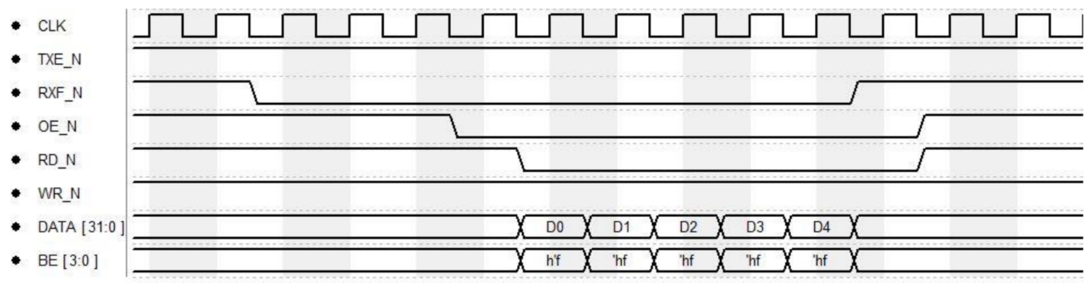


Figure 2.3: *Waveform of a Master Read transaction for the FT245 Synchronous mode.*

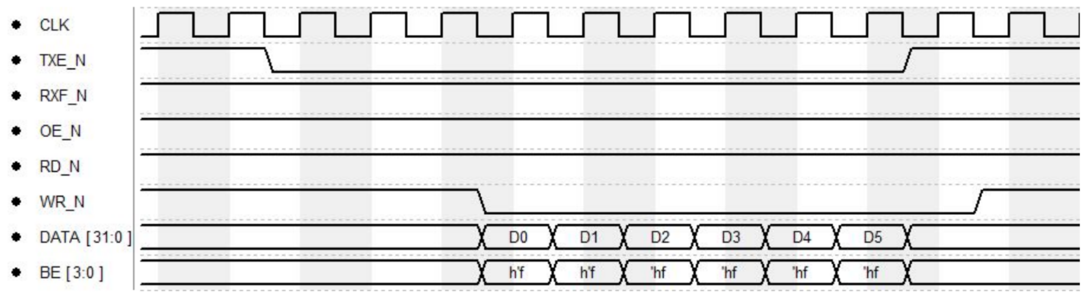


Figure 2.4: *Waveform of a Master Write transaction for the FT245 Synchronous mode.*

2.2 Firmware adaptation

Examining the needs of the board application, the FT245 Synchronous mode was chosen to build the fastest possible interface. Considering also that the IPBus is built around a single master, there was no need to utilize more than one channel to communicate with the board.

The working principle behind the board interface operations follows a precise sequence of steps involving operations made by the USB Host, the FTDI bus master and the IPBus master. The procedure is shown in Figure 2.5.

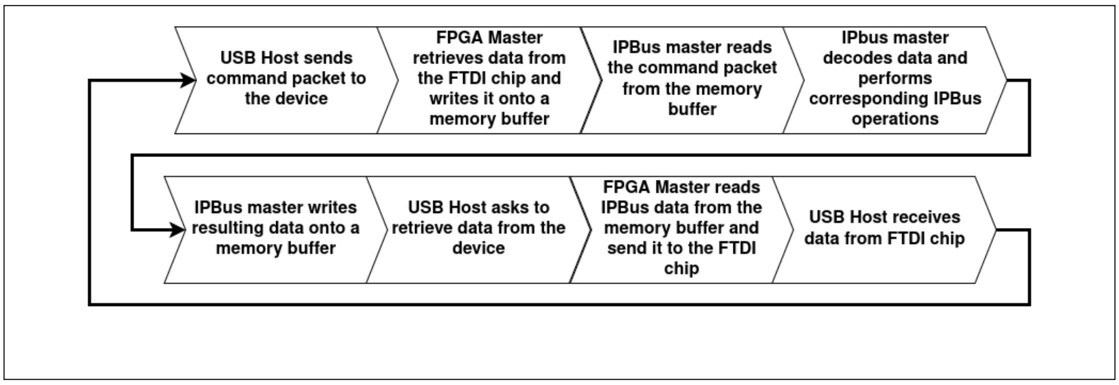

USB Host sends command packet to the device
FPGA Master retrieves data from the FTDI chip and writes it onto a memory buffer
IPBus master reads the command packet from the memory buffer
IPbus master decodes data and performs corresponding IPBus operations
IPBus master writes resulting data onto a memory buffer
USB Host asks to retrieve data from the device
FPGA Master reads IPBus data from the memory buffer and send it to the FTDI chip
USB Host receives data from FTDI chip

Figure 2.5: *Communication procedure used to perform board control and readout operations.*

In order to integrate this new interface with the previous IPBus SoC, several changes were made at firmware level that will be further explained in this section.

2.2.1 FTDI Control Interface

The control interface firmware implementation provides I/O ports to communicate with both the FTDI bus and the IPBus SoC and contains two Finite States Machines, two FIFO memory buffers and a process which manages the reset logic for both the IPBus SoC and the control interface. A block diagram of the control interface is provided in Figure 2.6.

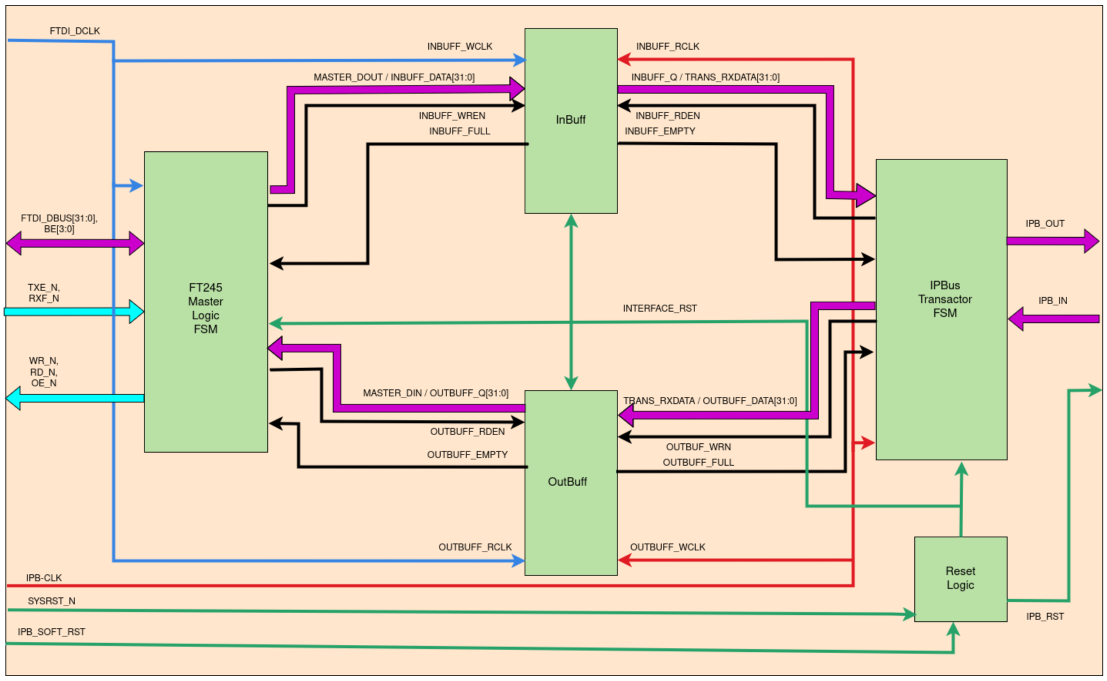


Figure 2.6: *Block Diagram of the control interface firmware. Blue and red thin arrows are related to the two clock domains (FTDI side and IPBus side respectively), magenta arrows represents data lines, cyan arrows represent FTDI control signals, green arrows are related to the reset logic and black arrows represent memory management signals.*

Following the data flow, the first block involved is the one that manages the communication protocol with the FTDI USB chip (FT245 Master Logic FSM), it receives signals and data from the chip and writes the received data into the first 1024 words x 32 bits deep FIFO memory buffer (InBuff) while monitoring the FULL flag.
After data has been written onto the InBuff buffer the EMPTY flag is deasserted signaling to the IPBus master (IPBus Transactor FSM) that one or more command packets are ready to be read; this condition makes the Transactor initiate the reading, decoding and execution of the received commands on the IPBus side and writes the resulting data into the second 65536 words x 32 bits deep FIFO memory

buffer (OutBuff).
Then, upon request from the FTDI chip, the FT245 Master starts to read data from the OutBuff memory and writes it onto the chip memory buffers from where they are then sent to the external USB Host.

Since the FTDI FT245 protocol and the IPBus protocol work at different clock frequencies, 100 MHz and 40 MHz respectively, and are synchronous to different clock edges, the two FIFO memory buffers needed to have Dual Clock capabilities in order to match the two clock domains, performing operations on the falling edge of the 100 MHz clock for the FTDI side and on the rising edge of the 40 MHz clock for the IPBus side. Furthermore, the InBuff memory is set to work into First Word Fall Through mode, providing the first word as soon the EMPTY flag signal is deasserted, while the OutBuff memory is set to work in Prefetch mode allowing single clock cycle reading for the first word[15].
To optimize the FPGA memory usage the InBuff is set to have a much smaller depth, with regard to the OutBuff, since the command packets contains only 2 to 3 words per command, whereas the data size coming from the PicoTDC readout procedure can contain thousands of words at the time.

2.2.2 FT245 Master Logic

The FT245 Master Logic FSM provides the FPGA with the necessary information to achieve communication with the FTDI chip using the FT245 protocol and implements some fail-safe procedures. The State Machine is divided in two logic loops starting both from the idle state where it waits for the FTDI chip to request for a Read/Write transaction. A logic scheme of the state machine is shown in Figure 2.7.

Starting by the Master Write loop, it begins when the FTDI chip asserts the TXE_N; at this point if the OutBuff memory buffer is not empty the FPGA starts the corresponding procedure to write data to the FTDI chip by asserting simultaneously the WR_N and the OUTBUFF_RDEN signals and latching the FTDI data bus to the data output signals of the memory buffer. The cycle then ends correctly by the deassertion of the TXE_N by the FTDI chip or ends in a timeout error if there were less data words in the memory buffer than there were requested by the USB Host to the FTDI chip. A waveform example is shown in Figure 2.8

On the other hand the Master Read loop starts with the assertion of the RXF_N signals by the FTDI chip. After that the FPGA state machine switches to a turn-around phase in which it asserts the OE_N signal while latching the FTDI data bus to the input data of the InBuff memory and sets the FTDI data bus to high impedence to let the FTDI chip drive the bus. In the next clock cycle the INBUFF_WREN and the RD_N are asserted starting the data reading phase. This phase can be ended by the FTDI chip with the deassertion of the RXF_N signal

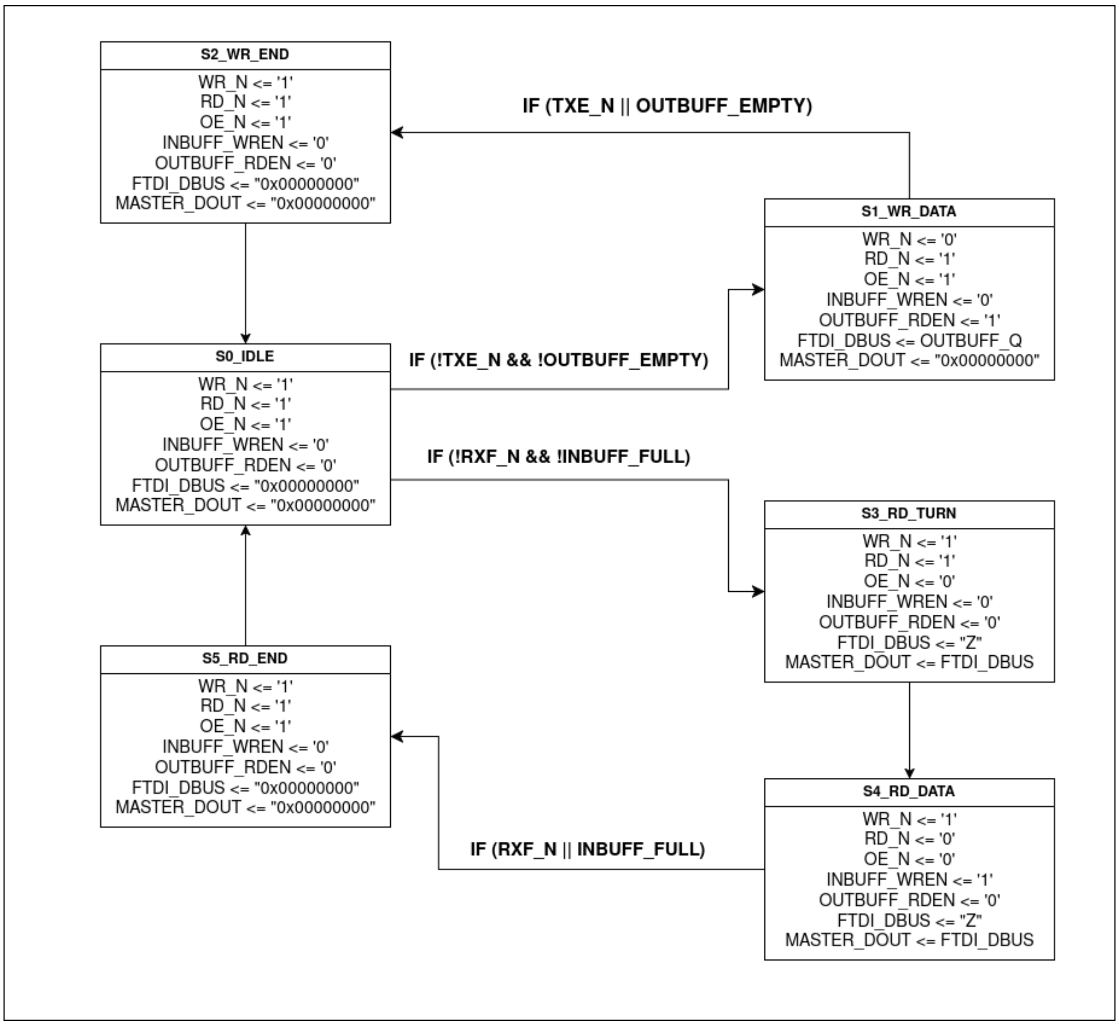


Figure 2.7: *Logic scheme of the Finite State Machine that implements master logic for the FT245 Synchronous protocol.*

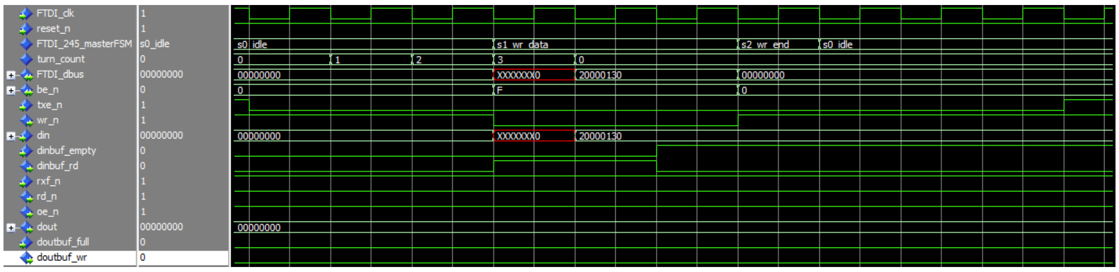


Figure 2.8: *Waveform simulation of the Master Write protocol managed by the FT245 Master Logic FSM.*

upon data transmission completion, or by the FPGA master in the case the InBuff memory becomes full causing a timeout error. A simulation of a Master Read cycle is shown in Figure 2.9.

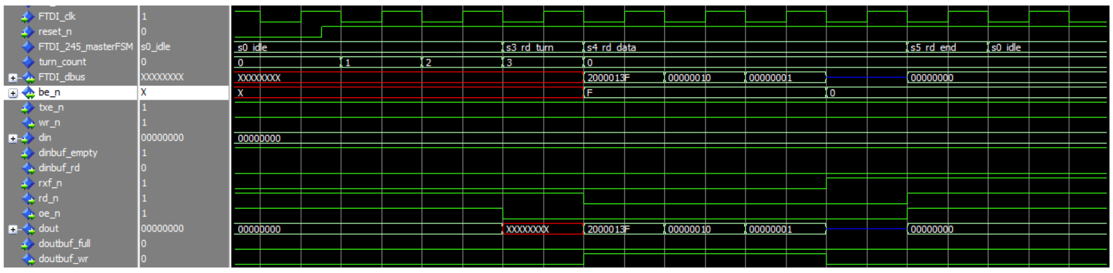
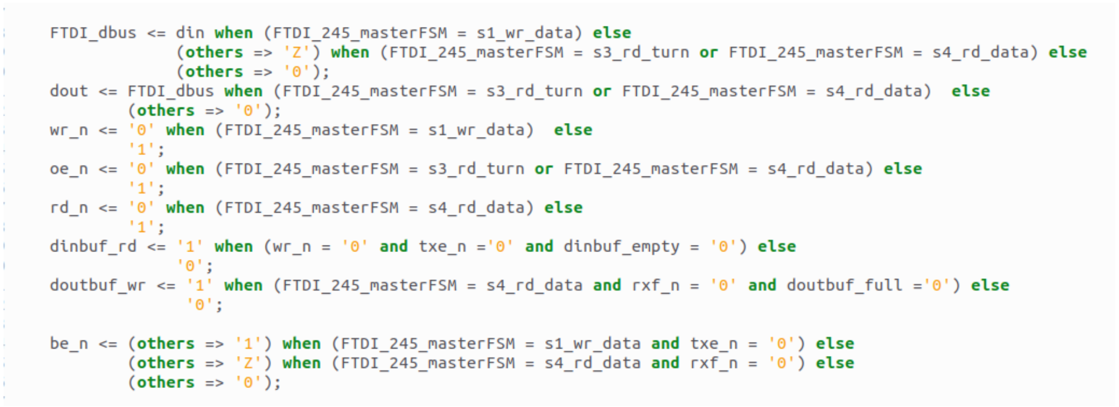


Figure 2.9: *Waveform simulation of the Master Read protocol managed by the FT245 Master Logic FSM.*

The VHDL code of the state machine which contains the output signals assignements is reported in Figure 2.10. It is visible that both InBuff Write Enable (doutbuf_wr in the code) and OutBuff Read Enable (dinbuff_rd in the code) are asserted conditionally to the state of the state machine and to the input signals coming from the FTDI chip in order to prevent an incorrect data transfer. In fact these two signals are deasserted immediately if the FTDI chip terminates the transfer or the buffer flags get asserted.



```vhdl
FTDI_dbus <= din when (FTDI_245_masterFSM = s1_wr_data) else
             (others => 'Z') when (FTDI_245_masterFSM = s3_rd_turn or FTDI_245_masterFSM = s4_rd_data) else
             (others => '0');
dout <= FTDI_dbus when (FTDI_245_masterFSM = s3_rd_turn or FTDI_245_masterFSM = s4_rd_data)  else
        (others => '0');
wr_n <= '0' when (FTDI_245_masterFSM = s1_wr_data)  else
        '1';
oe_n <= '0' when (FTDI_245_masterFSM = s3_rd_turn or FTDI_245_masterFSM = s4_rd_data) else
        '1';
rd_n <= '0' when (FTDI_245_masterFSM = s4_rd_data) else
        '1';
dinbuf_rd <= '1' when (wr_n = '0' and txe_n ='0' and dinbuf_empty = '0') else
             '0';
doutbuf_wr <= '1' when (FTDI_245_masterFSM = s4_rd_data and rxf_n = '0' and doutbuf_full ='0') else
              '0';

be_n <= (others => '1') when (FTDI_245_masterFSM = s1_wr_data and txe_n = '0') else
        (others => 'Z') when (FTDI_245_masterFSM = s4_rd_data and rxf_n = '0') else
        (others => '0');
```

Figure 2.10: *VHDL code containing the main signals assignements of the FT245 Master Logic FSM.*

2.2.3 IPBus Transactor

Looking at the IPBus side of the control infrastracture, the IPBus communication with the on-chip slaves is managed by the IPBus transactor FSM block containing the informations necessary to perform IPBus transactions and the interface with the system transport layer[13]. A waveform example of the functioning of the IPBus transactor is shown in Figure 2.11.

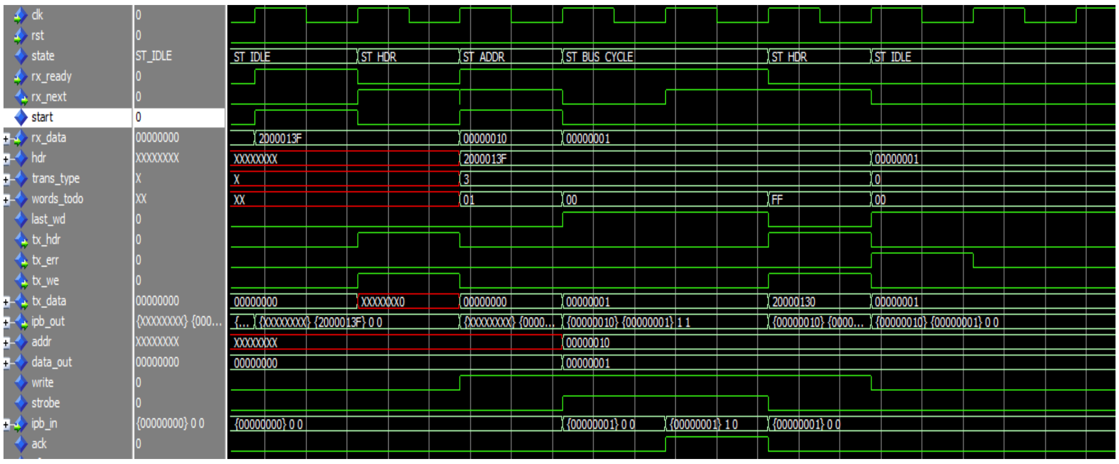


Figure 2.11: *Waveform simulation for an IPBus transaction made by the Transactor state machine.*

At first the state machine waits in the idle state for the start condition to trigger the transaction; the start signal is given by the transition of the rx_ready input signal from a logic 0 to a logic 1 which is related to the deassertion of the InBuff EMPTY flag.
When this signal is received by the state machine, the FSM transitions to the header state where the rx_next signal is asserted enabling the reading of the InBuff FIFO memory in which the command packet is contained; once the IPBus transaction header is received by the Transactor it gets decoded in its fields setting the internal variables of the state machine (i.e. trans_type, write bit, etc...) and the state machine switches to the address phase where the IPBus slave address and sub-address are read from the FIFO memory.

Once the relevant informations of the IPBus transactions are retrieved by the transactor, it starts the IPBus bus cycle accordingly to the protocol and the transaction requested. If a write operation has to be made the Transactor asks the FIFO for one more word that contains the data that has to be written into the addressed slave register. If a read operation has been requested then upon receiving the acknowledge signal from the slave, it asserts the tx_we signal and put the read data onto the tx_data bus that is then written onto the OutBuff FIFO memory.

After the IPBus transaction has ended the Transactor returns to the header state where the transaction status header is written onto the OutBuff memory. During this final header state the Transactor tries to read another word from the InBuff FIFO memory since it does not know how many transactions have been queued in the transport layer. If the word retrieved is not a valid transaction header it asserts an error signal and returns to the idle state; if a valid transaction header is found instead, the process repeats until all the queued operations are done[16].

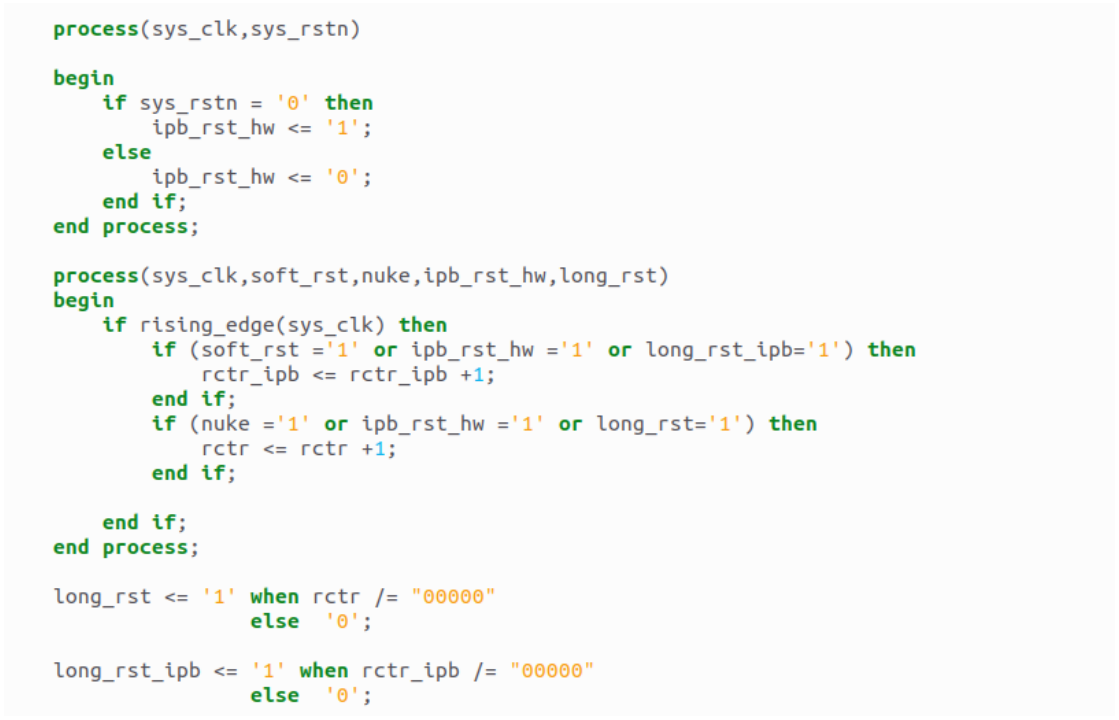
By taking a closer look to the waveform in Figure 2.11, during the first header phase at the beginning, a junk status header is written onto the OutBuff FIFO memory. This word is written only during this first phase but does not appear during the processing of the following transaction requests if more than one are queued and it gets removed at software level after the data gets retrieved by the USB Host.

2.2.4 Reset Logic

The reset logic process, whose code is shown in Figure 2.12, manages the reset signal going simultaneously to the interface blocks and to the IPBus slaves.

The IPBus side of the system can be reset in two ways: the first by pressing a button connected to the sys_rstn signal, and the second by setting to a logic 1 a certain bit of an IPBus slave register to assert the soft_rst signal. The interface is instead reset by pressing the onboard reset button or by setting another bit to 1 in the corresponding IPBus slave register.

In particular when a reset signal is received by this process, it starts one or both 5-bits reset counters, keeping the reset signals asserted until their values roll over to 0. This ensures that every component has enough time to reset properly.



```
process(sys_clk,sys_rstn)

begin
    if sys_rstn = '0' then
        ipb_rst_hw <= '1';
    else
        ipb_rst_hw <= '0';
    end if;
end process;

process(sys_clk,soft_rst,nuke,ipb_rst_hw,long_rst)
begin
    if rising_edge(sys_clk) then
        if (soft_rst ='1' or ipb_rst_hw ='1' or long_rst_ipb='1') then
            rctr_ipb <= rctr_ipb +1;
        end if;
        if (nuke ='1' or ipb_rst_hw ='1' or long_rst='1') then
            rctr <= rctr +1;
        end if;

    end if;
end process;

long_rst <= '1' when rctr /= "00000"
                else  '0';

long_rst_ipb <= '1' when rctr_ipb /= "00000"
                else  '0';
```

Figure 2.12: *VHDL code of the process which manages the reset signals.*

2.3 Software adaptation

As mentioned in Section 1.3.1 the IPBus project already comes with some pieces of software and the libraries which provide many functions to communicate with the Ethernet interface of the IPBus SoC; furthermore some other programs were already developed by the ALICE-TOF collaboration to construct the board control and DAQ software needed to manage the board operations[17].

Specifically there were three main programs that needed to be adapted:

- **PicoTOF**: A program that manages the configuration and start-up procedure of the picoTDCs.
- **PicoRead**: A program responsible for the PicoTDCs' data readout.
- **PicoLiroc**: A program that provides the interface to configure the LIROC chipsets mounted onto the two mezzanine cards connected to the PicoTDCs, which will be discussed in detail later.

Since the driver provided by the FTDI manufacturer did not permit multiple processes to be simultaneously connected to the same device a server application was developed to allow the contemporary board control and readout procedures. The resulting architecture is shown in Figure 2.13

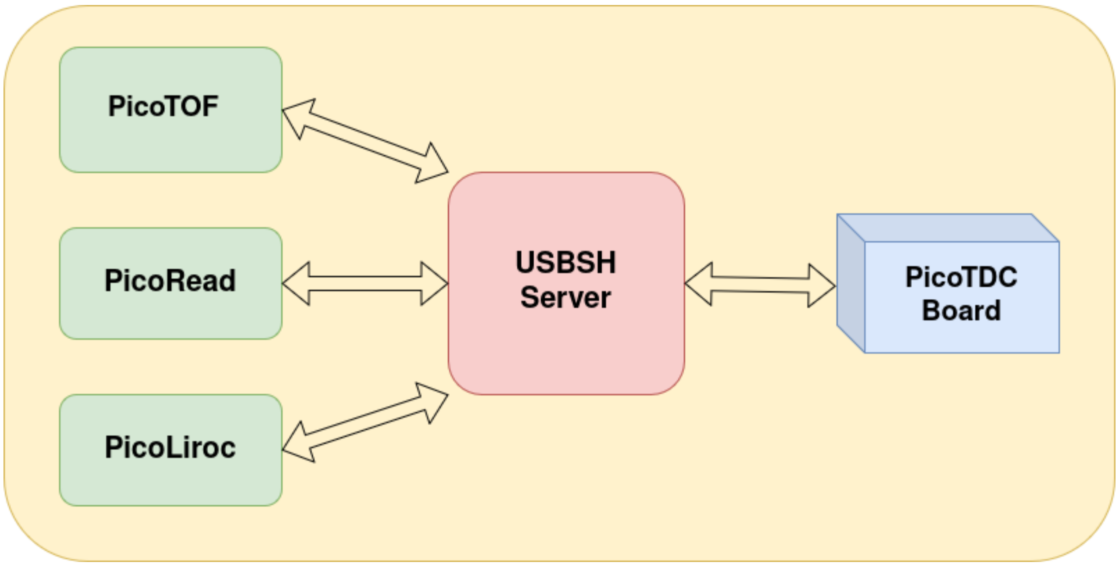
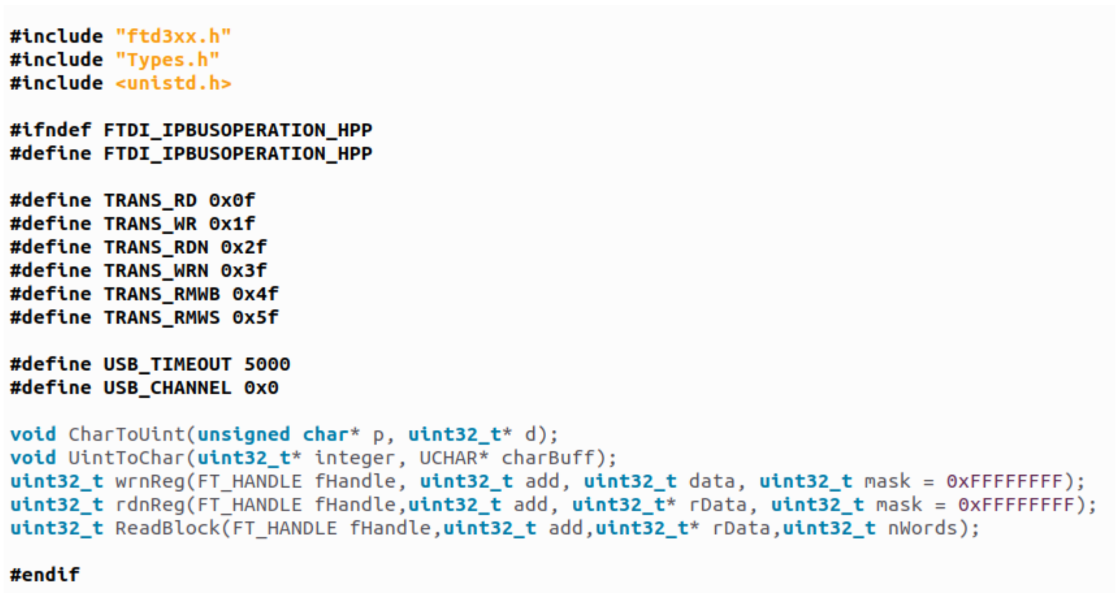


Figure 2.13: *Simplified scheme of the final system architecture implemented at software level.*

This section will describe the implementation of the basic libraries to provide communication with the board, using the API libraries provided by FTDI, and some software tools developed to overcome some limitations of this new interface.

2.3.1 The FTDI_ipbusOperation library

The first piece of software that has been developed was the FTDI_ipbusOperation library which contains the implementation of the basic functions needed to communicate with the IPBus SoC through the USB connection. In Figure 2.14 the declarations of the methods are shown.



```
#include "ftd3xx.h"
#include "Types.h"
#include <unistd.h>

#ifndef FTDI_IPBUSOPERATION_HPP
#define FTDI_IPBUSOPERATION_HPP

#define TRANS_RD 0x0f
#define TRANS_WR 0x1f
#define TRANS_RDN 0x2f
#define TRANS_WRN 0x3f
#define TRANS_RMWB 0x4f
#define TRANS_RMWS 0x5f

#define USB_TIMEOUT 5000
#define USB_CHANNEL 0x0

void CharToUint(unsigned char* p, uint32_t* d);
void UintToChar(uint32_t* integer, UCHAR* charBuff);
uint32_t wrnReg(FT_HANDLE fHandle, uint32_t add, uint32_t data, uint32_t mask = 0xFFFFFFFF);
uint32_t rdnReg(FT_HANDLE fHandle,uint32_t add, uint32_t* rData, uint32_t mask = 0xFFFFFFFF);
uint32_t ReadBlock(FT_HANDLE fHandle,uint32_t add,uint32_t* rData,uint32_t nWords);

#endif
```

Figure 2.14: *Declarations of the FTDI_ipbusOperation header methods together with the definitions of some useful constants.*

Since the IPBus works with 32-bits wide data words (i.e. uint32_t type), while the API provided by FTDI uses a 1-byte wide (i.e. unsigned char type) structure, two utility functions, CharToUint and UintToChar, were implemented to convert 4-bytes integer into an array of 4 unsigned char and viceversa providing a simple translation method between the two data structures.

The remaining three functions implement the procedure to send specific command packets to the IPBus SoC and retrieve the corresponding data from the device. The first two functions (wrnReg and rdnReg) implement the procedure to send single word non-incremental Write/Read operations at the address specified by the **add** value and return a status code indicating if an error has occurred during the operations; the ReadBlock function implements instead the procedure for multiple words non-incremental Read operation and is mainly used during the PicoTDCs' data readout. All of these three functions follows a similar procedure shown in Figure 2.15.

Each function constructs the appropriate command packet by combining an IPBus transaction header (as described in Section 1.3.4), an address and optionally

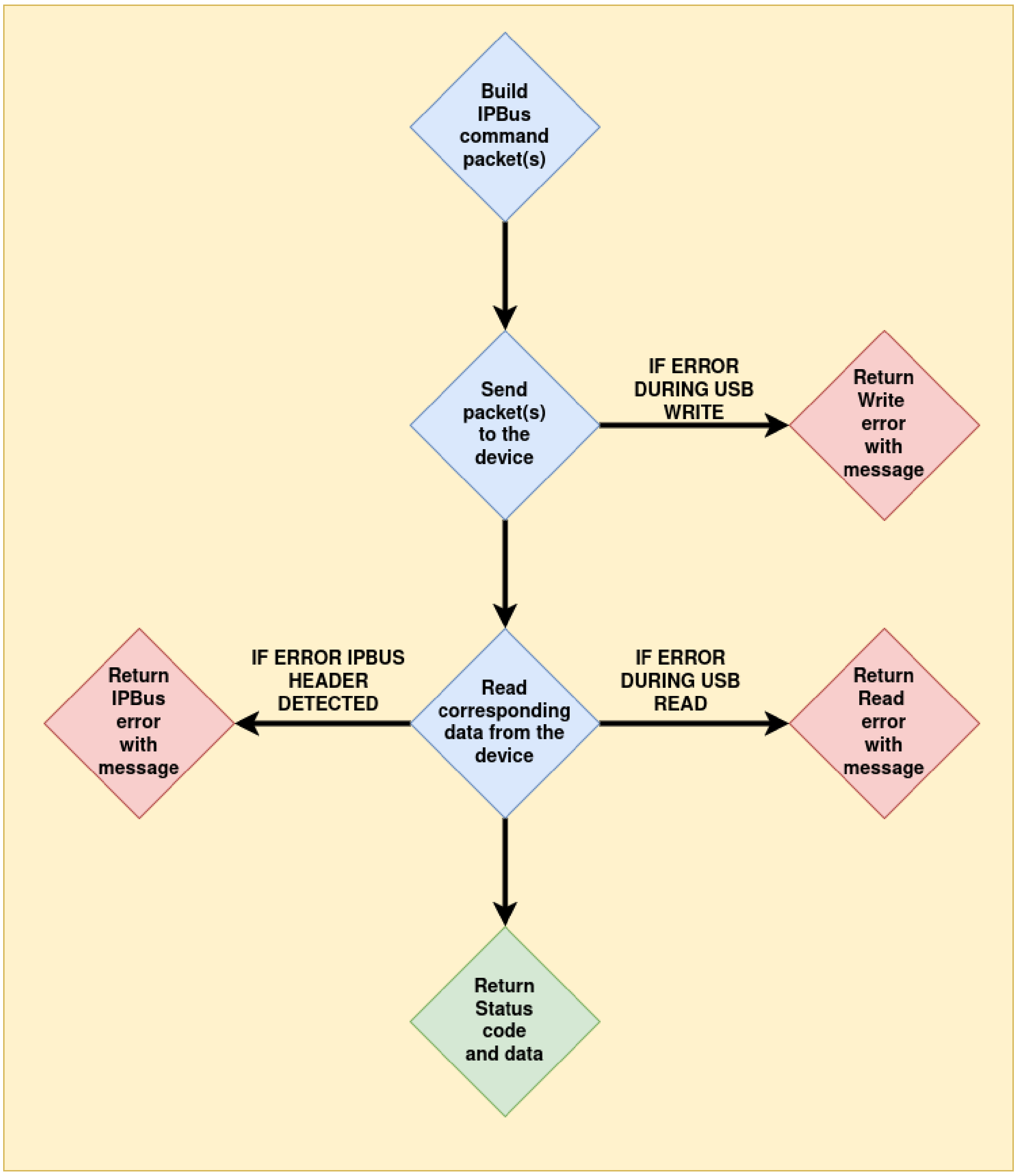

Build IPBus command packet(s)
Send packet(s) to the device
IF ERROR DURING USB WRITE
Return Write error with message
IF ERROR IPBUS HEADER DETECTED
Return IPBus error with message
Read corresponding data from the device
IF ERROR DURING USB READ
Return Read error with message
Return Status code and data

Figure 2.15: *Scheme of the functioning procedure of the basic software interface functions.*

the word to be written onto the slave register in case of a Write operation. For the ReadBlock function in particular, if the number of words to be read is greater than 255, the maximum size for a single IPBus transaction, the function builds automatically the minimum number of read commands to be sent to the device. After the command packets are built, the PC sends them using the API function FT_WritePipeEX and if an error code is returned the function exits while printing a message, otherwise proceeds to the next step. The function then asks to read the expected amount of data bytes from the device and again if an error code is found from the USB Read operation or an IPBus error header is found, the function exits with a message, otherwise the data that has been read from the IPBus slave is stored into the PC memory and the status of the IPBus transaction is returned. The full source code of these function implementation is reported in appendix A.1.

2.3.2 USBSH Server implementation

The functions described in the previous section were then used to implement a server application to manage the data I/O to and from the PicoTDC board since the driver for the FTDI FT601Q chip does not allow multiple processes to be connected simultaneously to the same device. Since the aim of this project is to have full control over the board configuration parameters even during the data acquisition phase, this was a major issue that had to addressed.

To overcome this limitation, a simple server-like application was written to provide a single process which communicates directly with the PicoTDC board and manages all the data flow between the main programs and the board itself. A detailed scheme of the system architecture is shown in Figure 2.16.

In order to provide reliable access to the board on-chip system, multiple IPC[1] tool were used:

- **Message Queue**: This interface allows Client processes to send messages to the USBSH Server application asking the server to perform various operations.
- **Shared Memory**: A memory space, divided in channels, where the data coming from the PicoTDC board is stored waiting to be read by the client process that requested the operation.
- **Named semaphores**: One for each channel plus a general one, used to make the client process wait until the requested operation is completed by the server.

In Figure 2.17 are represented the data structures implemented for the server functioning. In the messages are contained the informations about the message type, the PID of the process which sent it, the command identifier and the values needed to perform the IPBus operations. The shared memory channels contain the data regarding the PID of the client process assigned to that channel, the command identifier and the status code of the last operation requested, together

[1]Inter Process Communication

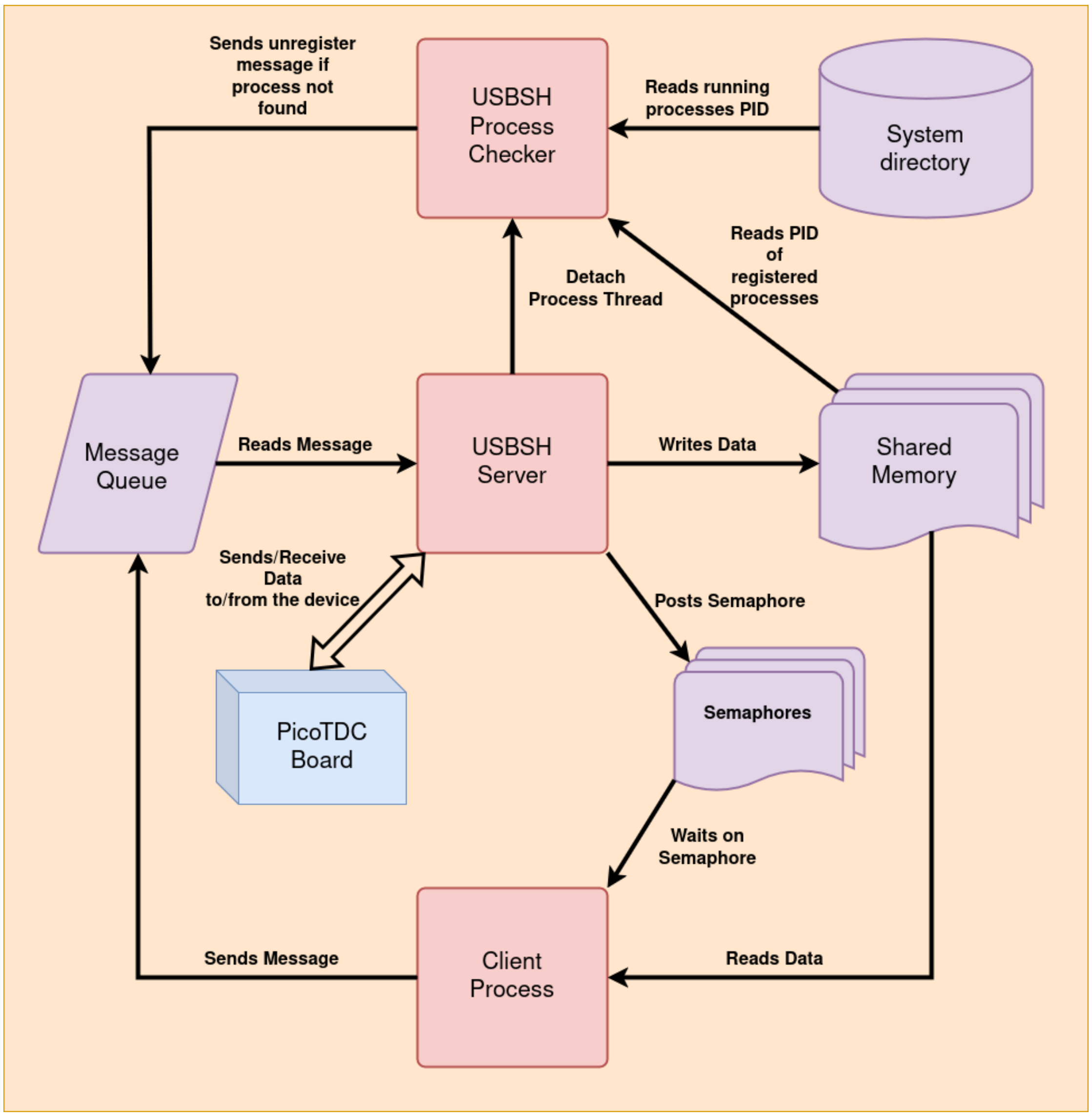


Figure 2.16: *Scheme of the server architecture implemented at software level.*

with a memory buffer wide enough to store the resulting 32-bit words coming from the PicoTDC board.

The working principle behind the server process is that, at first, a client submits a message into the message queue containing its PID[2] and the request to be registered to a shared memory channel; the client process waits an answer from the server waiting on the general semaphore. The server receives the message from the queue and registers the received PID into the first free channel of the

[2]Process ID

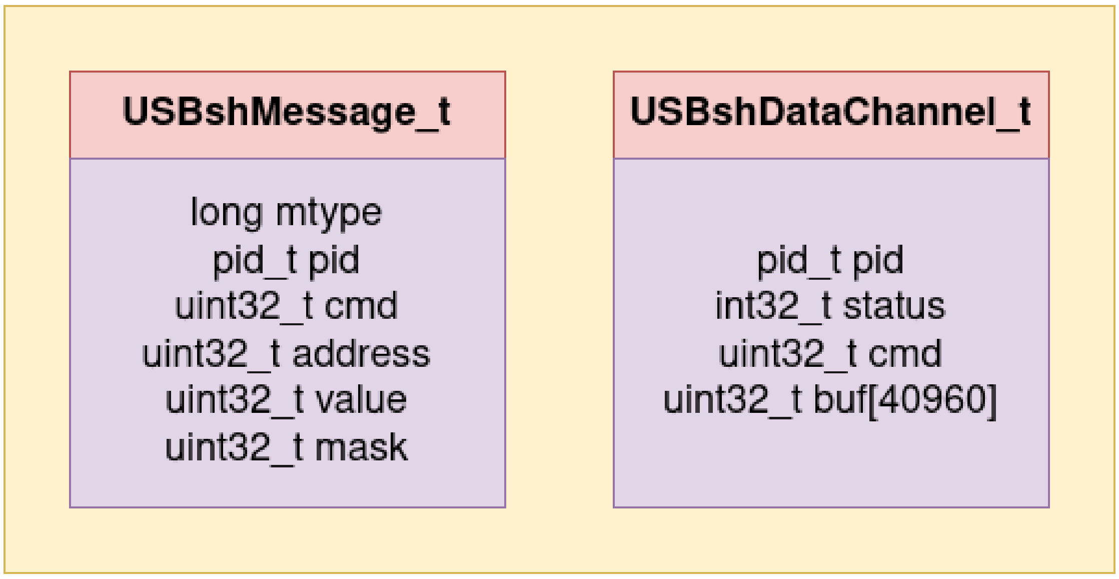


Figure 2.17: *Scheme of the implemented data structures for the message queue and the shared memory channels.*

shared memory and posts to the general semaphore to signal the process that the registration process is done; if no channel is available the server posts the general semaphore creating then an error on the client side. After the client receives the general semaphore post signal, it cycles through the shared memory channels until it finds its own PID, memorizes the channel number for further accesses at runtime and attaches to the corresponding channel semaphore.
Once this registration procedure is completed, the client is then free to send messages to the server requesting Read/Write operations that have to be made to the PicoTDC board and, in a similar way to the registration process, the server receives the messages, performs the requested operation writing the resulting data and status code onto the shared memory channel associated to the requesting process and posts onto the channel semaphore to signal the client that the operation has been done. Before the client process ends, it has to send a last message telling the server to free the channel it was assigned to, setting the channel PID to 0. A simple flowchart of the server working operation is provided in Figure 2.18.

It's now important to notice that since the server is only responsible for managing the data flow between the clients and the PicoTDC board, it does not have any error handling procedure; if an error occurred during the USB operations or on the on-chip IPBus side of the system the server simply transfers it to the requesting process that handles the exception. The only error managed by the server is during the opening of the connection to the FT601Q chip; if the connection between the server and the physical chip cannot be established, the server ends its execution.

As a failsafe measure to manage an abrupt ending of a client process that has already been registered into one of the channels, a thread is detached from the server when it starts. This thread executes a process checking function in the "/proc/" system directory.
The function cycles on the registered PIDs channel by channel and checks the system directory for the corresponding process directory; if no directory under that specific PID is found it means that the specific client ended its execution without sending the unregister message to the server, this condition triggers the process checker function to send the unregister request to the server in place of the client process that has already ended preventing the server channels saturation.

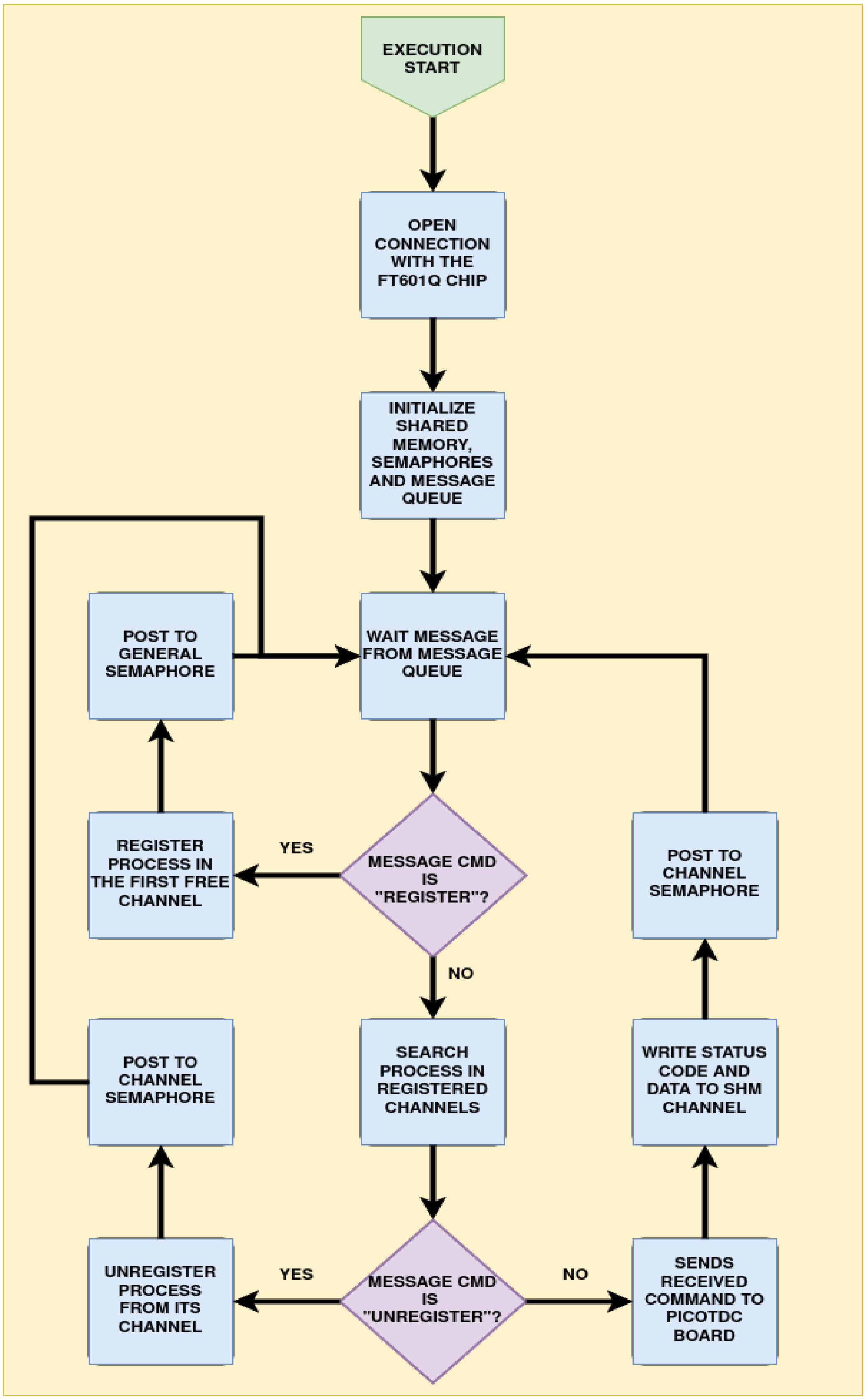

EXECUTION START
OPEN CONNECTION WITH THE FT601Q CHIP
INITIALIZE SHARED MEMORY, SEMAPHORES AND MESSAGE QUEUE
POST TO GENERAL SEMAPHORE
WAIT MESSAGE FROM MESSAGE QUEUE
REGISTER PROCESS IN THE FIRST FREE CHANNEL
YES
MESSAGE CMD IS "REGISTER"?
POST TO CHANNEL SEMAPHORE
NO
POST TO CHANNEL SEMAPHORE
SEARCH PROCESS IN REGISTERED CHANNELS
WRITE STATUS CODE AND DATA TO SHM CHANNEL
UNREGISTER PROCESS FROM ITS CHANNEL
YES
MESSAGE CMD IS "UNREGISTER"?
NO
SENDS RECEIVED COMMAND TO PICOTDC BOARD

Figure 2.18: *Flowchart of the server functioning.*

2.3.3 The libUSBsh library

After the server application was made, the libraries containing the classes and the functions used in the main programs needed to be adapted; in order to do so the method implementing the communication process from the client processes to the new USBSH server were made (full code implementation in Appendix A.2).

In this process five main functions were developed, two for the registering and unregistering of the process in the server, called FTC_Create and FTC_Close, and the other three that implement the Read/Write/ReadBlock operations requests and the data retrieval process from the system shared memory, together with dummy uhal classes and functions to keep software transparency between the two different interfaces. Furthermore, some global variables were used to store informations about the process assigned channel, the assigned semaphore and the addresses of the message queue and the shared memory.

As the name suggests, the first two functions make the client process start or end the communication with the USBSH server and make the software attach to all the necessary IPC tools. In particular the FTC_Create function first attaches the process to the shared memory, the message queue and the general semaphore before sending the register message to the server and retrieving all the information about the channel number and semaphore after the registration process has been completed by the server. If an error occured at any time, a non-null status code is returned by the function causing the client process to exit; furthermore, whenever the process attaches to a semaphore, it checks its value and in case it is non null makes the correspondent amount of sem_wait function calls to set the semaphore value to 0.
On the other hand, in a similar way, the FTC_Close function reverses this process detaching the client process from the IPC tools after the server unregister it from the assigned channel.

The FTC_WriteReg, FTC_ReadReg were substituted into the already existing classes implementation to execute equivalent operations in the three main programs using the newly implemented USB interface. In these functions, the client process asks the server to perform the corresponding operation on the board and waits for the server to store the data into the assigned shared memory channel word buffer. In the FTC_ReadBlock function, it's however important to notice that the "value" field of the message is used to give the server the information on the number of words that need to be read at the specified address.

Chapter 3

LIROC front-end card

As mentioned in Section 1.2 the PicoTDC board features two FMC connectors where the front-end electronics can be plugged to process (tipically amplify and discriminate) sensor signals and then send the output to the two PicoTDC ASICs. During this work a custom card was developed by the ALICE-TOF collaboration featuring an on-board LIROC ASIC for signal amplification and discrimination. This front-end card, shown in Figure 3.1, features, together with the LIROC ASIC, an FMC connector to plug the front-end card onto the PicoTDC Board, and various SMA[1] connectors to probe the ASIC output digital signals and to sample the ASIC analogue probe signals.

In this chapter the characteristics of this ASIC together with the firmware development to integrate the control and configuration protocols for this specific ASIC into the IPBus SoC will be discussed.

[1]SubMiniature version A

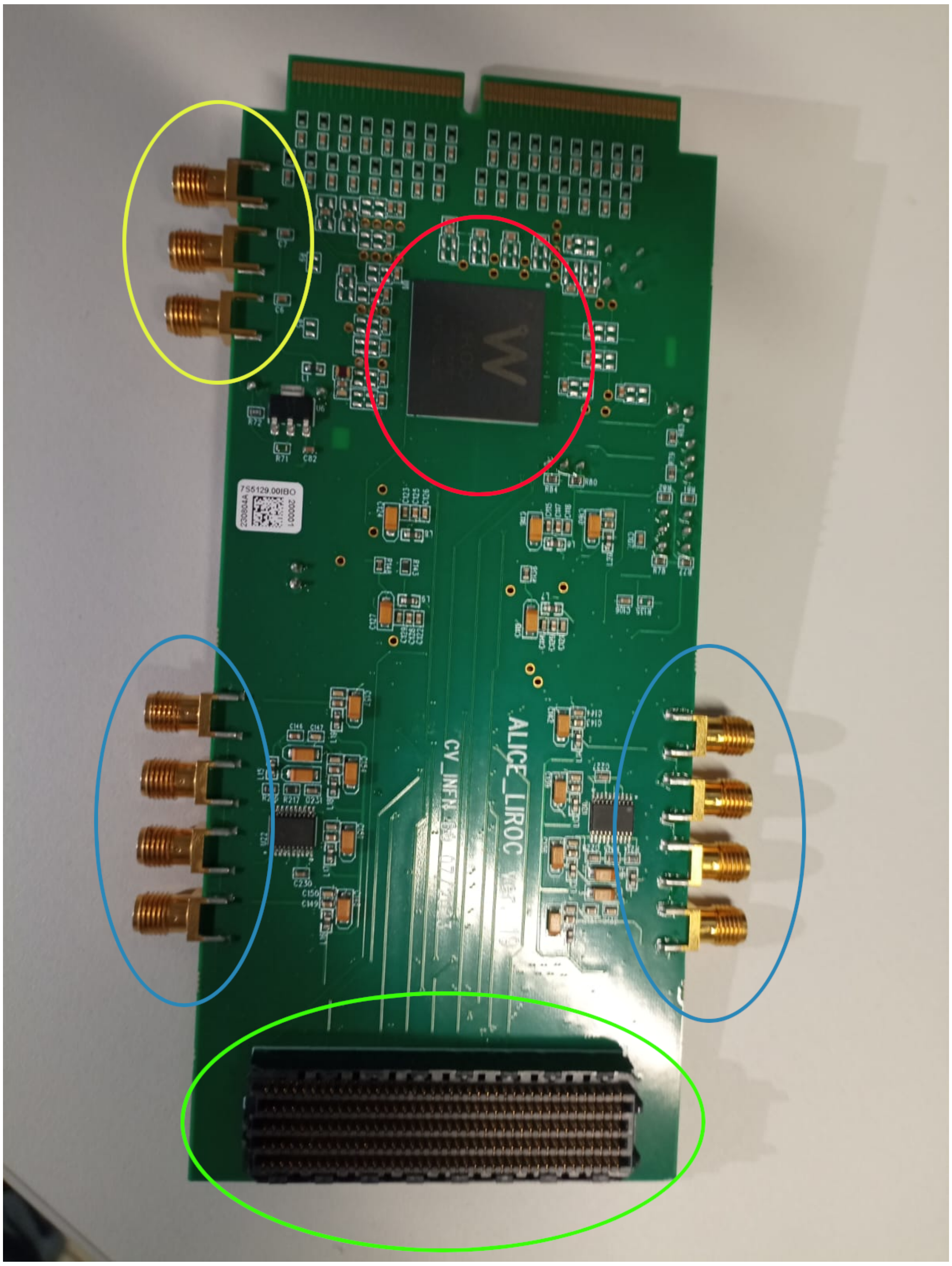


Figure 3.1: *Photo of the custom LIROC card. The components highlighted are: the LIROC ASIC (red), the FMC connector (green), Analog probe SMA connectors (yellow) and digital probe SMA connectors (blue).*

3.1 LIROC ASIC overview

LIROC is a 64-channel front-end ASIC developed by Weeroc and designed to readout SiPMs[2]; in particular, it provides low-voltage differential trigger output for each channel with good time resolution (≤ 20 ps) and excellent double-peak separation. It also allows to adjust the SiPMs high voltage by using a channel-by-channel 6-bit DAC[3] connected to the ASIC inputs together with channel-by-channel calibration of the trigger threshold via 7-bit DACs. The output signal is produced using an RF preamplifier with pole zero cancellation followed by a fast discriminator and low swing LVDS fast drivers[18]. The ASIC features also a programmable analog probe to test the input analog signals before discrimination and the threshold value of the discriminator of each channel.

The internal parameters of the ASIC can be configured via I2C using a custom protocol that does not follow the IEEE standard.

3.1.1 ASIC Architecture

As introduced before, LIROC is a fully analog 64-channel ASIC developed for SiPMs readout and provides fast trigger output signals. The ASIC block diagram is shown in Figure 3.2.

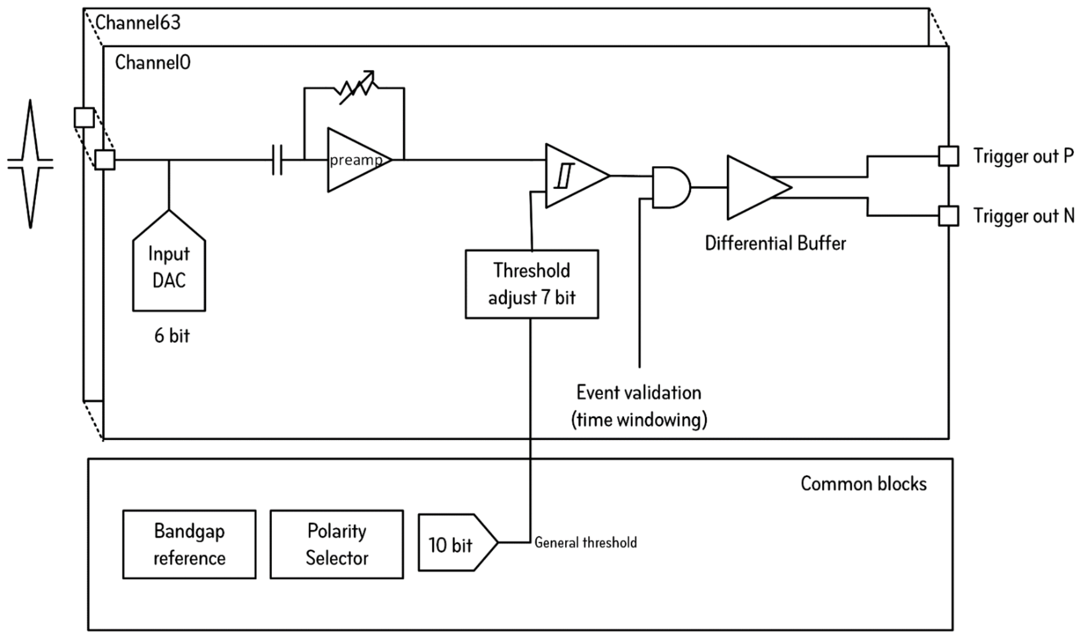


Figure 3.2: *LIROC ASIC block diagram.*

[2]Silicon Photo-Multipliers
[3]Digital to Analog Converter

The main features of the ASIC include:

- **6-bit input DAC**: individual input DAC for trimming SiPM overvoltage for detector gain corrections.
- **Adjustable preamplifier**: High bandwidth preamplifier with adjustable gain.
- **Adjustable Discriminator**: Fast discriminator with adjustable threshold through a 10-bit general threshold and 7-bit individual threshold trimming.
- **Differential outputs**: Differential buffer for low-jitter trigger outputs signals.

Additionally, the user is able to perform time windowing for the trigger output signals through an external signal together with output signal masking for each individual channel.

3.1.2 LIROC I2C configuration

The LIROC ASIC includes an I2C interface through which a set of internal registers can be configured to manage the internal working parameters of the ASIC. The internal I2C slave core can be programmed using a custom I2C protocol. In order to work properly the I2C slave core must receive a clock signal through a port named clk_sm_i2c with a frequency 20 times higher than the clock sent by the I2C master on the SCL line together with the request that these clock signals must be synchronous.

The slave I2C interface features 68 main registers with two or three 8-bit sub-registers each that are identified by an 11-bit address and a 5-bit sub-address. A full list of these registers with their description is reported in Table A.4.

Addresses from 0 to 63 correspond to the 64 input channels parameters (i.e. 7-bit threshold adjustment, 6-bit input DAC value, etc...), while addresses from 64 to 67 contains parameters working for the whole ASIC (i.e. General threshold, Polarity selection, etc...). The protocol to read and write one of these registers is performed by sending three separate I2C frames of 16 bits each as described in Figure 3.3 and 3.4.

In particular during the I2C addressing phase the master sends the 4-bit chip ID sequence, which can be set externally by providing 1.2 V voltage to the correspondent package pins, followed by a 3-bit sequence that identifies the frame number. The first two frames have the Read/Write bit set to 0, indicating a write operation, followed by the 8-bit sequence containing first the full address LSB[4] and

[4]Least Significant Byte

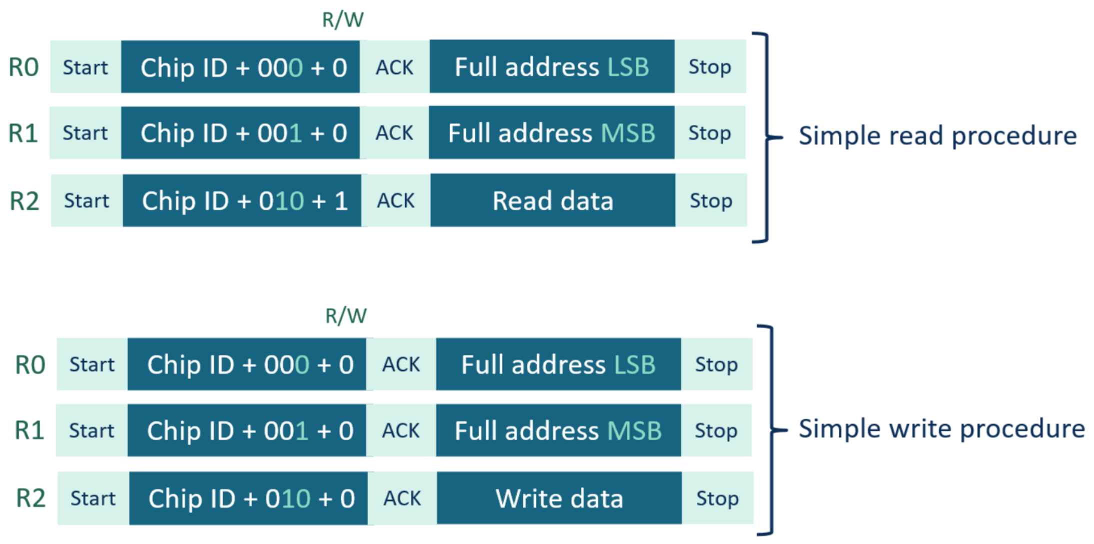

R/W
R0 Start Chip ID + 000 + 0 ACK Full address LSB Stop
R1 Start Chip ID + 001 + 0 ACK Full address MSB Stop
R2 Start Chip ID + 010 + 1 ACK Read data Stop
Simple read procedure
R/W
R0 Start Chip ID + 000 + 0 ACK Full address LSB Stop
R1 Start Chip ID + 001 + 0 ACK Full address MSB Stop
R2 Start Chip ID + 010 + 0 ACK Write data Stop
Simple write procedure

Figure 3.3: *Simple Read/Write procedures for LIROC I2C slow control and configuration.*

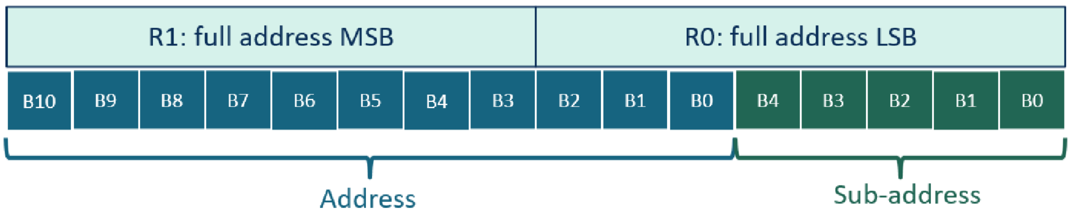


Figure 3.4: *LIROC I2C internal register addressing.*

second the full address MSB[5]. The third frame then contains the information of the intended operation.

3.1.3 LIROC analog operation

As shown in Figure 3.5, the analog part of the LIROC ASIC is composed by the dual polarity pre-amplifier followed by a discriminator for input signal discrimination and triggering. The threshold of the discriminator is set via a 10-bit DAC that sets a threshold value for the whole ASIC and that can be further adjusted via a 7-bit trimmer available at channel level.

The pre-amplifier used in the ASIC is a transimpedance amplifier that converts input current pulses into voltage signals through a resistor.

[5]Most Significant Byte

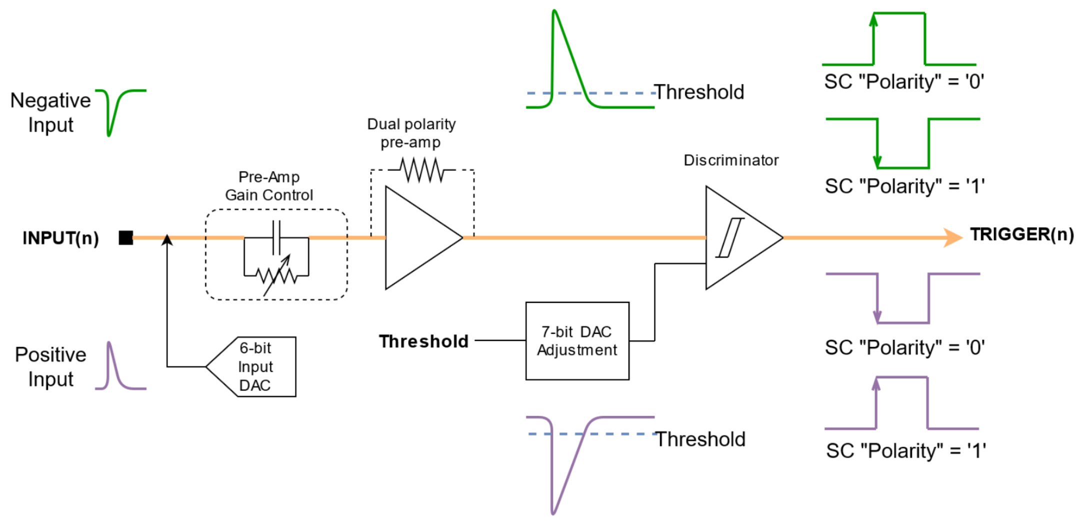


Figure 3.5: *LIROC analog section block diagram. The behaviour for both output polarity is shown for the different input polarities.*

Additionally, there is a 6-bit input DAC used for adjusting the SiPM voltage non-uniformity for each channel. This value is accessible through Slow Control bits 7 to 2 at address 0-63 and sub-address 0. The values of the input DAC varies from 395 mV to 672 mV with an incrementing step of 4.4 mV.

To set the pre-amplifier pole zero cancellation, which is done through an RC network with a fixed capacitor of 5 pF, the resistor value can be changed via bits 5 to 2 of address 64 and sub-address 0; these for 4 bits are associated to four different resistor values of 16 kΩ, 8 kΩ, 4 kΩ, 2 kΩ going from the most significant bit to the least significant one. The resulting resistor value is the parallel combination of the enabled resistors.

The threshold value for the discriminator is provided first by the 10-bit general threshold that can be set by modifying bits 1 to 0 at address 65 and sub-address 1 that represents bit 9 and 8 of the general threshold DAC together with bits 7 to 0 at sub-address 2 and same address. The value of the general threshold ranges between 374.8 mV and 814.8 mV with an incrementing step of 0.43 mV.
Additionally, the threshold can be fine-tuned for each channel through the 7-bit trimmer. The combination of the general threshold and the trimming value provides the effective trigger threshold for the discriminator. The value of the trimmer is accessible via bits 6 to 0 at address 0-63 and sub-address 1 with values ranging from 0 mV to -152.4 mV with an incrementing step of -12 mV.

Looking now at the discriminator, the one embedded in the system is a 3-stage discriminator designed for fast output response. A few settings for the discriminator are available for the user. The first one is the discriminator output polarity that can be set through bit 5 at address 64 and sub-address 2 to either 0, meaning positive polarity for negative input signals, or 1, meaning positive polarity for positive input signals. The second is the channel output mask, which can be set through bit 7 at address 0-63 and sub-address 1 for each channel; this bit enables/disables the channel output signal.

3.1.4 LIROC analog probe

The LIROC ASIC is furthermore equipped with two analog probing points for the pre-amplifier signal and the threshold voltage value. These two probing points are connected on the custom card to two SMA connectors that can be used to monitor these signals with an oscilloscope. These probing points are accessible through a 128-bit shift register that, depending on the position of a logic 1 bit, enables one of the 128 probing points available.

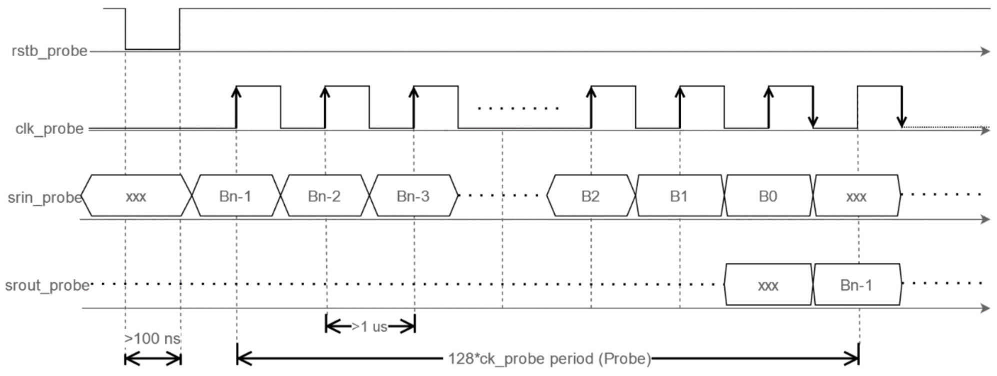


Figure 3.6: *LIROC analog probe setting procedure.*

The procedure to enable one of the probing point is shown in Figure 3.6; the first step is to reset the shift register by asserting a reset signal for at least 100 ns, then the 128 bits must be inserted in the shift register providing a clock signal and driving the srin_probe port to the desired values in the correct order. The input data of the shift register is sampled on the rising edge of the clock signal, while the output data is toggled on the falling edge. After 128 clock cycles the shift register is completely full, enabling the desired probing point.

The data contained in the shift register can be readout through srout_probe port after that the shift register has been filled by continuing to provide the clock signals for another 128 cycles. To check that the desired bit is set to 1, one should provide the input sequence two times in order to set the shift register the first time, and read out its content while filling it again with the same values.

3.2 Firmware implementation for LIROC operations

During the preparation of the new USB interface for the PicoTDC board, some firmware modifications were developed in order to provide all necessary components for the LIROC Slow Control and Analog Probe settings through the FMC connector.

In particular the IPBus I2C Master was integrated with additional registers to provide the necessary signals for the LIROC I2C slave core to work and a new IPBus slave was developed to manage the setting of the LIROC analog probes.

Furthermore, a 6-bit clock divider, which divides the IPBus 40 MHz clock, was added at the firmware top level to provide the LIROC I2C slave core with a clock of frequency lower than 1 MHz as required by the manufacturer specifications.

3.2.1 LIROC I2C Master

The IPBus I2C Master is already provided by CERN and implements an FSM that manages the I2C protocol. This module connects directly to the SDA and SCL lines connected to the LIROC I2C slave core through the FMC connector and its structure is shown in Figure 3.7.

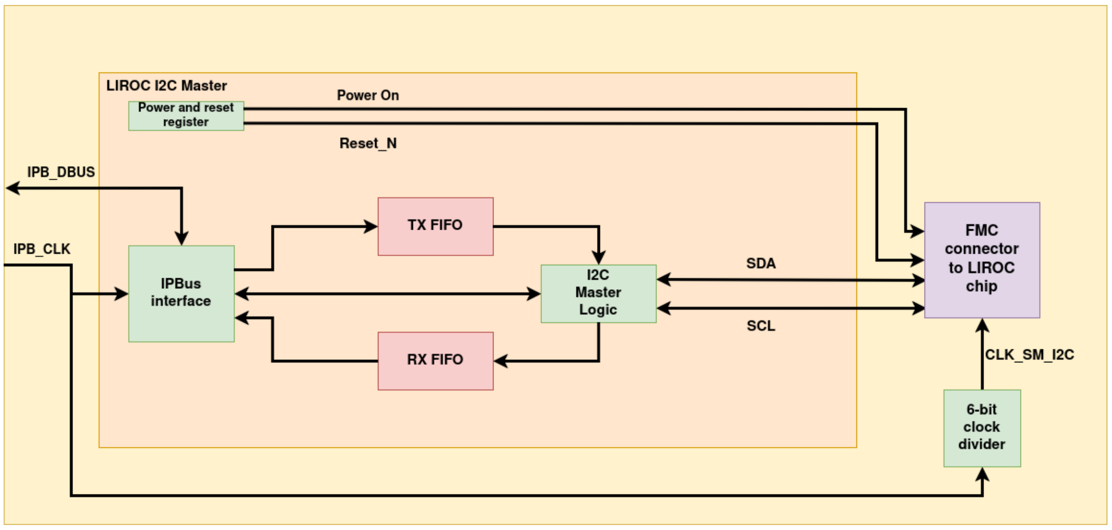


Figure 3.7: *LIROC I2C master block diagram with all necessary signals for the LIROC I2C Slow Control.*

Starting from the left, the IPBus Interface module provides the necessary procedures for the IPBus communication protocol to control a set of internal registers containing some parameters needed to drive the I2C Master Logic FSM. This submodule contains also two single clock FIFOs to store data that has been read (RX FIFO) or will be written (TX FIFO) through the I2C protocol by the I2C Master Logic FSM. The IPBus interface uses its internal registers, shown in Table 3.1, to build the I2C transactions by using IPBus software commands. In particular for the LIROC master one more register was added (register address 0x7) to control the reset and power-on signals needed by the LIROC I2C slave core.

To perform a general I2C transaction, the I2C Master Logic state machine needs some parameters to be defined. First of all, by writing the prescaler register through an IPBus write transaction, the SCL pulse length and the data_setup time need to be set. In particular the SCL pulse length (bits 15 to 0) is defined as the number of clock cycles of the IPBus clock for both state 0 and state 1 of the SCL line, while the data_setup time (bits 31 to 16) defines the number of clock cycles of the IPBus clock after which the SDA line commutes when the SCL is in the 0 state; for practicality the data setup time is set to half the SCL pulse length value. After the prescaler register is set, it's compulsory to set the device_addr register to the I2C slave address; this value is used then by the I2C Master Logic state machine during the addressing phase of the I2C protocol to address the desired slave. When this preliminary steps are done, the state machine has all the needed parameters to start an I2C transaction; to trigger an I2C write the wr register is written through an IPBus transaction causing the state machine to write a single 8-bit word that has been previously written into the TX FIFO through the IPBus, while to trigger an I2C read transaction the rd register is written with the number of 8-bit words that need to read from the I2C slave, these words are then written into the RX FIFO memory ready to be read through an IPBus transaction.

Thanks to the generality of this state machine, it was possible to implement at software level the I2C protocol needed by the LIROC I2C slave core represented in Figure 3.3, since it is composed by three separated I2C frames of the proper kind with different I2C addresses during the addressing phase.

name	**Register_addr**	**bits [31:0]**	**function**
prescaler	0x0	[31:16] - [15:0]	Sets the length of the SCL pulse and the data_setup, using the IPbus clock cycle as step unit.
device_addr	0x1	[6:0]	Sets the slave 7-bit address used in the transactions.
rd	0x2	[8:0]	Triggers an I2C read cycle providing the total number of desired bytes.
wr	0x3	[31:0]	Triggers an I2C write cycle.
wr_data	0x4	[7:0]	Writes a byte inside the Tx_FIFO that will be used for an I2C write.
rd_data	0x5	[7:0]	Reads a byte inside the Rx_FIFO, using an IPbus read.
status	0x6	[3:0]	These are status and control outputs signals for the I2C_master_interface.
pwr_rst	0x7	[1:0]	Sets LIROC Power_On signal [1] and LIROC reset signal [0].

Table 3.1: List of the internal registers of the IPbus_interface module of the LIROC I2C Master module.

3.2.2 LIROC Analog Probe Setup

During the board operation, the possibility to monitor the analog signals internal to the LIROC ASIC was useful to check its functioning. In order to program the shift register, an ad-hoc IPBus slave was developed to provide the user with this feature.

The basic idea behind this new module, whose block diagram is shown in Figure 3.8, is similar to the one of the I2C Master; two main components were needed: one providing an interface for the IPbus protocol and the other implementing the procedure for the shift register programming using the information received from the IPBus. Using these two components, the new module called LIROC Analog Setup was implemented to manage the procedure for both LIROC chip connected through the two FMC connectors.

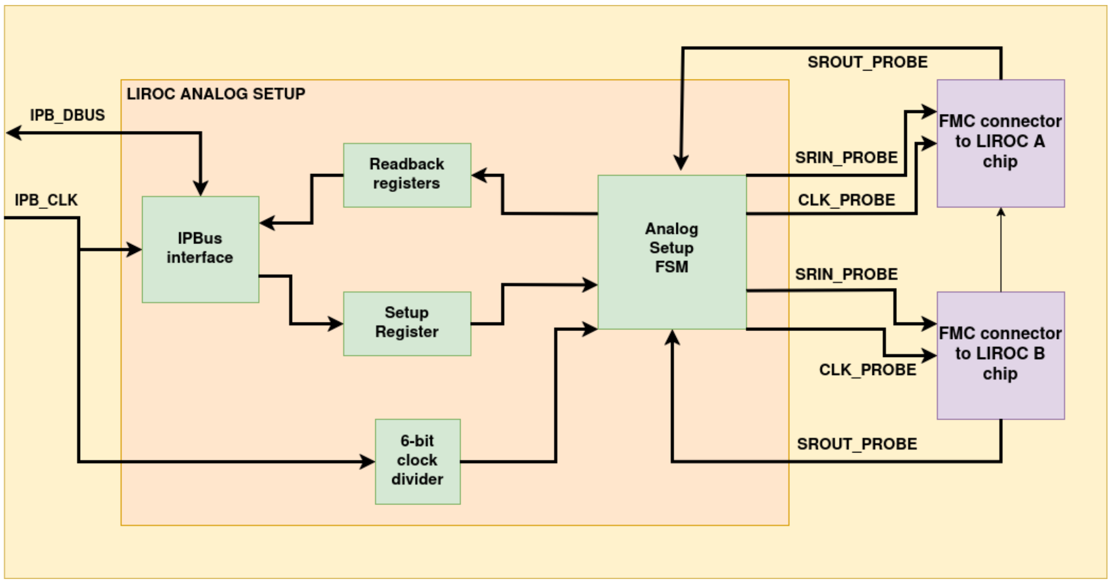


Figure 3.8: *Block diagram of the LIROC Analog Setup slave with all necessary signals for the LIROC Analog Probe programming.*

As for the I2C Master, the IPBus interface provides the logic for the handling of the IPBus transactions together with three total registers, one input register where the information for the programming of analog probe is written and two output ones where the shift registers readback data are stored to keep track of their current values; the register structure is shown in detail in Table 3.2. The information of the bit that has to be set to 1 in the shift register is encoded into two 7-bit sections of the first register, one for each LIROC, together with two start bits that signal to the state machine to start the analog probe programming procedure for the correspondent LIROC.

name	Register_addr	bits	function
LIROCAbit	0x0	[6:0]	Position of the 1 bit for the LIROC A shift register.
LIROCBbit	0x0	[14:8]	Position of the 1 bit for the LIROC B shift register.
LIROCAstart	0x0	[30]	Start bit to trigger Analog Setup FSM to start the procedure for the LIROC A.
LIROCBstart	0x0	[31]	Start bit to trigger Analog Setup FSM to start the procedure for the LIROC B.
LIROCAreadback	0x1	[6:0]	LIROC A shift register readback value.
LIROCBreadback	0x2	[6:0]	LIROC B shift register readback value.

Table 3.2: List of the internal registers of the IPbus_interface module of the LIROC Analog Setup module.

The state machine takes the data from the input register to get the position of the bit that needs to be set to 1 in the shift register and if the correspondent start bit is asserted the procedure starts. As from the ASIC specifications, the clock fed as input to the shift register must have a frequency $\leq$ 1MHz, so a 6-bit clock divider was implemented to reduce the frequency of the 40 MHz IPBus clock before feeding it to the LIROC ASIC.

When the conditions for the start of the procedure are met, the state machine starts by asserting the shift register reset signal for two clock cycles; after that, contemporarily to the reset deassertion, it enables the output clock signal of the LIROC specified by the start bits for the next 256 clock cycles necessary to complete the programming and readback procedure. A counter keep track of the bit number the state machine is providing to the shift register and sets the SRIN_PROBE signal to the correct value as specified by the register value. An example of this first stage of the procedure is shown in Figure 3.9.
After the bit sequence is inserted in the shift register the first time (128 clock cycles), the readback procedure starts to retrieve the bit position through the SROUT_PROBE line and write it into the readback register; in particular the value of a counter is written into the readback register when a 1 bit is received. Due to the nature of shift registers the input sequence must be repeated during the readback procedure to keep the register set to the same values.

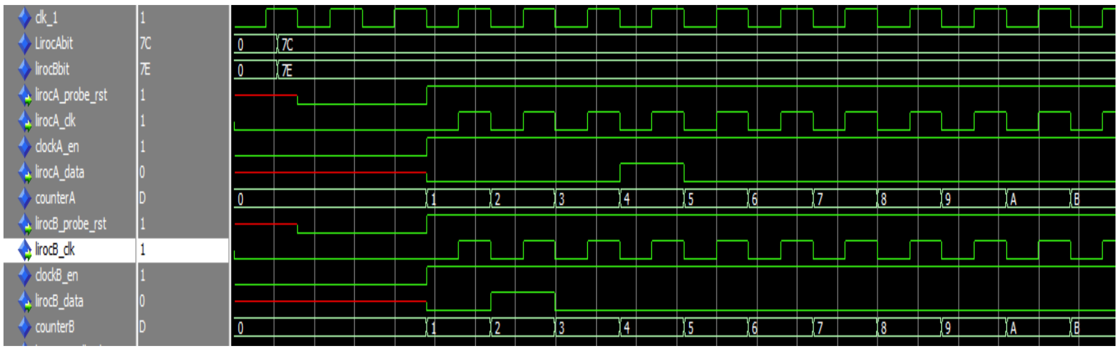


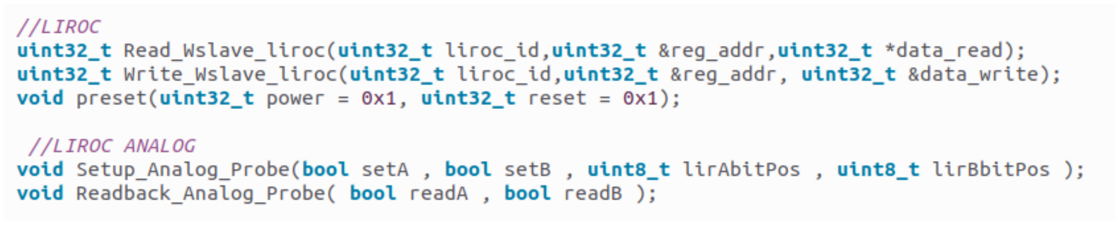
Figure 3.9: *Waveform simulation of the first stage of the LIROC Analog Probe programming procedure for both LIROC A and LIROC B.*

3.3 Software adaptation for LIROC configuration

As shown in Section 2.3, the program called PicoLiroc is responsible for the configuration and control of the LIROC ASICs on the mezzanine cards. During the development of this software, the libraries for the software control of the slaves I2C Master and Analog Probe Setup modules of the FPGA were modified to implement the required operations.

3.3.1 I2C library integration for LIROC

Since the I2C slave core mounted into the LIROC ASIC does not follow the standard IEEE I2C protocol, a series of methods of the I2C library were created to implement this custom protocol. Furthermore, the functions for the Analog Probe settings were also integrated in this library to keep all the software together into a single class. The method declarations are shown in Figure 3.10 and the full implementation is reported in Appendix A.3.



```
//LIROC
uint32_t Read_Wslave_liroc(uint32_t liroc_id,uint32_t &reg_addr,uint32_t *data_read);
uint32_t Write_Wslave_liroc(uint32_t liroc_id,uint32_t &reg_addr, uint32_t &data_write);
void preset(uint32_t power = 0x1, uint32_t reset = 0x1);

 //LIROC ANALOG
void Setup_Analog_Probe(bool setA , bool setB , uint8_t lirAbitPos , uint8_t lirBbitPos );
void Readback_Analog_Probe( bool readA , bool readB );
```

Figure 3.10: *Method declarations of the software integrated into the I2C library.*

The first method that needs to be considered is the preset function; this function takes as input the values of the LIROC power_on and reset signals that are contained in the register at address 0x7 of the LIROC I2C Master module. In order to get the LIROC I2C slave core to work the power_on signal must be set to a logic 1 together with the reset bit; if one whishes instead to reset the LIROC I2C slave core internal registers to the default values the power_on value must be set to 1 while the reset is set to 0 since it works in negative logic.

The implementation for the LIROC I2C read and write operations are programmed to exploit the flexibility of the I2C Master Logic module to implement the correct I2C protocol required. To reproduce the scheme of the protocol given in Figures 3.3 and 3.4 the functions must follow a precise series of steps.

The Read_Wslave_liroc function manages the procedure to read the content of a register of the LIROC I2C slave core starting by writing the LIROC chip ID, which is known and set by 4 jumpers on the card, followed by the first 3-bit sequence into the device address register of the I2C Master module; then the least significant byte of the 16-bit address of the LIROC I2C slave core register, which is computed at runtime by the LADD(a,sa) function, is written into the TX FIFO memory and an I2C write is triggered by writing the correspondent register. This same procedure is then repeated for the second I2C frame of the read protocol using the second 3-bit sequence for the device address and the most significant byte of the LIROC slave core register address. In the last step then, the device address is modified by using the last 3-bit sequence and a read operation is triggered by writing the rd register of the I2C Master module; after that the status of the I2C Master is checked and if no error occurred the data word retrieved from the RX FIFO memory by reading the correspondent IPBus register.

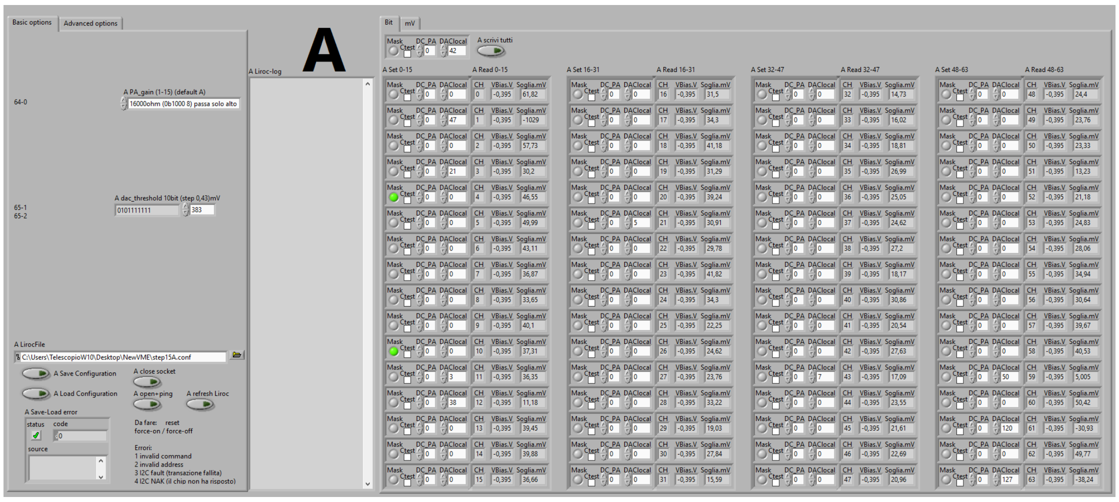
To write one of the LIROC I2C slave core registers the Write_Wslave_liroc function is called instead. The procedure to perform this operation is almost the same as the one illustrated for the Read_Wslave_liroc function; the only difference stands into the last step where after that the device address register is written with the last 7-bit sequence, the data word that will be written into the LIROC register is written into the TX FIFO memory and then a write operation is triggered; after that the status of the I2C Master module is checked to ensure no I2C errors occurred.

In order to control the newly developed LIROC Analog Setup IPBus slave two new mehods were implemented.
To set the Analog Probe shift register the Setup_Analog_Probe is called and it takes in input the desired LIROC that need to be set through the two boolean variables and their respective bit number that has to be set to 1. If a boolean value is TRUE then the corresponding bit position is checked to control if it stays in the correct range of values; if the values are accepted, the information of the bit position is added, together to the corresponding start bit, into the corresponding bits of the 32-bit word that will be written into the IPBus slave register; in the other case the corresponding start bit and the bit position field are set to 0. This process is performed for either one or both LIROCs at the same time since the LIROC Analog Setup slaves can perform this operation for both contemporarily.

To read the current bit position of the two LIROC shift registers the Readback_Analog_Probe function reads the specified values from the corresponding IPBus registers. The user can read one or both register by specifying through the boolean input values which registers are to be read.

3.3.2 LIROC server implementation

The methods explained in the previous section were used to implement a server process to provide remote control for the LIROC operations. This program, called lirocsrv, provides a standard socket TCP/IP interface through a port to allow the reception of remote commands from another PC that uses a software developed in LabVIEW. A screenshot of the LabVIEW interface for the LIROC control is shown in Figure 3.11.



64-0

A PA_gain (1-15) (default A): 16000ohm (0b1000 8) passa solo alto

65-1
65-2

A dac_threshold 10bit (step 0,43)mV: 0101111111 | 383

A LirocFile: C:\Users\TelescopioW10\Desktop\NewVME\step15A.conf

A Save-Load error
status ✓ code 0
source

Da fare: reset
force-on / force-off

Errori:
1 invalid command
2 invalid address
3 I2C fault (transazione fallita)
4 I2C NAK (il chip non ha risposto)

Mask ○ Ctest ☐ DC_PA 0 DAClocal 42

A Set 0-15 / A Read 0-15

Mask	Ctest	DC_PA	DAClocal	CH	VBias.V	Soglia.mV
○	☐	0	0	0	-0,395	61,82
○	☐	0	47	1	-0,395	-1029
○	☐	0	0	2	-0,395	57,73
○	☐	0	21	3	-0,395	30,2
●	☐	0	0	4	-0,395	46,55
○	☐	0	0	5	-0,395	49,99
○	☐	0	0	6	-0,395	43,11
○	☐	0	0	7	-0,395	36,87
○	☐	0	0	8	-0,395	33,65
○	☐	0	0	9	-0,395	40,1
●	☐	0	0	10	-0,395	37,31
○	☐	0	3	11	-0,395	36,35
○	☐	0	38	12	-0,395	11,18
○	☐	0	0	13	-0,395	39,45
○	☐	0	0	14	-0,395	39,88
○	☐	0	0	15	-0,395	36,66

A Set 16-31 / A Read 16-31

Mask	Ctest	DC_PA	DAClocal	CH	VBias.V	Soglia.mV
○	☐	0	0	16	-0,395	31,5
○	☐	0	0	17	-0,395	34,3
○	☐	0	0	18	-0,395	41,18
○	☐	0	0	19	-0,395	31,29
○	☐	0	0	20	-0,395	39,24
○	☐	0	5	21	-0,395	30,91
○	☐	0	0	22	-0,395	29,78
○	☐	0	0	23	-0,395	41,82
○	☐	0	0	24	-0,395	34,3
○	☐	0	0	25	-0,395	22,25
○	☐	0	0	26	-0,395	24,62
○	☐	0	0	27	-0,395	23,76
○	☐	0	0	28	-0,395	33,22
○	☐	0	0	29	-0,395	19,03
○	☐	0	0	30	-0,395	27,84
○	☐	0	0	31	-0,395	15,59

A Set 32-47 / A Read 32-47

Mask	Ctest	DC_PA	DAClocal	CH	VBias.V	Soglia.mV
○	☐	0	0	32	-0,395	14,73
○	☐	0	0	33	-0,395	16,02
○	☐	0	0	34	-0,395	18,81
○	☐	0	0	35	-0,395	26,99
○	☐	0	0	36	-0,395	25,05
○	☐	0	0	37	-0,395	24,62
○	☐	0	0	38	-0,395	27,2
○	☐	0	0	39	-0,395	18,17
○	☐	0	0	40	-0,395	30,86
○	☐	0	0	41	-0,395	20,54
○	☐	0	0	42	-0,395	27,63
○	☐	0	7	43	-0,395	17,09
○	☐	0	0	44	-0,395	23,55
○	☐	0	0	45	-0,395	21,61
○	☐	0	0	46	-0,395	22,69
○	☐	0	0	47	-0,395	20,96

A Set 48-63 / A Read 48-63

Mask	Ctest	DC_PA	DAClocal	CH	VBias.V	Soglia.mV
○	☐	0	0	48	-0,395	24,4
○	☐	0	0	49	-0,395	23,76
○	☐	0	0	50	-0,395	23,33
○	☐	0	0	51	-0,395	13,23
○	☐	0	0	52	-0,395	21,18
○	☐	0	0	53	-0,395	24,83
○	☐	0	0	54	-0,395	28,06
○	☐	0	0	55	-0,395	34,94
○	☐	0	0	56	-0,395	30,64
○	☐	0	0	57	-0,395	39,67
○	☐	0	0	58	-0,395	40,53
○	☐	0	50	59	-0,395	5,005
○	☐	0	0	60	-0,395	50,42
○	☐	0	120	61	-0,395	-30,93
○	☐	0	0	62	-0,395	49,77
○	☐	0	127	63	-0,395	-38,24

Figure 3.11: *Screenshot of the LabVIEW interface for the LIROC configuration.*

At the start of the program execution the lirocsrv process connects to the USBSH server and configure the selected IPBus I2C master with the proper prescaler parameters and start the LIROC initialization procedure. This procedure starts by asserting the power on signal together with the reset signal, then the reset is deasserted and a series of write commands contained in a configuration file are sent to the board to configure the LIROC I2C slave core register with a custom default configuration. If an error occurs at this stage the process is terminated with an error code.

After the LIROC initialization procedure has been completed correctly, the lirocsrv disconnects from the USBSH server and starts listening onto the assigned port. At this point when a client connects to the port the process forks and the child process is the one responsible for the interpretation and execution of the received commands and reports the operation status code to the requesting client.

To inform the client user of the operation status the lirocsrv sends back a string of the form ”LSTS address sub-address value status”, i.e. ”LSTS 31 1 34 0” meaning that a read/write operation has been done at address 31, sub-address 1 and value 0x34 was correctly read/written (status 0).

After a successful connection of a client, the child process establishes connection with the USBSH server and starts waiting for a command to be received from the client. Upon reception the command gets interpreted and executed by the lirocsrv with the data provided by the client; if an error occurs during the command reception or an invalid command is received the lirocsrv process sends an error code to the client and closes automatically the connection with the client before terminating. If a valid command is received, the server checks the validity of the values provided by the client and if they fall into the correct range the correspondent operation is executed, if the values are incorrect or an error occurs during the board operation an error message is sent to the client together with the appropriate error code. A list with the implemented commands, parameters and their description is reported in Table 3.3.

Command	Parameters	Description
save	**string** filename	Retrieve the current LIROC configuration and writes it into a file specified by the value of filename.
init	**string** filename	Initialize the LIROC configuration with the data contained in the file specified by the value of filename.
ping	None	Sends to the client a message confirming that the connection is established.
quit	None	Close the connection to the USBSH server, disconnects the client from the lirocsrv and terminates the lirocsrv child process.
reset	None	Reset the LIROC registers to the default values.
on	None	Assert the LIROC Power_On signal.
off	None	Deassert the LIROC Power_On signal.
rreg	**int** addr, **int** sub_addr	Reads the register at the address specified by addr and sub-address sub_addr then returns the read value to the client in the form "LSTS addr sub_addr val status".
wreg	**int** addr, **int** sub_addr, **uint32_t** val	Writes the value specified by val in the register at the address specified by addr and sub-address sub_addr then returns operation status to the client in the form "LSTS addr sub_addr val status".
probe	**uint8_t** bitPos	Set to 1 the LIROC Analog Probe shift register bit specified by the value of bitPos and returns the operation status to the client in the form "LSTS 0 0 bitPos status".

Table 3.3: *List of the implemented lirocsrv commands with the necessary parameters and description.*

Chapter 4

Test beam data analysis

This chapter will report about a beam test ran at CERN T10 in June 2024 to evaluate the timing performance of prototypes of Low Gain Avalanche Detectors (LGADs) and Silicon Photo-Multipliers (SiPM) in the detection of charged particles. The discrimination of both LGAD and SiPM signals was performed by a LIROC card and the digitization by the PicoTDC board, which was fully integrated in the data acquisition chain. The beam-test setup as well as the results, for a subset of the LGAD sensors, are reported in the following sections.

4.1 Experimental setup

The study of the LGAD-LIROC-PicoTDC DAQ chain perfomance was done at CERN T10 in June 2024 with a particle beam composed mainly by pions and protons with a momentum of $10\,\mathrm{GeV/c}$ and a beam intensity of $\approx 2 \times 10^6$ particles per spill over an area of few cm^2 and a spill duration of $\approx$ 400 ms.

The experimental setup used was composed of five independently movable planes on which the sensors could be mounted and aligned; a scheme of the telescope is shown in Figure 4.1. The movement of the planes was performed through two independent micropositioners in the x and y directions (those perpendicular to the beam axis) for each plane with a precision of $\approx$10 µm. These positioners can be driven remotely and were used during the alignment phase of the sensors in order to maximize the number of output signals. Each plane was furthermore equipped with a cooling system, composed of Peltier cells, to maintain the detectors' temperature constant around 20° C.

The sensors were then mounted into the planes and connected, through two extension cables, to both the LIROC cards and the PicoTDC Board. In particular an LGAD detector with a sensitive area of $1\,\mathrm{mm}^2$, as the other LGADs used, was placed on the nearest plane to the beam entrance point (Plane 0 in Figure 4.1) and chosen to act as trigger for the PicoTDCs, thus defining the active area of the particle beam.

The planes are then enclosed in a black box to prevent external light to hit the

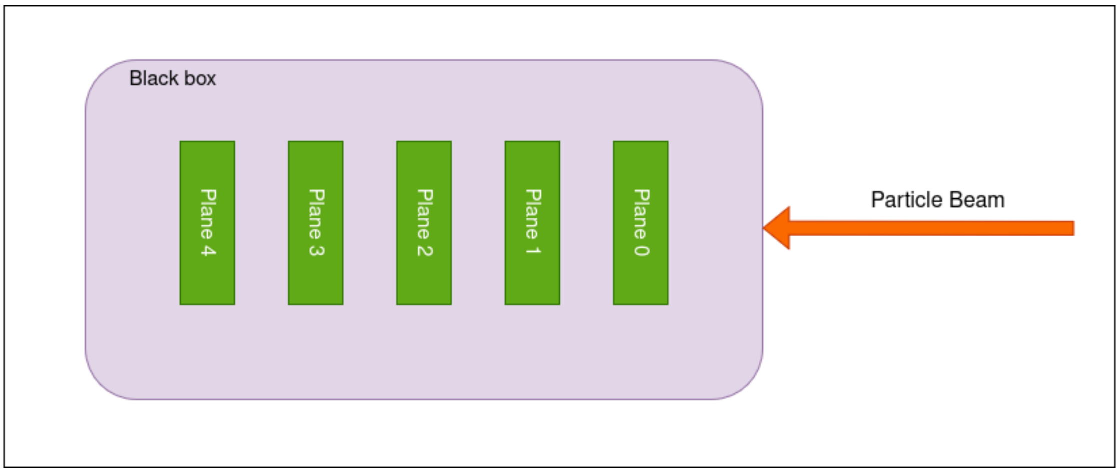


Figure 4.1: *Scheme of the telescope used during the test beam.*

detectors and to shield the telescope from the environmental noise.

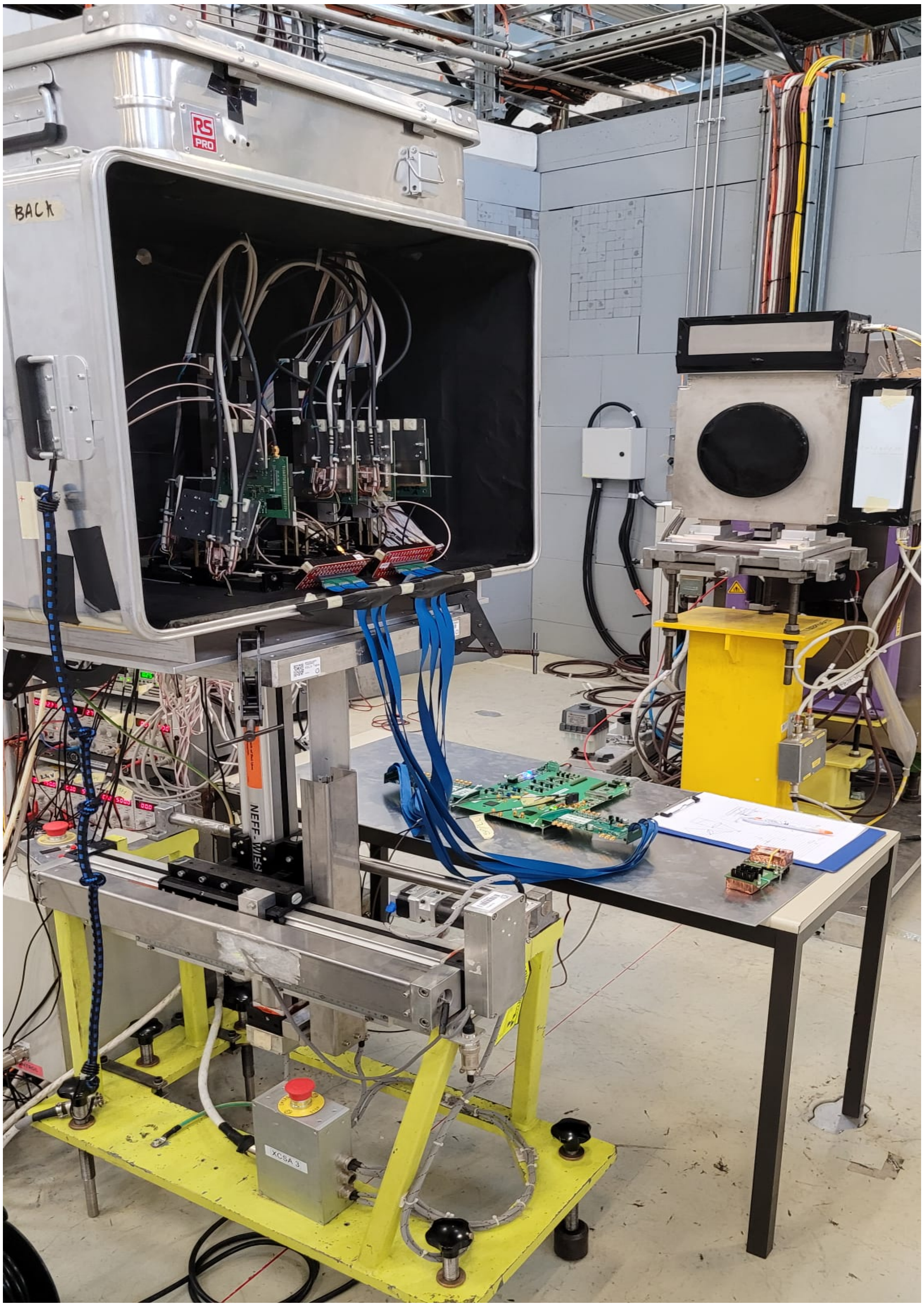


Figure 4.2: *Photo of the beam-test setup. The PicoTDC Board is visible on the table, with the LIROC cards mounted.*

4.1.1 Low Gain Avalanche Detectors

LGADs are an evolution of the n-on-p planar silicon sensors, optimized to provide both good spatial resolution and high timing performance using an internal low multiplication mechanism obtained via the implantation of an heavily doped layer below the p-n juction. Figure 4.3 shows the typical base elements of an LGAD.

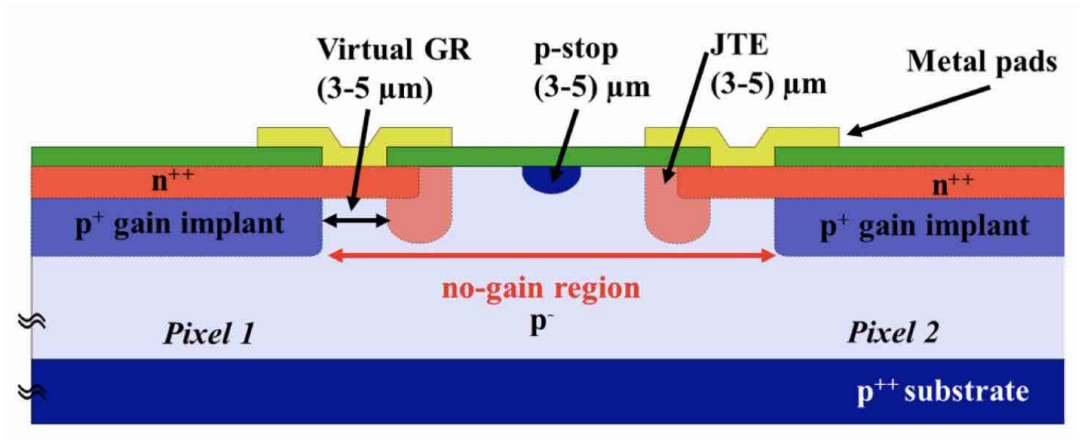


Figure 4.3: *LGAD design, with an additional p+ layer right below the n++ type silicon.*

The one that is reported is an n++/p+/p-/p++ junction, where the p+ zone under the n++ electrode acts as gain layer and it's where the ionization process takes place. In particular the detectors are equipped with an additional virtual Guard Ring and a Junction Termination Extension (JTE) at the pixel edges. These elements are implanted to prevent premature breakdown at the pad borders and to ensure the uniformity of the electric field [19]. The introdution of the p+ doped layer creates an high electric field which accelerates the electrons enough to start the multiplication process.

LGADs are especially used to detect charged particles; looking in particular at MIPs, they create ≈ 70 electron-hole pairs per micrometer, providing a signal high enough to allow for the usage of low gain (O(10)). This feature avoids the increase in sensor noise together with difficulties in sensor segmentation and the high power consumption after irradiation.

During the test beam a 25 μm thick PIN-LGAD [20] with an area of 1 mm^2 whose structure is shown if Figure 4.4 was placed on the fourth plane of the telescope. Both the PIN and the LGAD share the same intrinsic structure with the only difference that the PIN is not provided with the gain layer and both are surrounded by five concentric guard rings where the inner one is the biasing ring.

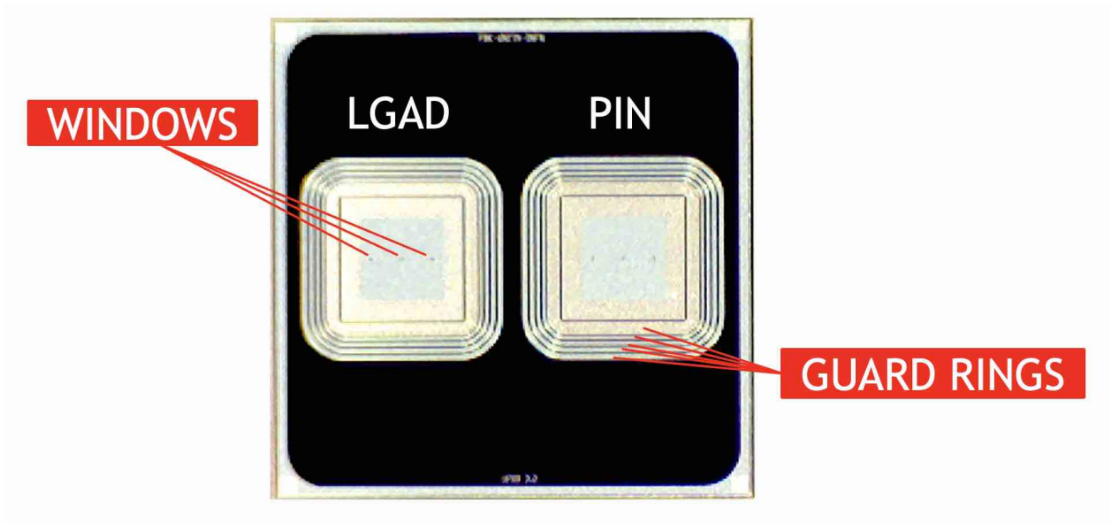


Figure 4.4: *PIN-LGAD structure. The single LGAD on the left and the equivalent PIN on the right. The PIN shares the same structure but without the gain layer. Each pad is surrounded by five concentric guard rings.*

The whole surface of the sensor is covered by metallization with the exception of three 400 µm^2 windows. The PIN shares the same design of the LGAD but without the gain layer.

Other LGAD sensors utilized during the data taking were two 50 µm thick LGADs with an area of 1.3x1.3 mm^2 manifactured by Hamamatsu Photonics K.K. One was placed on the plane 0 of the telescope and acted as a trigger for the TDC time measurement, while the other was placed on the plane 1 to provide the first time tag.

4.1.2 SiPM matrices

Other sensors under study during the test beam were SiPM detectors. In particular, the ones used consisted of matrices of 3x3 FBK NUV-HD-LFv2 SiPMs with active area 1 mm^2 and pixel pitch of 40 µm. Every matrix is covered by a standard silicone protection layer of 1, 1.5 and 3 mm thickness based on the sample [21][22]. Each SiPM of the matrix was connected independently to one channel of the LIROC card, from which the voltage supply was also provided.

SiPMs are single photon detectors based on an array of 10^2-10^4 SPADs[1] with a pitch of 10-100 µm and can be modeled by the equivalent circuit shown in Figure 4.5 and composed by a resistance R_d of about 1 kΩ in parallel to a capacitance C_d of about 10 fF which represents the junction depletion region.

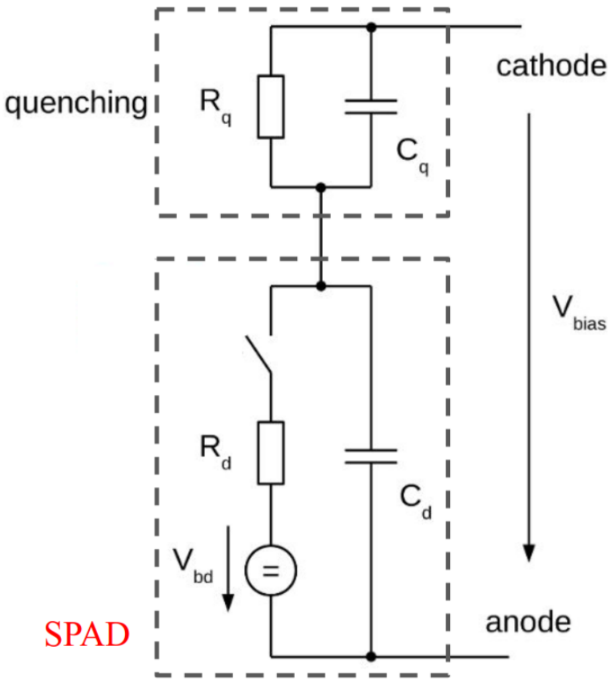


Figure 4.5: *Equivalent circuit of a SPAD.*

During operation the SiPMs are supplied with a voltage higher than the breakdown voltage of the junction. This mode, known as Geiger mode, allows holes to perform impact ionization and participate to the avalanche. The resulting signal is not proportional to the charge released by the impinging particle and allows single photon detection. The gain factor in this working mode is typically of the order of 10^6 in analog SPADs.

[1]Single Photon Avalanche Diodes

In order to stop the avalanche, a series of quenching resistor R_q of the order of 10 kΩ-10 MΩ in parallel to a capacitance C_q of the order of the fF is introduced for each SPAD of the SiPM matrix. The passage of a particle closes the equivalent circuit causing an exponential voltage drop in the node between C_d and C_q which time constant corresponds to $\tau_d = R_d(C_q + C_d)$; the process ends when the current passing through R_d reaches the threshold value given by $I_d \approx V_{ov}/(R_q + R_d) \approx V_{ov}/R_q$ where $V_{ov} = V_{bias} - V_{bd}$ is the overvoltage (i.e. the voltage above the breakdown voltage value of the device) provided to the SiPM. When this condition is reached, the avalanche is said to be "quenched" and the SPAD recovers with a recharge time costant $\tau_r = R_q(C_q + C_d)$.

The gain of the SPAD is defined to be $G = (V_{bias} - V_{bd})C_d/e$ where e is the electron charge; usually this value, which is directly proportional to the product of the overvoltage and of the capacitance associated to the detector producing a single photon signal well over the electronic noise level.

4.2 Data Analysis

The detector was configured to have, going from plane 0 to plane 4, the trigger LGAD (L0), one head LGAD (L1), two SiPM matrices, for which the results are not reported in this thesis, and the final tail LGAD (L4). The main focus of the analysis is to evaluate the timing performance for the two LGADs connected to the PicoTDC board through the LIROC B card and evaluate the Time of Flight (TOF) resolution of the sensors before and after the correction for the time slewing of the output signals.

The PicoTDC was working in double edge mode to retrieve the information of the Time Over Threshold (TOT) of the signals coming from the detectors that will be used for the time slewing correction. The data was selected to contain only events with a single rising edge and the corresponding falling edge of both LGADs signals. The dataset used in this analysis contains the information of six different runs each identified by a run number; the runs used are numbers 579, 582, 583, 584, 585 and 586.

4.2.1 Time slewing correction

For the measurement of the time resolution (of the LGAD sensors) a correction for the time slewing is needed. The difference between the time measure of the discriminated rising edges was correlated to the TOT of the two signals; the data was then fitted with a polynomial function and the value of the function was subtracted to the time difference.

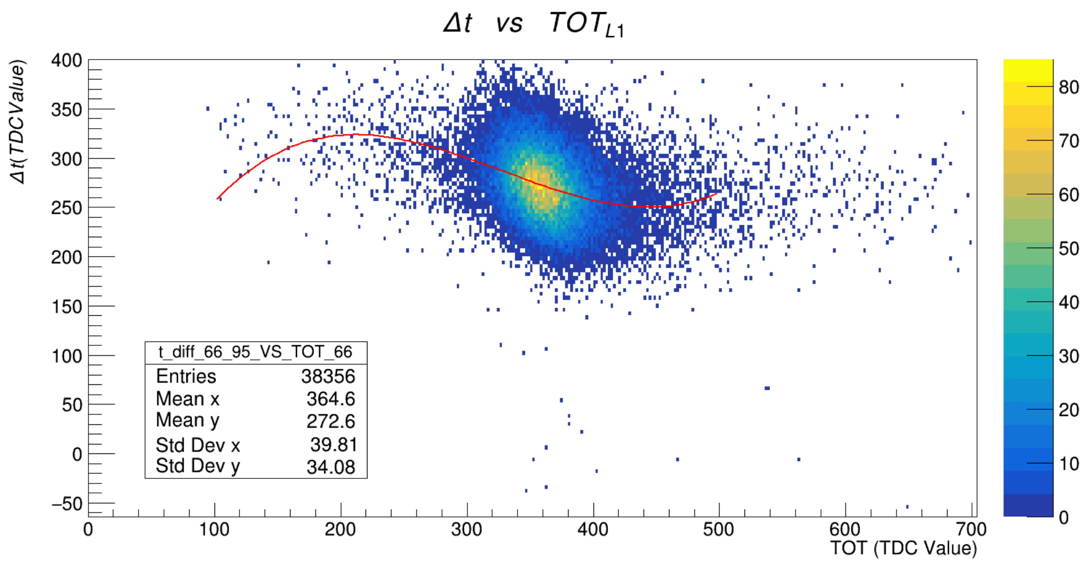
The correction runs as in the following:

- The raw time differences were correlated to the TOT of the head LGAD sensor (named L1) and the data was fitted with a 3^{rd} grade polynomial function.

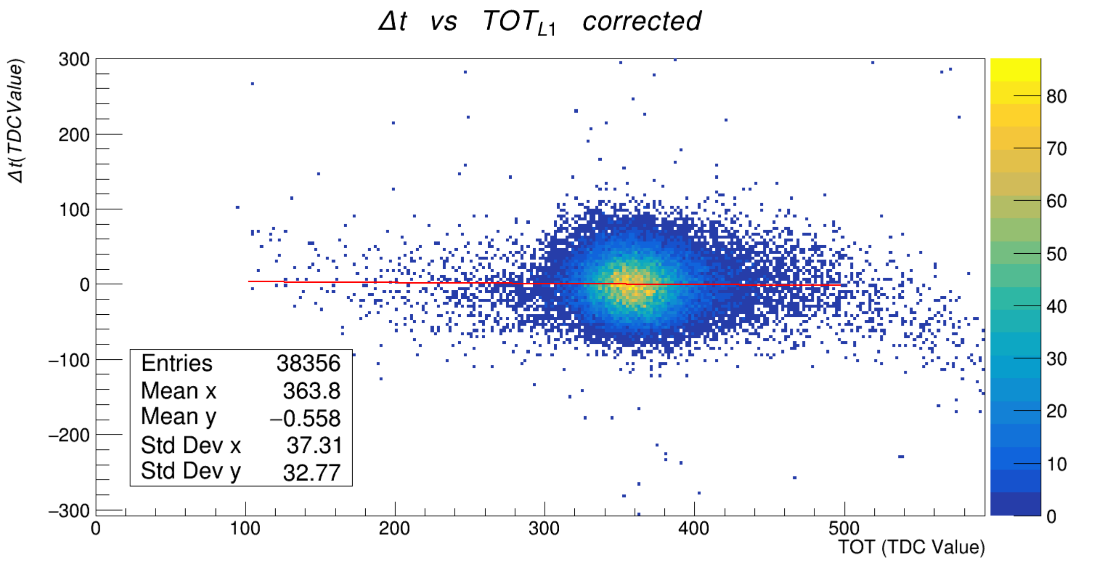
- For each time difference, the value of the function evaluated for the corresponding L1 TOT was subtracted.
- These corrected time differences were correlated to the TOT of the tail LGAD sensor (named L4) and was again fitted with a 3$^{\text{rd}}$ grade polynomial function.
- The same procedure of the second step is then repeated using the function for the L4 TOT onto the already corrected data to correct the data for both sensors' time slewing.
- After the data was corrected for both sensors, each time difference was fitted with a gaussian function to retrieve the measure of the time of flight resolution.

Figures 4.6 and 4.7 report the main quantities for run 582 among the data acquisition runs considered. This correction removes the dependence of the time of flight measurement from the TOT measure of the two LGADs signals, narrowing the spread of the time of flight measurement and improving the resolution.

The histograms relative to the other runs considered can be found in Appendix B.1

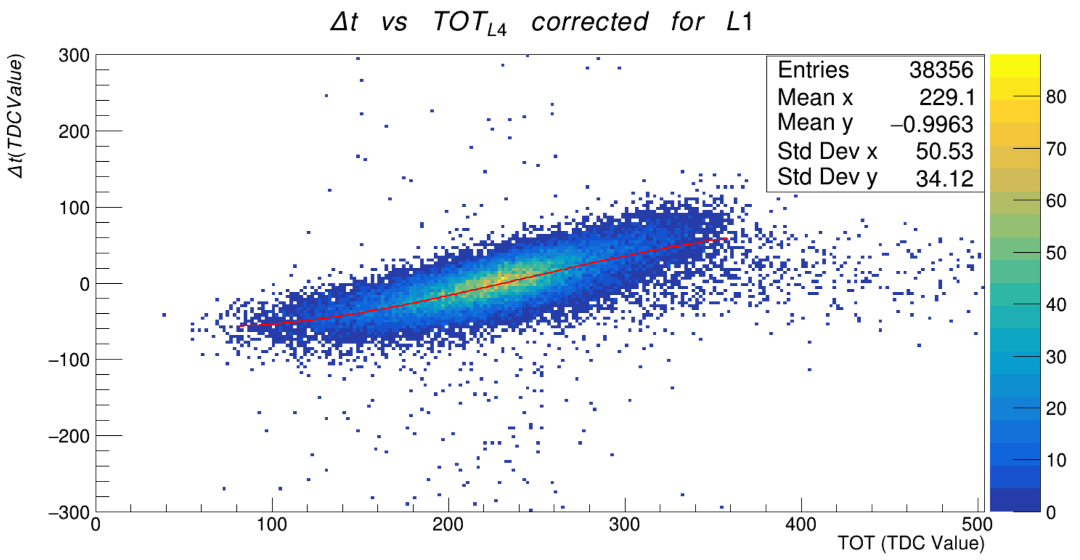


(a) Time difference versus head LGAD detector (L1) TOT.

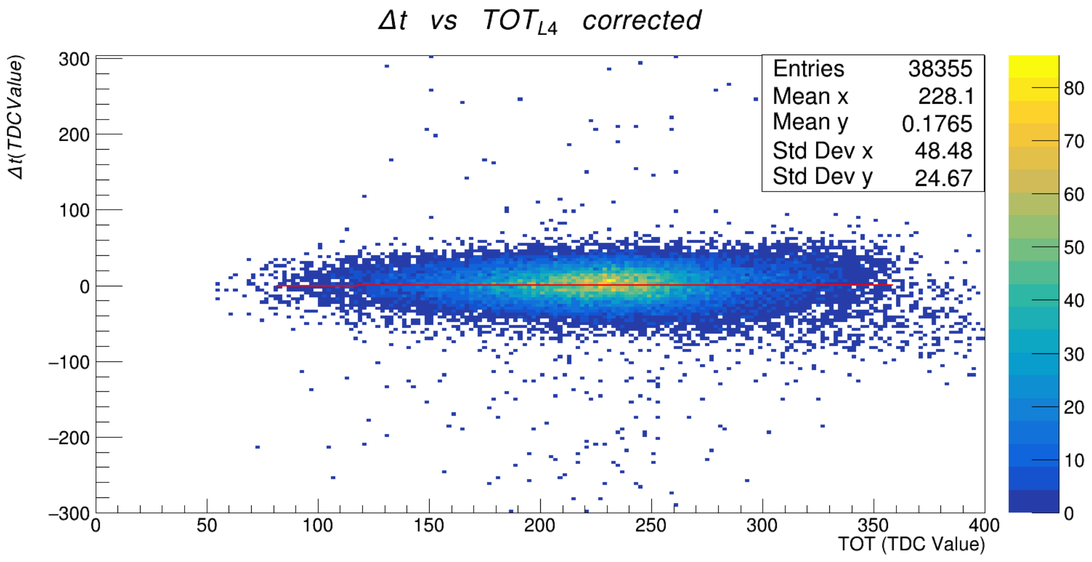


(b) Time difference versus head LGAD detector (L1) TOT after correction.

Figure 4.6: *2D histograms of the time of flight measure versus the L1 TOT values before (a) and after (b) time slewing correction.*



(a) Time difference versus tail LGAD detector (L4) TOT after L1 correction.



(b) Time difference versus tail LGAD detector (L4) TOT after both corrections.

Figure 4.7: *2D histograms of the time of flight measure versus the L4 TOT values before (a) and after (b) time slewing correction.*

4.2.2 Time resolution

After the time slewing correction, the resulting time differences of the two LGADs were plotted and fitted with a gaussian function PDF[2]:

$$f(x) = \frac{\text{Const}}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{4.1}$$

in which the "Const" factor represents the gaussian area, while the mean μ and the standard deviation σ represents the measured time of flight and the estimated resolution for the 2-channel time measurement. By using two channels and two independent sensors the time resolution on difference of the two channels time measurements are given by:

$$\sigma_{TOF} = \sqrt{\sigma_{L1}^2 + \sigma_{L4}^2} \tag{4.2}$$

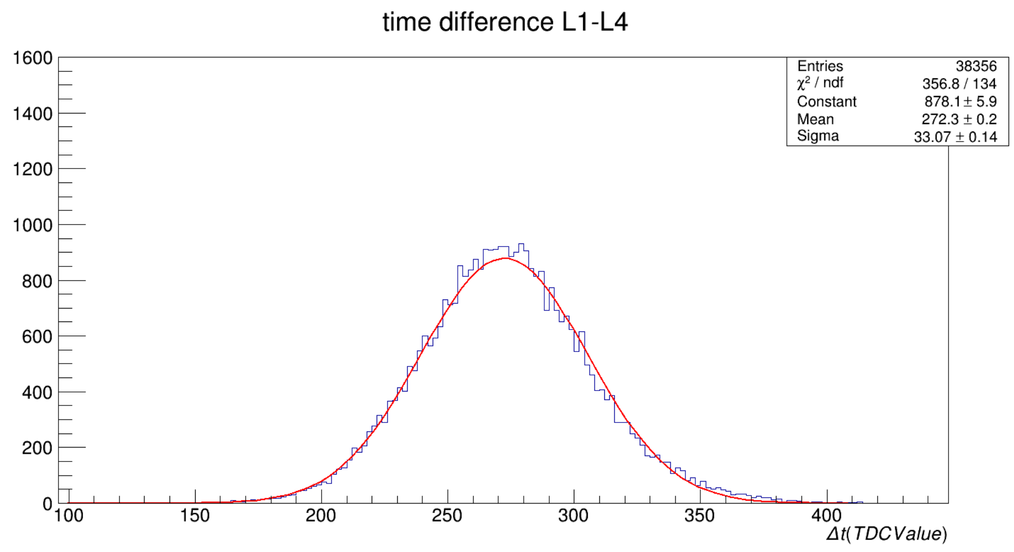
An example of the time difference histograms before and after the correction are reported in Figure 4.8. The histograms relative to the other run acquired are reported in Appendix B.2.

The resulting time resolution is expressed in TDC counts, so to get the corresponding time value in picoseconds must be multiplied by the PicoTDC LSB value of 3.05 ps.

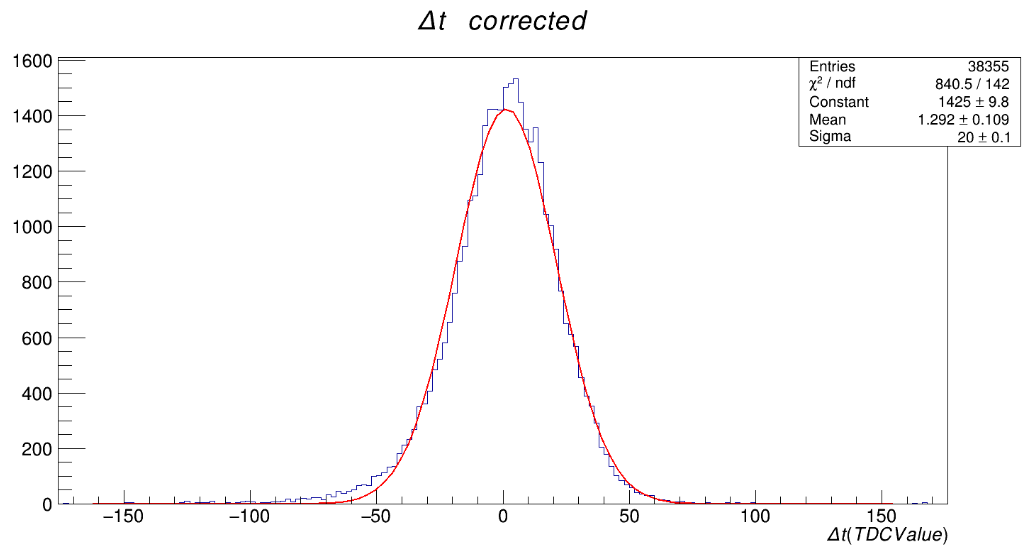
To estimate the time of flight resolution for the LGAD couple the measure were averaged obtaining a value of $\sigma_{L1-L4} = 19.86 \pm 0.08$ TDC counts which corresponds to a time value of $\sigma_{L1-L4} = 60.57 \pm 0.25$ ps.

As a first approximation the time resolution of the two LGADs is similar, therefore one can deduce the time resolution of the individual sensor as $\sigma_{LGAD} = \sigma_{TOF}/\sqrt{2}$ which gives a resolution of $\sigma_{LGAD} = 42.83 \pm 0.18$ ps. The time resolution results for each run analyzed are reported in Table 4.1.

[2]Probability Density Function



(a) Time difference distribution before time slewing correction.



(b) Time difference distribution after time slewing corrections.

Figure 4.8: *Histograms of the Time Of Flight distribution before (a) and after (b) time slewing corrections of run 582.*

Run Number	σ_{L1-L4} (TDC counts) before correction	σ_{L1-L4} (TDC counts) after correction	σ_{L1-L4} (ps) after correction	$\sigma_{L1(L4)}$ (ps)
579	33.16 ± 0.08	20.00 ± 0.04	61.00 ± 0.12	43.13 ± 0.08
582	33.07 ± 0.14	20.00 ± 0.10	61.00 ± 0.31	43.13 ± 0.22
583	33.25 ± 0.10	19.79 ± 0.07	60.36 ± 0.21	42.68 ± 0.15
584	33.07 ± 0.13	19.72 ± 0.09	60.15 ± 0.27	42.53 ± 0.19
585	33.47 ± 0.16	19.98 ± 0.10	60.94 ± 0.31	43.09 ± 0.22
586	33.52 ± 0.14	19.65 ± 0.09	57.95 ± 0.27	40.98 ± 0.19

Table 4.1: *Table with the time resolution results on the time difference distribution histograms before and after the correction for time slewing.*

Conclusion

In this thesis I described several aspects that I followed closely related to the PicoTDC Board. This board was designed by the ALICE-TOF collaboration of Bologna together with INFN electronics laboratory as a test environment for the development of the new TDC Readout Module (TRM2) which will replace the damaged TRM cards of the ALICE-TOF detector during LHC Run 3 and Run 4. In particular:

- I described the newly developed USB Super-Speed interface through the FTDI FT601 chip. I implemented a new firmware architecture to provide a fast and reliable interface with the already existent IPBus SoC through the 5Gbps USB Super-Speed interface integrated on board to perform configuration and readout operations of the two PicoTDC ASICs. I also adapted the existing software using Interprocess Communication mechanism to overcome some limitations of the driver provided by the FTDI company.

- I worked on the LIROC front-end chip, establishing the I2C interface and developing firmware modifications for its configuration to process the analog signals coming from the sensors before sending them to the PicoTDCs.

- Finally, the DAQ chain was then tested at CERN in June 2024 during a test beam to evaluate its stability and performance. The preliminary results of the data analysis for a couple of LGAD sensors gave an estimate of the Time of Flight resolution for the LGAD-LIROC-PicoTDC DAQ chain of $\sigma = 42.83 \pm 0.18$ ps.

In conclusion, the DAQ system has demonstrated to be reliable and a good tool for test beams and laboratory analysis. Next steps will be to optimize the front-end electronics configurable parameters of the LIROC to reach the maximum timing performance with a given sensor.

Bibliography

[1] 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`

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

[3] Texas Instruments, *Fundamental Theory of PMOS Low-Dropout Voltage Regulators*, application report (2018).

[4] Microchip Technology Inc., *PolarFire FPGA Product Overview*, product report (2021).

[5] Microchip Technology Inc. and its subsidiares, *PolarFire FPGA Packaging and Pin Descriptions User Guide*, product report (2024).

[6] Cisco systems, *Serial-GMII Specification*, specification revision 1.8 (2005).

[7] 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`

[8] Microchip Technology Inc., *VSC8541-02 and VSC8541-05 datasheet Single Port Gigabit Ethernet Copper PHY with GMII/RGMII/MII/RMII Interfaces*, product data sheet revision 4.2.

[9] Infineon Ltd., *EZ-USB FX3 SuperSpeed USB Controller data sheet*, product datasheet revision 21.0

[10] Future Technology Devices International Ltd., *FT600Q-FT601Q IC Datasheet (USB 3.0 to FIFO bridge)*, product datasheet revision 1.05

[11] 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`

[12] OpenCores, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, specification revision B.3 (2002).

[13] CERN, IPbus user guide, *On-chip bus*, available:
`https://ipbus.web.cern.ch/doc/user/html/firmware/bus.html`

[14] R. Frazier et al., *The IPbus Protocol, an IP-based control protocol for ATCA/μTCA*, IPbus version 2.0. (2013).

[15] Microchip Technology Inc., *CoreFIFO IP User Guide*, available at `https://ww1.microchip.com/downloads/aemDocuments/documents/CoreFifo_v3.1.pdf`.

[16] CERN , GitHub, *ipbus/ipbus-firmware*, available:
`https://github.com/ipbus/ipbus-firmware`

[17] S. Geminiani, P. Antonioli, M. Giacalone, J. Succi, Baltig, "Software_PicoTDCboard", *geminian/PicoTDCboard*, available:
`https://baltig.infn.it/geminian/picotdcboard/-/tree/main/So-ftware_PicoTDCboard?ref_type=heads`

[18] Weeroc, *LIROC Datasheet*, product datasheet revision 0.4, available at `https://www.weeroc.com/read_out_chips/liroc/`

[19] P. Fernandez-Martinez, D. Flores, S. Hidalgo, V. Greco, A. Merlos, G. Pellegrini, D. Quirion, *Design and Fabrication of an Optimum Peripheral Region for Low Gain Avalanche Detectors*, Nucl. Instrum. Methods Phys. Res. A 821, pp. 93–100, 2016, DOI: 10.1016/j.nima.2016.03.049.

[20] F. Carnesecchi, S. Strazzi, A. Alici et al. *A new low gain avalanche diode concept: the double-LGAD.* Eur. Phys. J. Plus 138, 990 (2023). https://doi.org/10.1140/epjp/s13360-023-04621-x

[21] A. Gola et al., NUV-Sensitive Silicon Photomultiplier Technologies Developed at Fondazione Bruno Kessler, Sensors 2019, 19(2), 308; https://doi.org/10.3390/s19020308

[22] S. Gundacker et al., On timing-optimized SiPMs for Cherenkov detection to boost low cost time-of-flight PET, 2023 Phys. Med. Biol. 68 165016, DOI 10.1088/1361-6560/ace8ee

Appendix A

Library Source Code

A.1 FTDI_ipbusOperation function definitions

```
//utility functions
void CharToUint(unsigned char* p, uint32_t* d)
{
        *d =0;
        for(int i =0 ; i<4;i++,p++){
                uint8_t u8 = (uint8_t)*p;
                *d |= (u8 <<8*i);
        }
};

void UintToChar(uint32_t* integer, UCHAR* charBuff)
{
        for(int i = 0; i< 4; i++, charBuff++){
                *charBuff = (*integer >>8*i)&0xFF;
        }
};
```

```
//single word non-incremental write operation
uint32_t wrnReg(FT_HANDLE fHandle, uint32_t add, uint32_t data, uint32_t mask){
        UCHAR wrBuff[4*3];
        UCHAR rdBuff[4*2];
        FT_STATUS fStatus = FT_OK;
        ULONG bytes = 0;
        uint32_t mData = data&mask;
        uint32_t ipbStatus = 0;

        //compose ipbus command header
        wrBuff[0] = TRANS_WRN;
        wrBuff[1] = 0x01;
        wrBuff[2] = 0x00;
        wrBuff[3] = 0x20;
        //compose address
        UintToChar(&add,&wrBuff[4]);
        //data to be written
        UintToChar(&mData,&wrBuff[8]);

        //write data packet to FPGA cmd buffer
        fStatus = FT_WritePipeEx(fHandle,0,&wrBuff[0],12,&bytes,5000);
        if(FT_FAILED(fStatus)){
                printf("Error during USB write operation! Status : %d\n",fStatus);
                printf("Bytes transfered : %d\n",bytes);
                fStatus = FT_AbortPipe(fHandle,0x0);
                printf("Aborting pipe with status = %d\n",fStatus);
                return fStatus;
        }
        //read ipbus operation status from FPGA output buffer
        fStatus = FT_ReadPipeEx(fHandle,0,&rdBuff[0],8,&bytes,5000);
        if(FT_FAILED(fStatus)){
                printf("Error during USB read operation! Status : %d\n",fStatus);
                printf("Bytes transfered : %d\n",bytes);
                fStatus = FT_AbortPipe(fHandle,0x0);
                return fStatus;
        }
        else{
                CharToUint(&rdBuff[4],&ipbStatus);
                if(ipbStatus== 0x20000130  ){
                        return ipbStatus;
                }
                else{
                        printf("Error during ipbus operation! Status %x\n",ipbStatus);
                        return ipbStatus;
                }
        }
}
```

```
//single word non-incremental read operation
uint32_t rdnReg(FT_HANDLE fHandle,uint32_t add, uint32_t* rData, uint32_t mask ){

        UCHAR wrBuff[4*2];
        UCHAR rdBuff[4*3];
        FT_STATUS fStatus = FT_OK;
        ULONG bytes = 0;
        uint32_t ipbStatus = 0;
        uint32_t data = 0;

        //compose ipbus command header
        wrBuff[0] = TRANS_RDN;
        wrBuff[1] = 0x01;
        wrBuff[2] = 0x00;
        wrBuff[3] = 0x20;

        //compose address
        UintToChar(&add,&wrBuff[4]);
        //write data packet to FPGA cmd buffer
        fStatus = FT_WritePipeEx(fHandle,0,&wrBuff[0],8,&bytes,5000);
        if(FT_FAILED(fStatus)){
                printf("Error during USB write operation! Status : %d\n",fStatus);
                printf("Bytes transfered : %d\n",bytes);
                fStatus = FT_AbortPipe(fHandle,0x0);
                return fStatus;
        }

        //read data and ipbus operation status from FPGA output buffer
        fStatus = FT_ReadPipeEx(fHandle,0,&rdBuff[0],12,&bytes,5000);
        if(FT_FAILED(fStatus)){
                printf("Error during USB read operation! Status : %d\n",fStatus);
                printf("Bytes transfered : %d\n",bytes);
                fStatus = FT_AbortPipe(fHandle,0x0);
                return fStatus;
        }
        else{
                CharToUint(&rdBuff[8],&ipbStatus);
                CharToUint(&rdBuff[4],&data);
                *rData = data&mask;
                if(ipbStatus== 0x20000120 ){
                        return ipbStatus;
                }
                else{
                        printf("Error during ipbus operation! Status %x\n",ipbStatus);
                        return ipbStatus;
                }
        }
}
```

```
uint32_t ReadBlock(FT_HANDLE fHandle,uint32_t add,uint32_t* rData,uint32_t nWords)
{
  int nTrans=1;
  if(nWords>255) nTrans=(int)(nWords/255)+1;

  UCHAR wrBuff[8232];
  UCHAR rdBuff[1052700]; // 1MB buffer (262144 32-bit words) + space for junk and status words
  FT_STATUS status;
  ULONG readBytes = 0;
  uint32_t ipbStatus;

  int index = 0;
  for(int i=0; i<nTrans;i++)
    {
      wrBuff[index] = TRANS_RDN;
      if(i!= (nTrans-1)) wrBuff[index+1]=0xFF;
      else wrBuff[index+1]=(uint8_t)(nWords%255);
      wrBuff[index+2] = (uint8_t)i;
      wrBuff[index+3] = 0x20;
      UintToChar(&add, &wrBuff[index+4]);

      index+=8;

    }

  status = FT_WritePipeEx(fHandle,USB_CHANNEL,&wrBuff[0],8*nTrans,&readBytes,USB_TIMEOUT);
  fsleep();
  ULONG reqBytes = 4+(1024*(nTrans-1))+(((uint8_t)wrBuff[index-7])*4)+4;
  status = FT_ReadPipeEx(fHandle,USB_CHANNEL,&rdBuff[0],reqBytes,&readBytes,USB_TIMEOUT);
  memmove(&rdBuff[0],&rdBuff[4],(readBytes-4));

  int endWord = 255;
  for (int it = 0; it < nTrans; it++) {
    if ( (it == (nTrans -1)) ) endWord=wrBuff[index-7];
    for (int w = 0; w < endWord; w++) {
      CharToUint(&rdBuff[1024*it+w*4],rData); rData++;
    }
    CharToUint(&rdBuff[1024*it+endWord*4],&ipbStatus);
  }

  return ipbStatus;

}
```

A.2 libUSBsh Source Code

```
void uhal::disableLogging() { ; }
uhal::HwInterface::HwInterface() {d=0;}
uhal::ConnectionManager::ConnectionManager(std::string c) { conn=c; }
uhal::HwInterface uhal::ConnectionManager::getDevice(std::string c){
  int fStatus;
  fStatus = FTC_Create();
  if(fStatus){
    printf("ERROR OPENING CONNECTION WITH THE DEVICE!!! STATUS = %d\n",fStatus);
    exit(EXIT_FAILURE);
  }
  HwInterface a;
  return a;
}


int attachUsbshSharedMemory(int *shMemId, unsigned long *ps)
{
  int shsize,ret;
  key_t key;
  char keyFile[50];
  int usbshShMem;
  shsize = sizeof(usbshShMem_t);
  sprintf(keyFile,"/PS/src/usb/usbsh/usbshSharedMemory.key");

  key = ftok(keyFile,USBSH_PROJID);
  errno = 0;
  //  printf("KEY SHARED MEMORY: 0x%x\n",key);
  usbshShMem = shmget(key, shsize,IPC_CREAT|0666);
  //  printf("ID SHARED MEMORY: 0x%x\n",dShMem);
  if (errno) {
    perror("shmget:");
    *shMemId = -1;
    *ps = (unsigned long)NULL;
    ret=EXIT_FAILURE;
  } else {
    //    printf(" shmem (key 0x%x,size %d bytes)\n", key,shsize);
    *ps = (unsigned long)shmat(usbshShMem, 0, SHM_R|SHM_W);
    if (errno) {
      perror("shmat");
      ret=EXIT_FAILURE;
      usbshShMem=-1;
    } else {
      *shMemId = usbshShMem;
      ret=EXIT_SUCCESS;
    }
  }
  return (ret);
}
```

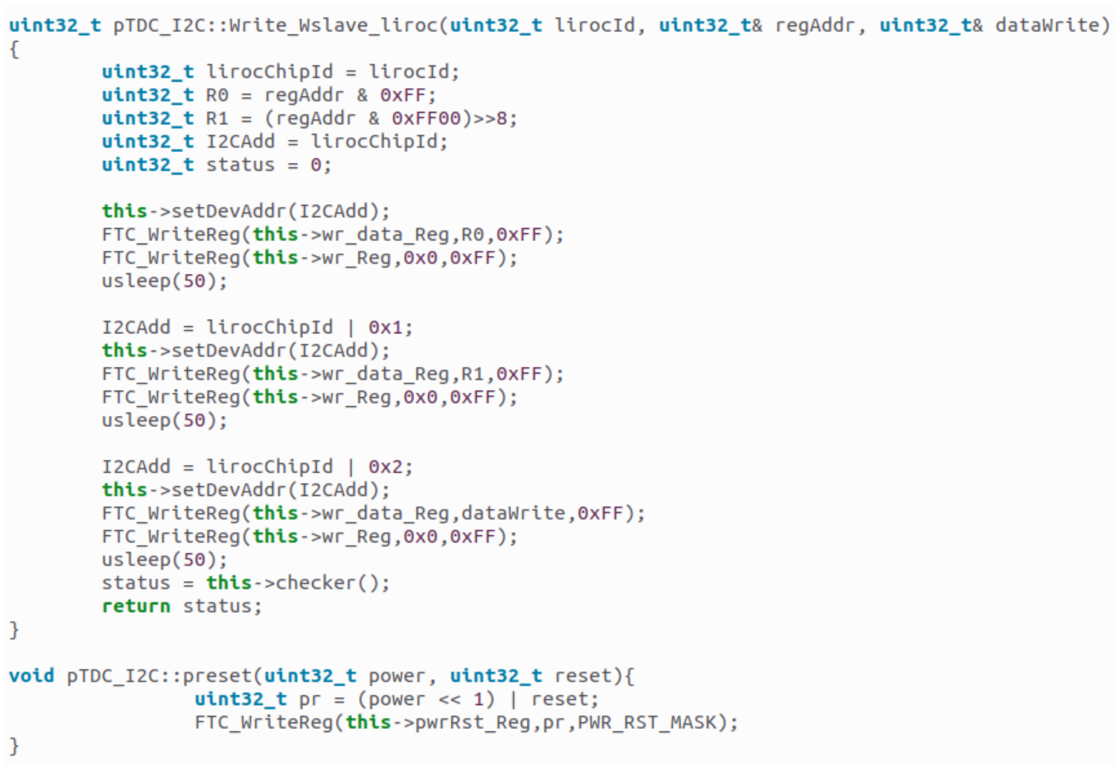
```
int setupUsbshMessageQueue(void)
{
  key_t key;
  char keyFile[100];
  int usbshMessageQueue;
  sprintf(keyFile,"/PS/src/usb/usbsh/usbshMessageQueue.key");
  key = ftok(keyFile,USBSH_PROJID);
  if ((usbshMessageQueue=msgget(key,IPC_CREAT|0666))==-1) {
    perror("msgget");
    return EXIT_FAILURE;
  } else {
    return usbshMessageQueue;
  }
}

sem_t *ftSem;
int ftCh;
int usbshMessageQueueClient;
usbshShMem_t *UClient;
```

```c
int FTC_Create()
{
    //attach to shared Memory
    int ret, usbshShMem;
    unsigned long p;
    ret = attachUsbshSharedMemory(&usbshShMem, &p);
     if (ret)
      {
        printf("Failed to attach shared memory\n");
        return -1;
      }
      else
      {
        UClient = (usbshShMem_t *)p;
      }

    // attach to message queue
    usbshMessageQueueClient = setupUsbshMessageQueue();

    //attach to general semaphore
      sem_t *genSem = sem_open("USBSH-CLIENT", O_CREAT);
      if (genSem == NULL ) return -2;
      //perror("sem_open");
      int semVal=0;
      sem_getvalue(genSem,&semVal);
      //printf("general semaphore Value %d|n",semVal);
      for(int i =0; i<semVal;i++) {sem_wait(genSem);}
      //sem_getvalue(genSem,&semVal);
      //printf("general semaphore final Value %d|n",semVal);

    usbshMessage_t reg;
    reg.pid = getpid();
      reg.mtype = 3;
      reg.cmd = USBSH_CMD_REGISTER;
      reg.value = 0x0;
      reg.address = 0x0;
      msgsnd(usbshMessageQueueClient,&reg,sizeof(usbshMessage_t)-4,IPC_NOWAIT);
      sem_wait(genSem);

      int channel=-1;
      for(int i = 0 ; i<USBSH_CHANNELS; i++)
      {
        if(UClient->usbWin[i].pid == reg.pid){
        //printf("found %d|n",i);
        channel = i;
        break;
        }
  }
  ftCh = channel;
  if(channel <0)
  {
    printf("FATAL! Process has not been registered! Exiting...\n");
    return -3;
  }
  else
  {
    char semName[15];
    printf("Process registered at channel %d! connecting to semaphore...\n",channel);
    sprintf(semName,"USBSH-CHANNEL%d",channel);
    ftSem = sem_open(semName,O_CREAT);
    //perror("sem_open");
    if(ftSem == NULL) return -4;
  }

  sem_getvalue(ftSem,&semVal);
  //printf("semaphore Value %d\n",semVal);
  for(int i =0; i<semVal;i++) {sem_wait(ftSem);}
  //sem_getvalue(ftSem,&semVal);
  //printf("semaphore final Value %d\n",semVal);
  return 0;
}

int FTC_Close(){
    //send release message
    usbshMessage_t unreg;
    unreg.pid = getpid();
      unreg.mtype = 3;
      unreg.cmd = USBSH_CMD_RELEASE;
      unreg.value = 0x0;
      unreg.address = 0x0;
      msgsnd(usbshMessageQueueClient,&unreg,sizeof(usbshMessage_t)-4,IPC_NOWAIT);
      sem_wait(ftSem);
      //close semaphore and detach ShMem
      sem_close(ftSem);
      shmdt(UClient);
      return 0;
}
```

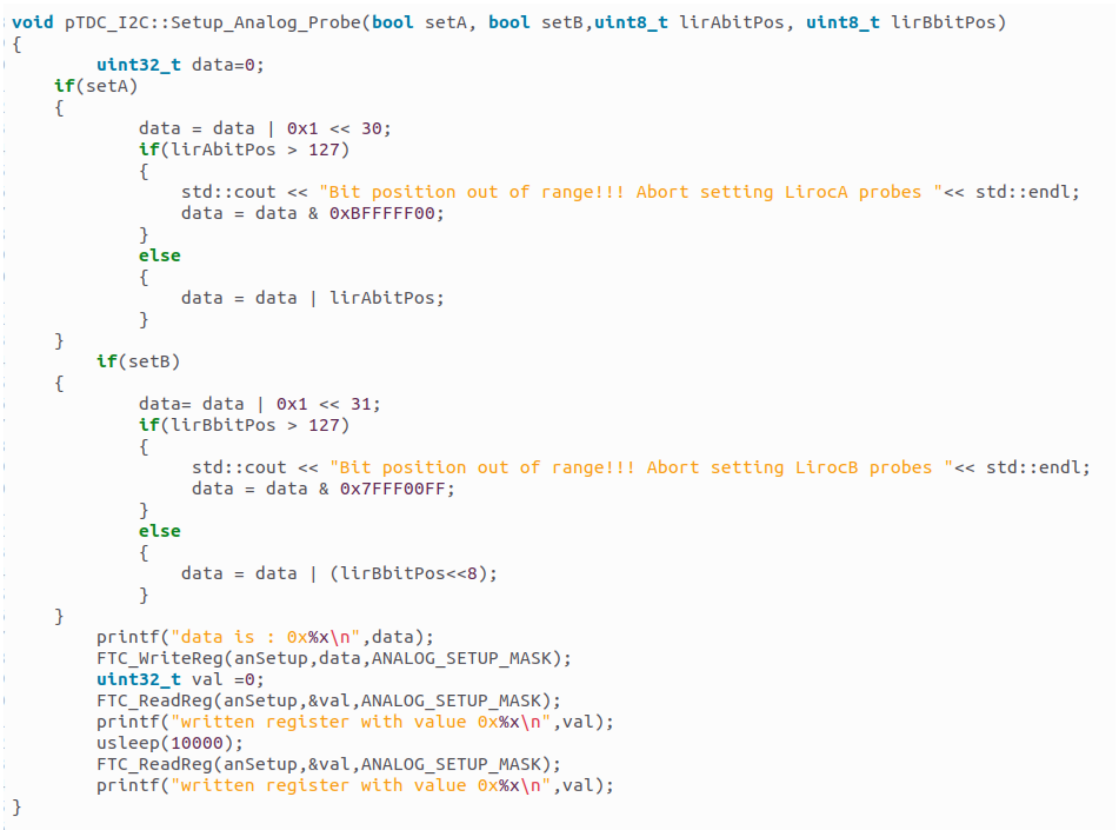
```c
int FTC_ReadReg( uint32_t address,uint32_t *rdData,uint32_t mask){
    usbshMessage_t m;
    m.pid = getpid();
    m.mtype = 3;
      m.cmd = USBSH_CMD_READ;
      m.value = 0x0;
      m.address = address;
      m.mask = mask;
      msgsnd(usbshMessageQueueClient,&m,sizeof(usbshMessage_t)-4,IPC_NOWAIT);
      sem_wait(ftSem);
      if (UClient->usbWin[ftCh].status == EXECUTED)
      {
          *rdData = UClient->usbWin[ftCh].buf[0];
      }
      return UClient->usbWin[ftCh].status;
}

int FTC_WriteReg( uint32_t address,uint32_t value,uint32_t mask){
    usbshMessage_t m;
    m.pid = getpid();
    m.mtype = 3;
      m.cmd = USBSH_CMD_WRITE;
      m.value = value;
      m.address = address;
      m.mask = mask;
      msgsnd(usbshMessageQueueClient,&m,sizeof(usbshMessage_t)-4,IPC_NOWAIT);
      sem_wait(ftSem);
      return UClient->usbWin[ftCh].status;
}

int FTC_ReadBlock( uint32_t address, uint32_t* rdData, uint32_t nWords){
    usbshMessage_t m;
    m.pid = getpid();
    m.mtype = 3;
      m.cmd = USBSH_CMD_RDBLOCK;
      m.value = nWords;
      m.address = address;
      m.mask = 0x0;
      msgsnd(usbshMessageQueueClient,&m,sizeof(usbshMessage_t)-4,IPC_NOWAIT);
      sem_wait(ftSem);
      if (UClient->usbWin[ftCh].status == EXECUTED)
      {
          for (int i =0;i<nWords;i++){
          *rdData = UClient->usbWin[ftCh].buf[i];
          rdData++;
          }
      }
      return UClient->usbWin[ftCh].status;
}
```

A.3 LIROC I2C library source code

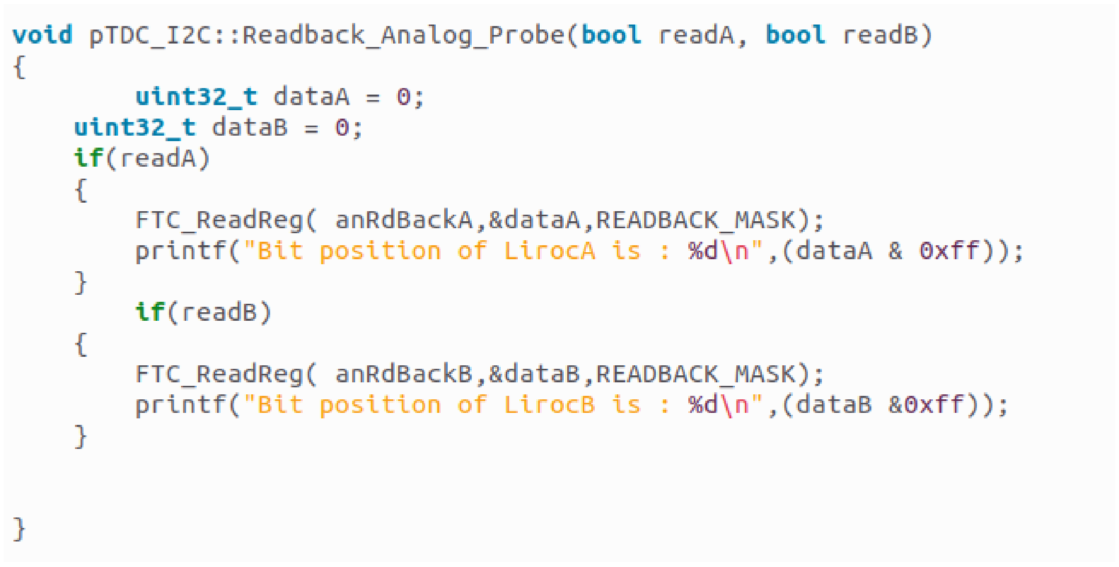
```
#define R0(a,sa) (((a&0x7)<<5)|(sa&0x1F))
#define R1(a) ((a&0x7F8)>>3)
#define LADD(a,sa) ((R1(a) << 8 ) | (R0(a,sa)))

//liroc slow control functions
uint32_t pTDC_I2C::Read_Wslave_liroc(uint32_t lirocId, uint32_t &regAddr, uint32_t* dataRead)
{
        uint32_t lirocChipId = lirocId;
        uint32_t R0 = regAddr & 0xFF;
        uint32_t R1 = (regAddr & 0xFF00)>>8;
        uint32_t I2CAdd = lirocChipId;
        uint32_t status = 0;

        this->setDevAddr(I2CAdd);
        FTC_WriteReg(this->wr_data_Reg,R0,0xFF);
        FTC_WriteReg(this->wr_Reg,0x0,0xFF);
        usleep(50);

        I2CAdd = lirocChipId | 0x1;
        this->setDevAddr(I2CAdd);
        FTC_WriteReg(this->wr_data_Reg,R1,0xFF);
        FTC_WriteReg(this->wr_Reg,0x0,0xFF);
        usleep(50);

        I2CAdd = lirocChipId | 0x2;
        this->setDevAddr(I2CAdd);
        FTC_WriteReg(this->rd_Reg,0x0,0xFF);
        usleep(50);

        status = this->checker();
        if(!status) FTC_ReadReg(this->rd_data_Reg,dataRead,0xFF);
        else *dataRead = 0xFF;
        return status;
}
```



```
uint32_t pTDC_I2C::Write_Wslave_liroc(uint32_t lirocId, uint32_t& regAddr, uint32_t& dataWrite)
{
        uint32_t lirocChipId = lirocId;
        uint32_t R0 = regAddr & 0xFF;
        uint32_t R1 = (regAddr & 0xFF00)>>8;
        uint32_t I2CAdd = lirocChipId;
        uint32_t status = 0;

        this->setDevAddr(I2CAdd);
        FTC_WriteReg(this->wr_data_Reg,R0,0xFF);
        FTC_WriteReg(this->wr_Reg,0x0,0xFF);
        usleep(50);

        I2CAdd = lirocChipId | 0x1;
        this->setDevAddr(I2CAdd);
        FTC_WriteReg(this->wr_data_Reg,R1,0xFF);
        FTC_WriteReg(this->wr_Reg,0x0,0xFF);
        usleep(50);

        I2CAdd = lirocChipId | 0x2;
        this->setDevAddr(I2CAdd);
        FTC_WriteReg(this->wr_data_Reg,dataWrite,0xFF);
        FTC_WriteReg(this->wr_Reg,0x0,0xFF);
        usleep(50);
        status = this->checker();
        return status;
}

void pTDC_I2C::preset(uint32_t power, uint32_t reset){
                uint32_t pr = (power << 1) | reset;
                FTC_WriteReg(this->pwrRst_Reg,pr,PWR_RST_MASK);
}
```



```
void pTDC_I2C::Setup_Analog_Probe(bool setA, bool setB,uint8_t lirAbitPos, uint8_t lirBbitPos)
{
        uint32_t data=0;
    if(setA)
    {
            data = data | 0x1 << 30;
            if(lirAbitPos > 127)
            {
                std::cout << "Bit position out of range!!! Abort setting LirocA probes "<< std::endl;
                data = data & 0xBFFFFF00;
            }
            else
            {
                data = data | lirAbitPos;
            }
    }
        if(setB)
    {
            data= data | 0x1 << 31;
            if(lirBbitPos > 127)
            {
                 std::cout << "Bit position out of range!!! Abort setting LirocB probes "<< std::endl;
                 data = data & 0x7FFF00FF;
            }
            else
            {
                data = data | (lirBbitPos<<8);
            }
    }
        printf("data is : 0x%x\n",data);
        FTC_WriteReg(anSetup,data,ANALOG_SETUP_MASK);
        uint32_t val =0;
        FTC_ReadReg(anSetup,&val,ANALOG_SETUP_MASK);
        printf("written register with value 0x%x\n",val);
        usleep(10000);
        FTC_ReadReg(anSetup,&val,ANALOG_SETUP_MASK);
        printf("written register with value 0x%x\n",val);
}
```



```
void pTDC_I2C::Readback_Analog_Probe(bool readA, bool readB)
{
        uint32_t dataA = 0;
    uint32_t dataB = 0;
    if(readA)
    {
        FTC_ReadReg( anRdBackA,&dataA,READBACK_MASK);
        printf("Bit position of LirocA is : %d\n",(dataA & 0xff));
    }
        if(readB)
    {
        FTC_ReadReg( anRdBackB,&dataB,READBACK_MASK);
        printf("Bit position of LirocB is : %d\n",(dataB &0xff));
    }


}
```

A.4 LIROC I2C register list

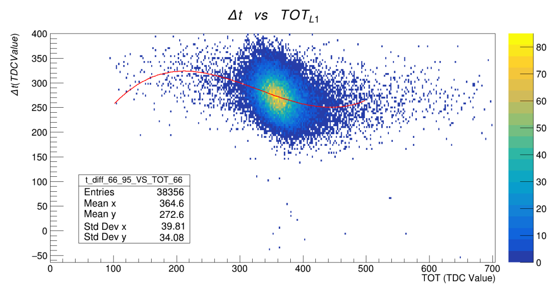
Address	**SubAdd**	**Bit#**	**Default value**	**Name**	**Description**
0-63	0	7-2	000000	DC_PA[5:0]	Channel-by-channel input DC level setting
		1	0	Ctest	Injection capacitance connection switch. Default is switch open(0).
		0	0	NC	Not Connected
	1	7	0	Mask	Mask Trigger. Default is not masked(0).
		6-0	1000000	DAC_local[6:0]	Channel-by-channel 7-bit threshold adjustment
64	0	7	1	EN_PA	Enable of Pre-Amp. Default is enabled(1).
		6	0	PP_PA	Power pusling of Pre-Amp. Default is not power pulsed(0).
		5-2	1010	PA_gain[3:0]	Pre-Amp DC gain adjustment.
		1-0	00	NC	Not Connected.
	1	7	1	EN_7b	Enable of 7-bit channle-by-channel threshold. Default is enabled(1).
		6	0	PP_7b	Power pulsing of 7-bit channel-by-channel threshold. Default is not power pulsed(0)
		5-0	000000	NC	Not Connected.
	2	7	1	EN_disc	Enable of discriminator. Default is enabled(1)
		6	0	PP_disc	Power pulsing of discriminator. Default is not power pulsed(0)
		5	1	Polarity	Discriminator polarity selection. Default is (1): negative trigger out polarity for negative input input charge.
		4	0	Cmd_hysteresis	Discriminator hysteresis.
		3-0	0000	NC	Not Connected.
65	0	7	1	EN_BG	Enable of bandgap. Default is enabled(1)
		6	0	PP_BG	Power pulsing of bandgap. Default is not power pulsed(0).
		5-0	000000	NC	Not Connected

Address	SubAdd	Bit#	Default value	Name	Description
65	1	7	1	EN_10bDAC	Enable 10-bit threshold DAC. Default is enabled(1).
		6	0	PP_10bDAC	Power pulsing of 10-bit threshold DAC. Default is not power pulsed(0).
		5-2	0000	NC	Not Connected
		1-0	01	DAC_threshold[9:8]	MSB DAC values. Default is 01
	2	7-0	11011000	DAC_threshold[7:0]	LSB DAC values. Default is 11011000.
66	0	7-4	0100	EN_CLPS[0:3]	CLPS buffer size trimming. Default value is 0100
		3-0	0000	EN_pE[0:3]	CLPS pre-emphasis trimming. Default value is 0000
	1	7-6	00	pE_delay[0:1]	CLPS pre-emphasis delay trimming. Default value is 00.
		5-0	0	NC	Not Connected.
	2	7	1	EN_RX	Enable LVDS of receiver for ValEvt. Default is enabled(1)
		6	0	PP_RX	Power pulsing of LVDS receiver for ValEvt. Default is not power pulsed(0)
		5	1	Forced_ValEvt	Internal ValEvt. Bypass of external is effective when EN_RX = 0.
		4-0	00000	NC	NotConnected
67	0	7	1	EN_probe	Enable of analogue probe. Default is enabled(1).
		6	0	PP_probe	Power pulsing of analogue probe. Default is not power pulsed(0).
		5-3	000	NC	Not Connected.
		2-0	100	MillerComp[2:0]	Probe amplifier compensation capacitance trimming. Default is 100. Range : 0-700 fF. Step : 100 fF. Default : 400 fF.
	1	7-6	10	Ibi_probe[1:0]	Input bias of probe amplifier. Default is 10. 00 = 20 µA. 01 = 30 µA. 10 = 40 µA. 11 = 80 µA.
		5-0	100000	Ib0_probe[5:0]	Output bias of probe amplifier. Default is 100000. Range: 0-38 µA. Step : 0.6 µA. Default : 20 µA.

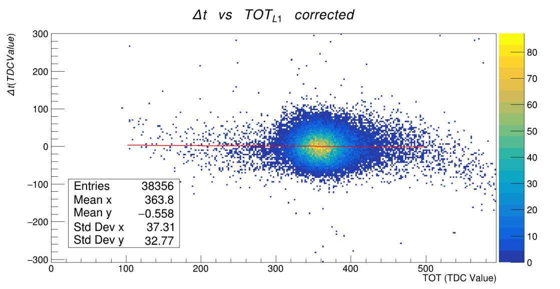
Appendix B

Data analysis histograms

B.1 Time Slewing correction histograms

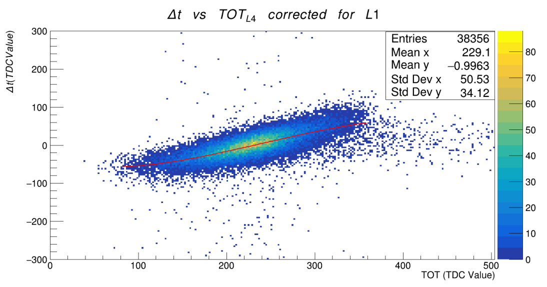


(a) Time difference versus head LGAD detector (L1) TOT.

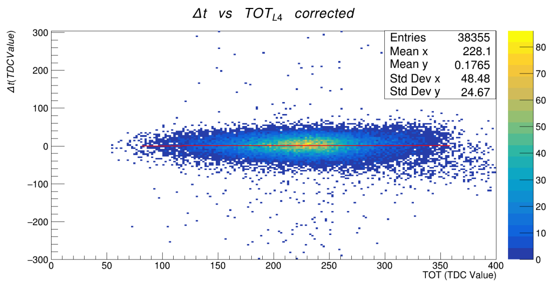


(b) Time difference versus head LGAD detector (L1) TOT after correction.

Figure B.1: *2D histograms of the time of flight measure versus the L1 TOT values before (a) and after (b) time slew correction.*

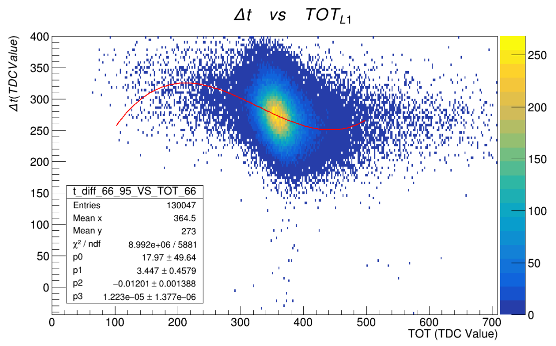


(a) Time difference versus tail LGAD detector (L4) TOT after L1 correction.

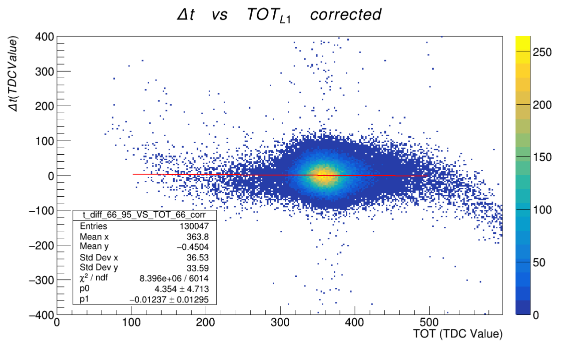


(b) Time difference versus tail LGAD detector (L4) TOT after both corrections.

Figure B.2: *2D histograms of the time of flight measure versus the L4 TOT values before (a) and after (b) time slewing correction.*

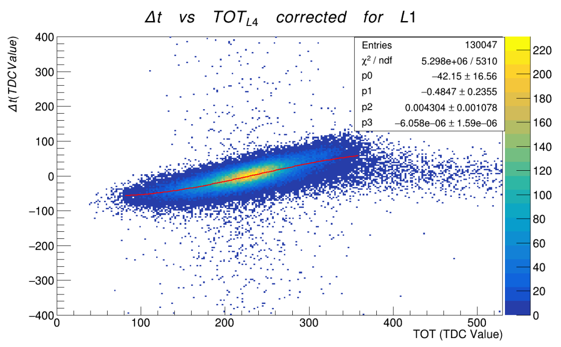


(a) Time difference versus head LGAD detector (L1) TOT.

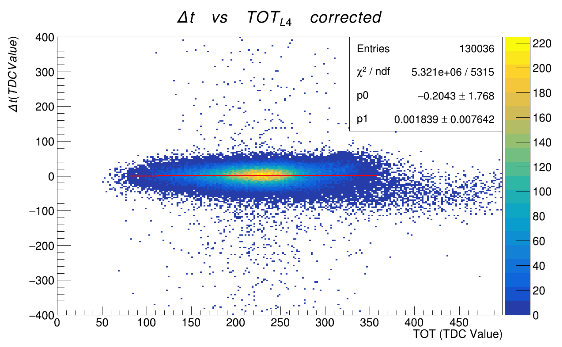


(b) Time difference versus head LGAD detector (L1) TOT after correction.

Figure B.3: *2D histograms of the time of flight measure versus the L1 TOT values before (a) and after (b) time slewing correction.*

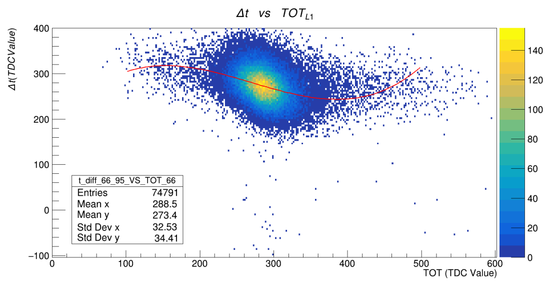


(a) Time difference versus tail LGAD detector (L4) TOT after L1 correction.

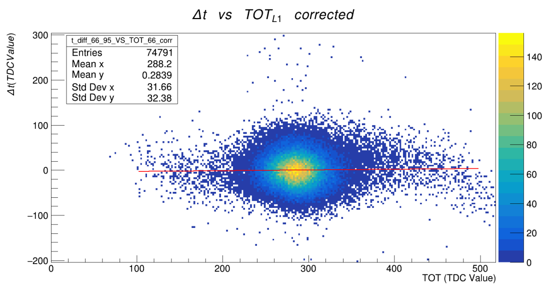


(b) Time difference versus tail LGAD detector (L4) TOT after both corrections.

Figure B.4: *2D histograms of the time of flight measure versus the L4 TOT values before (a) and after (b) time slewing correction.*

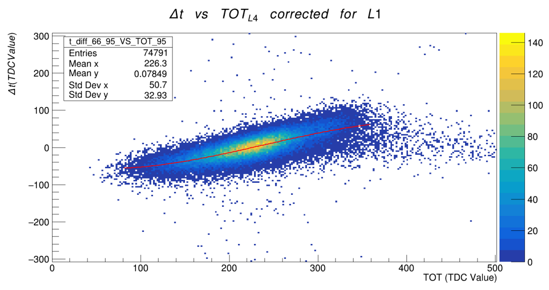


(a) Time difference versus head LGAD detector (L1) TOT.

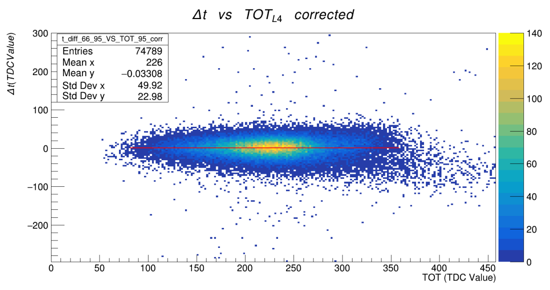


(b) Time difference versus head LGAD detector (L1) TOT after correction.

Figure B.5: *2D histograms of the time of flight measure versus the L1 TOT values before (a) and after (b) time slewing correction.*

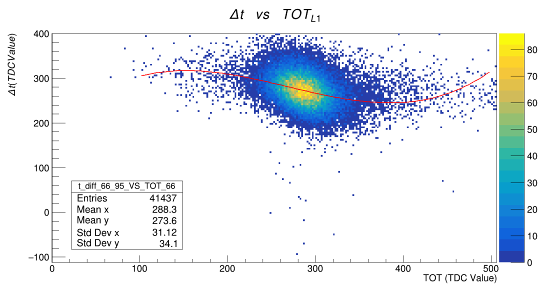


(a) Time difference versus tail LGAD detector (L4) TOT after L1 correction.

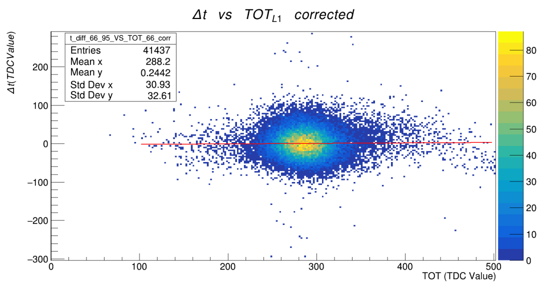


(b) Time difference versus tail LGAD detector (L4) TOT after both corrections.

Figure B.6: *2D histograms of the time of flight measure versus the L4 TOT values before (a) and after (b) time slewing correction.*

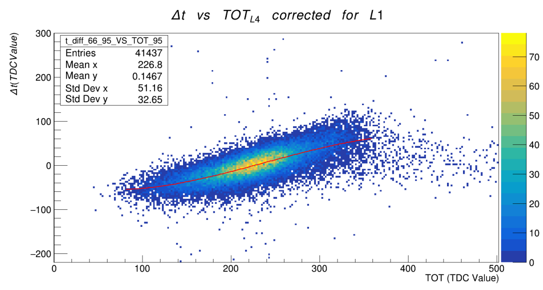


(a) Time difference versus head LGAD detector (L1) TOT.

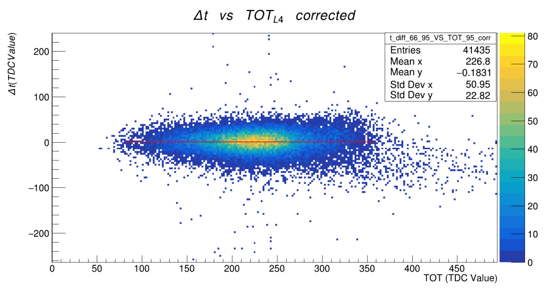


(b) Time difference versus head LGAD detector (L1) TOT after correction.

Figure B.7: *2D histograms of the time of flight measure versus the L1 TOT values before (a) and after (b) time slewing correction.*

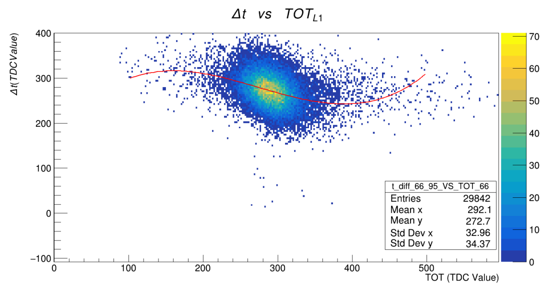


(a) Time difference versus tail LGAD detector (L4) TOT after L1 correction.

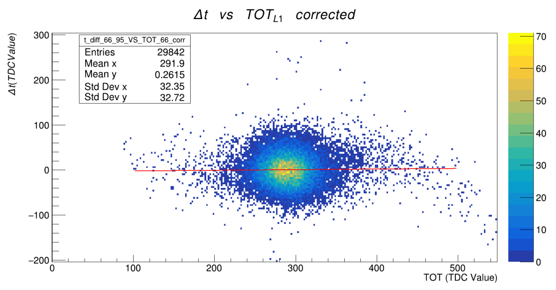


(b) Time difference versus tail LGAD detector (L4) TOT after both corrections.

Figure B.8: *2D histograms of the time of flight measure versus the L4 TOT values before (a) and after (b) time slewing correction.*

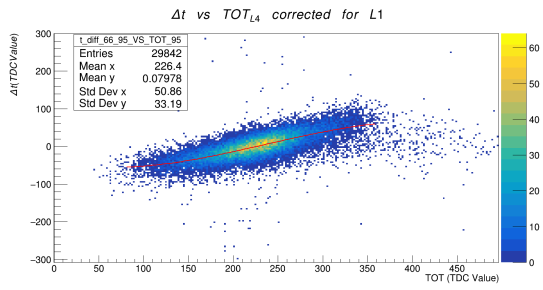


(a) Time difference versus head LGAD detector (L1) TOT.

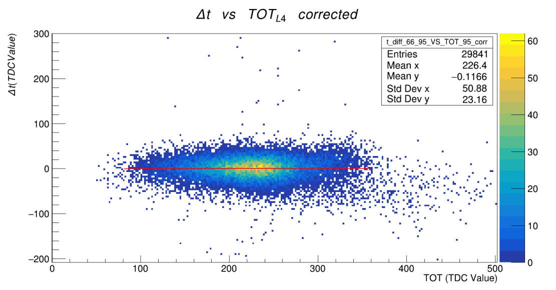


(b) Time difference versus head LGAD detector (L1) TOT after correction.

Figure B.9: *2D histograms of the time of flight measure versus the L1 TOT values before (a) and after (b) time slewing correction.*

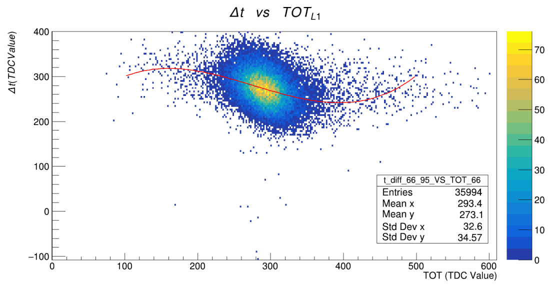


(a) Time difference versus tail LGAD detector (L4) TOT after L1 correction.

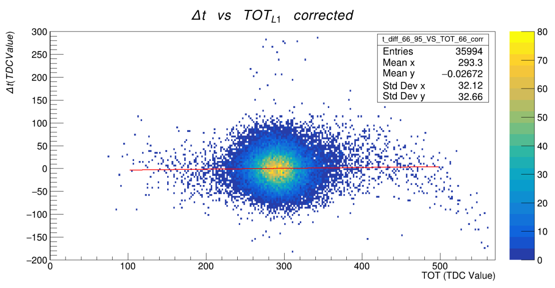


(b) Time difference versus tail LGAD detector (L4) TOT after both corrections.

Figure B.10: *2D histograms of the time of flight measure versus the L4 TOT values before (a) and after (b) time slewing correction.*

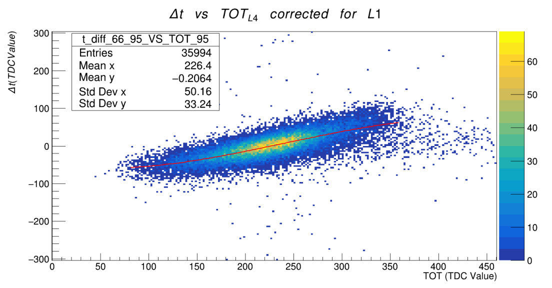


(a) Time difference versus head LGAD detector (L1) TOT.

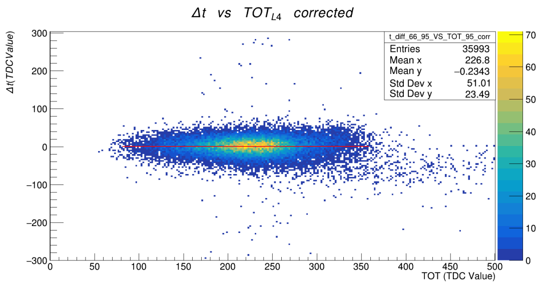


(b) Time difference versus head LGAD detector (L1) TOT after correction.

Figure B.11: *2D histograms of the time of flight measure versus the L1 TOT values before (a) and after (b) time slewing correction.*



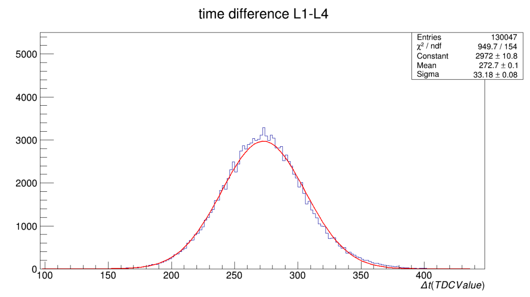
(a) Time difference versus tail LGAD detector (L4) TOT after L1 correction.



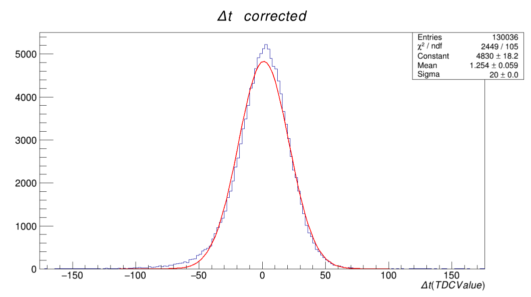
(b) Time difference versus tail LGAD detector (L4) TOT after both corrections.

Figure B.12: *2D histograms of the time of flight measure versus the L4 TOT values before (a) and after (b) time slewing correction.*

B.2 Time resolution histograms

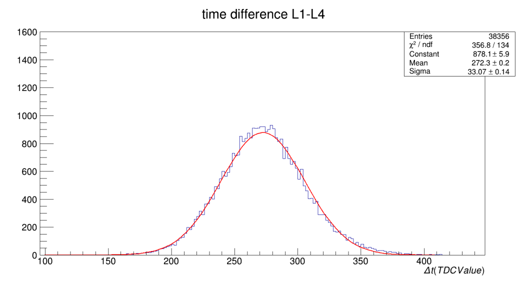


(a) Time difference distribution before time slewing correction.

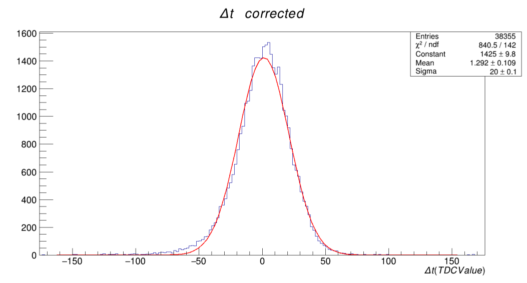


(b) Time difference distribution after time slewing corrections.

Figure B.13: *Histograms of the Time Of Flight distribution before (a) and after (b) time slewing corrections of run 579.*

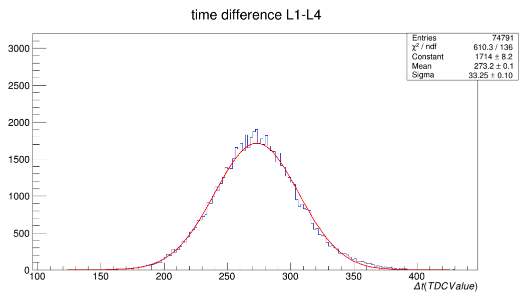


(a) Time difference distribution before time slewing correction.

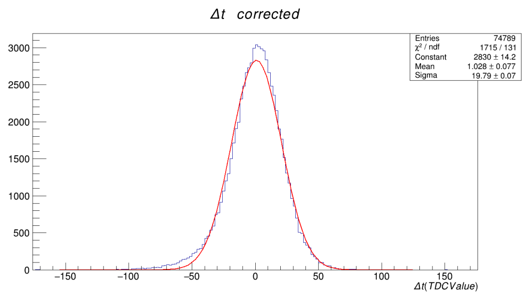


(b) Time difference distribution after time slewing corrections.

Figure B.14: *Histograms of the Time Of Flight distribution before (a) and after (b) time slewing corrections of run 582.*

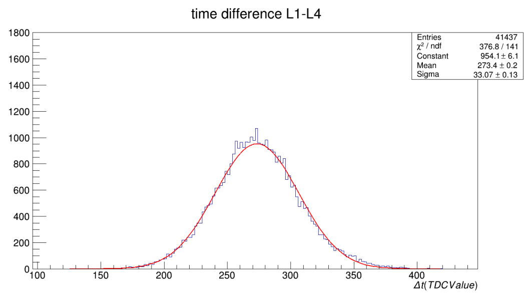


(a) Time difference distribution before time slewing correction.

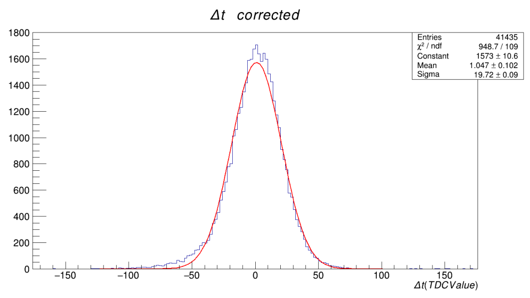


(b) Time difference distribution after time slewing corrections.

Figure B.15: *Histograms of the Time Of Flight distribution before (a) and after (b) time slewing corrections of run 583.*

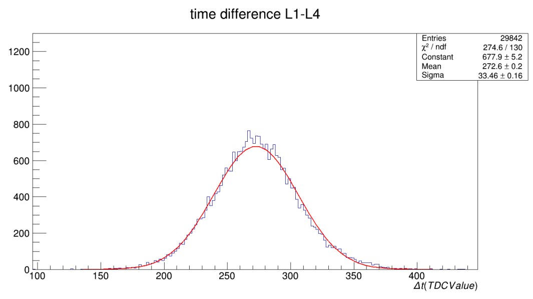


(a) Time difference distribution before time slewing correction.

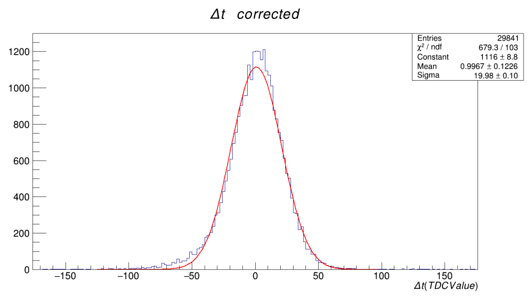


(b) Time difference distribution after time slewing corrections.

Figure B.16: *Histograms of the Time Of Flight distribution before (a) and after (b) time slewing corrections of run 584.*

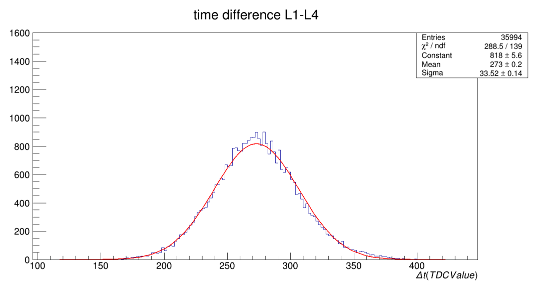


(a) Time difference distribution before time slewing correction.

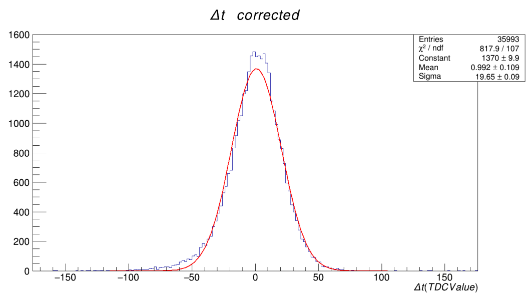


(b) Time difference distribution after time slewing corrections.

Figure B.17: *Histograms of the Time Of Flight distribution before (a) and after (b) time slewing corrections of run 585.*



(a) Time difference distribution before time slewing correction.



(b) Time difference distribution after time slewing corrections.

Figure B.18: *Histograms of the Time Of Flight distribution before (a) and after (b) time slewing corrections of run 586.*