



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Software Architecture for High-Throughput TCSPC Multichannel System: Reaching 2.56 Gbps in Time-Tagging mode measurements

MASTER'S DEGREE THESIS IN
COMPUTER SCIENCE AND ENGINEERING

Author: **Marco Fumagalli**

Student ID: 996112

Advisor: Prof.ssa Giulia Acconcia

Co-advisors: Dott. Ing. Piergiorgio Daniele, Dott. Ing. Gennaro Fratta

Academic Year: 2024-25

Abstract

Data acquisition through Single Photon Avalanche Diode (SPAD) detectors is radically changing sensors and quantum imaging, allowing information with high spatial and temporal resolution to be obtained even in extremely weak light signal conditions. The use of quantum methodologies allows researchers to optimize the use of the statistics associated with individual photons to reduce acquisition times and improve the quality of the information obtained. All this finds particular use in life-science applications (biomedical analysis techniques and micro and nano analytical genetic diagnostics) and in the acquisition of images and videos at very high sensitivity and speed, used for two-dimensional spectroscopy and for three-dimensional mapping of objects and scenes using the time of flight of individual photons.

The scientific collaboration between Politecnico di Milano, Politecnico di Bari and Agenzia Spaziale Italiana (ASI) fits into this context. Within this technological framework, this thesis work focuses on the development of advanced software for the management of data acquired by a 64-channel electronic system. The main objective is to fully exploit the bandwidth of USB 3.0 Superspeed to implement the time-tagging technique in the context of Time Correlated Single Photon Counting (TCSPC). Through a software architecture organized into classes, the program will offer a flexible interface for configuring the data received, allowing the user to alternate between the "histogram" mode for real-time graphical visualization and the "time-tagging" mode for detailed analysis of the arrival time of each photon.

In order to keep up with the continuous evolution of research, in addition to optimizing acquisition performance, the software will introduce a more efficient programming paradigm compared to the current system adopted in the laboratory, with the aim of optimizing the code writing method in use for decades. The adoption of object-oriented parallel programming will improve the modularity, scalability and maintainability of the code, facilitating any future extensions of the system and ensuring greater efficiency in the management of the acquired data.

Keywords: TCSPC, SPAD, time-tagging, object-oriented

Abstract in Italiano

L'acquisizione di dati tramite rivelatori a singolo fotone sta rivoluzionando la sensoristica e l'imaging quantistico, permettendo di ottenere informazioni con elevata risoluzione spaziale e temporale anche in condizioni di segnale luminoso estremamente debole. L'impiego di metodologie quantistiche consente di ottimizzare l'uso della statistica associata ai singoli fotoni per ridurre i tempi di acquisizione e migliorare la qualità dell'informazione ottenuta. Tutto ciò trova particolare impiego in applicazioni di tipo life-science (tecniche di analisi biomedica e diagnostica genetica micro e nano analitica) e nell'acquisizione di immagini e video ad elevatissima sensibilità e velocità, adoperate per la spettroscopia bidimensionale 2D e per la mappatura tridimensionale 3D di oggetti e scene mediante il tempo di volo dei singoli fotoni.

In questo contesto si inserisce la collaborazione scientifica tra il Politecnico di Milano, il Politecnico di Bari e l'Agenzia Spaziale Italiana (ASI), ed all'interno di questa cornice tecnologica, il presente lavoro di tesi si concentra sullo sviluppo di un software avanzato per la gestione dei dati acquisiti da un sistema elettronico a 64 canali. L'obiettivo principale è sfruttare interamente la larghezza di banda della USB 3.0 Superspeed per implementare la tecnica del time-tagging nel contesto del Time Correlated Single Photon Counting (TCSPC). Attraverso un'architettura software organizzata in classi, il programma offrirà un'interfaccia flessibile per la configurazione dei dati ricevuti, permettendo di alternare la modalità "istogramma" per la visualizzazione di grafici in tempo reale e la modalità "time-tagging" per l'analisi dettagliata del tempo di arrivo di ogni fotone.

Al fine di stare al passo con la continua evoluzione della ricerca, oltre all'ottimizzazione delle prestazioni di acquisizione, il software introdurrà un paradigma di programmazione più efficiente rispetto ai sistemi attuali, col proposito di ottimizzare il metodo di scrittura di codice in uso da decenni. L'adozione della programmazione orientata agli oggetti migliorerà la modularità, la scalabilità e la manutenibilità del codice, facilitando eventuali estensioni future del sistema e garantendo una maggiore efficienza nella gestione dei dati acquisiti.

Parole chiave: TCSPC, SPAD, time-tagging, orientato agli oggetti

Contents

Abstract	i
Abstract in Italiano	iii
Contents	v
Introduction	1
1 Time-Related Single Photon Counting	5
1.1 TCSPC Technique Overview	5
1.1.1 Historical background	7
1.1.2 Essential Components and Functional Blocks	8
1.1.3 TCSPC Standard Configuration	9
1.1.3.1 Single Photon Avalanche Diode	10
1.1.3.2 Time to Amplitude Converter	11
1.1.3.3 Reversed Start-Stop Triggering Configuration	12
1.2 Applications of TCSPC	13
1.2.1 Time-resolved fluorescence decay analysis	13
1.2.2 Photon-Based Tomographic and 3D Imaging	14
1.2.3 Time-of-Flight Ranging and Depth Estimation	16
1.2.4 Photon Coincidence Detection in Quantum Systems	16
1.3 Architecture of the ASI TCSPC implementation	18
1.3.1 SPAD Matrix	19
1.3.2 Detection module	19
1.3.3 Acquisition module	20
1.3.3.1 TAC Integration and Analog-to-Digital Conversion	21
1.3.3.2 Embedded Microcontroller for Peripheral Coordination	22
1.3.3.3 FPGA Logic for High-Speed Time Stamping	23
1.3.3.4 FTDI FT601 USB 3.0 Data Bridge	24

1.3.4	Power delivery and regulation	26
1.3.5	Communication module	26
1.4	Considered Approaches in TCSPC Implementation	28
1.4.1	Histogram Construction	28
1.4.2	Time-Tagging and Event Logging Mechanism	29
1.5	Data Throughput and Real-Time Processing Challenges	32
2	Software of the ASI module	35
2.1	Object-Oriented Programming Paradigm: Principles and Methodologies . .	36
2.1.1	Conceptual foundations of OOP	36
2.1.2	Properties of OOP	37
2.1.2.1	Encapsulation	37
2.1.2.2	Inheritance	38
2.1.2.3	Polymorphism	39
2.1.2.4	Abstraction	40
2.1.3	Advantages of OOP	41
2.1.3.1	Modularity	42
2.1.3.2	Scalability	43
2.1.3.3	Maintainability	43
2.2	Concurrent Programming and Thread-Based Architectures	44
2.2.1	Fundamentals of concurrency in software systems	44
2.2.2	Comparative analysis: sequential vs. parallel execution	44
2.2.3	Thread management and synchronization in C#	45
2.2.4	Practical implementation: the DataReaderTimeTag thread	46
2.3	Design and Engineering of the New Acquisition Software	47
2.3.1	Software Requirements and Performance Constraints	47
2.3.2	Architectural Decisions	48
2.3.2.1	Adoption of Object-Oriented Design Principles	48
2.3.2.2	FTDI FT601 Interface and Data Transmission Libraries	49
2.3.2.3	Scalable threading model for future parallelism	50
2.3.3	Class architecture and system overview	50
2.4	Implementation Details	53
2.4.1	Performance-oriented implementation choices	53
2.4.2	Class-level design and implementation	54
2.4.2.1	Form1	54
2.4.2.2	Form2	59
2.4.2.3	FTDIDController	64

2.4.2.4	WorkerThread	66
2.4.2.5	BufferManager	70
2.4.2.6	ConfHisto	72
2.4.3	Robustness: exception handling and error recovery	75
3	System Validation Framework and Future Directions	77
3.1	Hardware Measurement Setup	78
3.2	Software Verification Procedure	81
3.3	Prospects for Evolution and Performance Improvements	86
	Bibliography	89
	List of Figures	93
	List of Tables	95

Introduction

Single-photon detection has become a cornerstone of modern scientific and technological advancements, enabling unprecedented precision in measuring light at the quantum level. This capability is not only fundamental to understanding light-matter interactions but also critical for developing cutting-edge applications across diverse fields. For instance, in bioengineering, single-photon detectors are used in techniques like Fluorescence Lifetime Imaging Microscopy (FLIM) and Positron Emission Tomography (PET) to study cellular processes, diagnose diseases such as cancer and intervene surgically in real-time (exploiting Augmented Reality software) by analysing the fluorescence decay of biomarkers. In electronics and automation, single-photon detectors are integral to LiDAR systems for autonomous vehicles, enabling high-resolution 3D mapping and obstacle detection. Similarly, in quantum information technology, single-photon detection is essential for implementing Quantum Key Distribution (QKD) protocols, which provide unbreakable encryption by leveraging the principles of quantum mechanics.

Despite these advancements, traditional single-photon detection technologies, such as photomultiplier tubes (PMTs) and superconducting nanowire single-photon detectors (SNSPDs), face significant limitations. PMTs, while highly sensitive, are bulky, fragile, and require high operating voltages, making them unsuitable for compact or portable systems. SNSPDs, on the other hand, offer excellent timing resolution and low noise but require cryogenic cooling, which increases system complexity and cost. These drawbacks and the growing demand for single-photon detection in real-time have driven the development of Single Photon Avalanche Diodes (SPADs), which combine high sensitivity, compact size, and room-temperature operation, making them ideal for integration into modern systems. Indeed, their ability to precisely timestamp photons and operate in fast-gated modes has made them a preferred choice for applications requiring high temporal resolution and low noise.

In particular, SPADs are semiconductor devices that operate in reverse bias above their breakdown voltage, a region known as Geiger mode. In this mode, a single photon absorbed in the active region can trigger an avalanche multiplication process, generating a measurable current pulse. This mechanism allows SPADs to detect individual pho-

tons with high timing resolution, typically in the picosecond range. Key advantages of SPADs include their compact size, compatibility with Complementary Metal-Oxide-Semiconductor (CMOS) fabrication processes, and ability to operate at room temperature. These features make SPADs highly versatile and suitable for integration into large arrays, enabling applications such as high-resolution imaging and parallel photon counting. However, SPADs have some limitations. After detecting a photon, the device enters a "dead time" during which it cannot detect another photon, potentially leading to missed events in high-flux scenarios. Additionally, SPADs are susceptible to noise sources such as dark counts (false detections caused by thermal generation of carriers) and after pulsing (delayed avalanches triggered by trapped carriers). Modern SPAD designs incorporate quenching circuits and advanced cooling techniques to mitigate these effects, but careful system design and signal processing are still required to achieve optimal performance.

However, to fully exploit the potential of SPAD-based systems, it is crucial to develop efficient and scalable software frameworks that can handle the high data rates and complexity of modern photon detection systems.

In order to contextualize the scope of this thesis and the systems considered, it is necessary to introduce the Time Correlated Single Photon Counting technique and its two main variants. This procedure allows to measure fluorescence decays in the time domain, and consists in exciting periodically a sample with a laser and recording the timestamp of the detected photons within each excitation period. Since the luminescent signal has a low intensity, the probability to reveal a photon inside the period is way less than one, and so it's not necessary to consider the possibility of revealing more photons in a single period. After many iterations, a histogram representing the reconstruction of the luminescent intensity emitted by the sample as a function of time is built. An alternative acquisition method in TCSPC systems is called "time-tagging", which does not build photon distributions but instead stores detailed information for each individual photon. Specifically, it records the time interval between the laser sync signal and the photon detection ("micro-time"), as well as a global timestamp ("macro-time") referenced to a common start event, such as the system power-on. This flexibility is particularly beneficial in experimental setups requiring adaptive measurement strategies driven by the tracking of the waveform evolution over time.

Back to SPAD-based systems, their evolution places stringent demands on software architectures. Efficient photon counting at very high rates, while maintaining accuracy in timestamping and histogram visualization, necessitates a software framework optimized for computational efficiency, low-latency data handling, and scalability for future multi-channel systems.

In the present workflow, the data generated by the SPADs is first processed by an FPGA, which encodes the incoming electrical pulses into a digital format readable by the host system. This encoded data is then transferred to the PC via the FTDI FT601 interface using the USB 3.0 protocol. The logic for the two acquisition modes, histogram visualization and time-tagging, is implemented entirely in the FPGA through dedicated bitstreams.

This thesis project presents an object-oriented software solution designed to meet the mentioned challenges, integrating real-time histogram generation and high-bandwidth time-tagging acquisition. By leveraging principles such as encapsulation, modularity, and abstraction, the software design achieves a clear separation of concerns, enabling efficient handling of the main data stream from SPAD-based detectors. The modular design not only simplifies maintenance and extensibility but also facilitates adaptation to evolving experimental configurations. Additionally, the producer-consumer internal pattern forecasts an ideal separation between data acquisition (producer thread) and data writing (consumer thread), enabling continuous data flow and maximizing reading throughput (up to 320 MB/s).

In conclusion, this final project addresses the growing need for efficient software architectures in SPAD-based photon detection systems, particularly in the context of high-throughput time-tagging and histogram acquisition. While the core detection and encoding functionalities are implemented at the hardware level within the FPGA, the software developed here plays a crucial role in ensuring data is received, organized, and made readily available for analysis and visualization. By adopting an object-oriented and modular design, the software provides a reliable and scalable solution tailored to the laboratory's needs, capable of exploiting all the USB 3.0 bandwidth without losses. This work not only improves upon the limitations of previous implementations used in the same research environment, but also lays the foundation for future developments.

This thesis is divided into three chapters: the first one introduces the principles of Time-Correlated Single Photon Counting (TCSPC), its applications, and the architecture of the custom FPGA-based system developed within the ASI project. It details both histogramming and time-tagging acquisition modes and the related data handling challenges; the second chapter focuses on the software developed for data acquisition, explaining the object-oriented and multi-threaded design used to interface with the FTDI FT601 chip and manage high-throughput data streams; the final chapter presents configuration tests conducted to validate the system's performance in time-tagging mode and discusses potential avenues for future improvements.

1 | Time-Correlated Single Photon Counting

1.1. TCSPC Technique Overview

Time-Correlated Single Photon Counting (TCSPC) is a highly sensitive and precise photon detection technique widely employed in fields requiring ultrafast temporal resolution, such as fluorescence lifetime imaging microscopy (FLIM), time-resolved spectroscopy, quantum optics [9], and photon correlation studies. Basically, TCSPC measures the arrival times of individual photons emitted by a sample in response to a periodic excitation source, usually a pulsed laser with a known repetition rate and pulse duration [4][38].

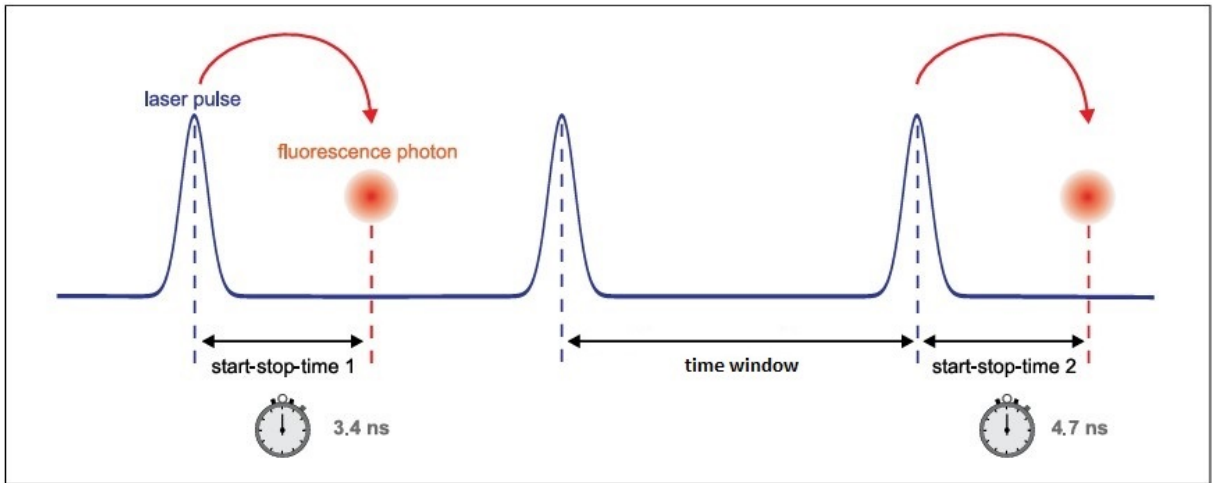


Figure 1.1: Measurement of start-stop times in time-resolved fluorescence measurement with TCSPC. [37].

The core principle relies on the detection of single photons within very short time windows synchronized with the excitation pulses. The emitted light signal is characterized by extremely low intensity combined with a high repetition frequency of excitation events and, consequently, the probability of detecting more than one light quantum per excitation period is deliberately kept much lower than one, typically in the range of 0.05 to 0.1

photons per period, to ensure operation in the so-called “single-photon regime”. This system is critical to avoid “pile-up” distortions, where multiple photons arriving in the same period can cause nonlinear distortions in the recorded histogram [4][33].

Figure 1 shows the photon detection timeline. In the example, fluorescence is excited repetitively by short laser pulses, and the time between excitation and emission is measured by electronics that act like a stopwatch. Often, no photon is detected in a cycle, such as after the second pulse, because photon detection follows the probabilistic nature of quantum physics. Whether a photon appears or not is entirely random, and so are the individual time measurements.

During data acquisition, TCSPC systems repetitively detect and time-stamp photon arrivals over millions of excitation cycles. These arrival times are then accumulated into a high-resolution histogram as it is displayed in Figure 1.2, representing the photon count as a function of time within each excitation period. Since the instantaneous probability of photon detection is proportional to the incident photon flux, the histogram effectively reconstructs the temporal decay profile or waveform of the sample’s emitted light and this enables quantitative characterization of fluorescence lifetimes, excited state dynamics, and other time-dependent optical phenomena with picosecond temporal precision.

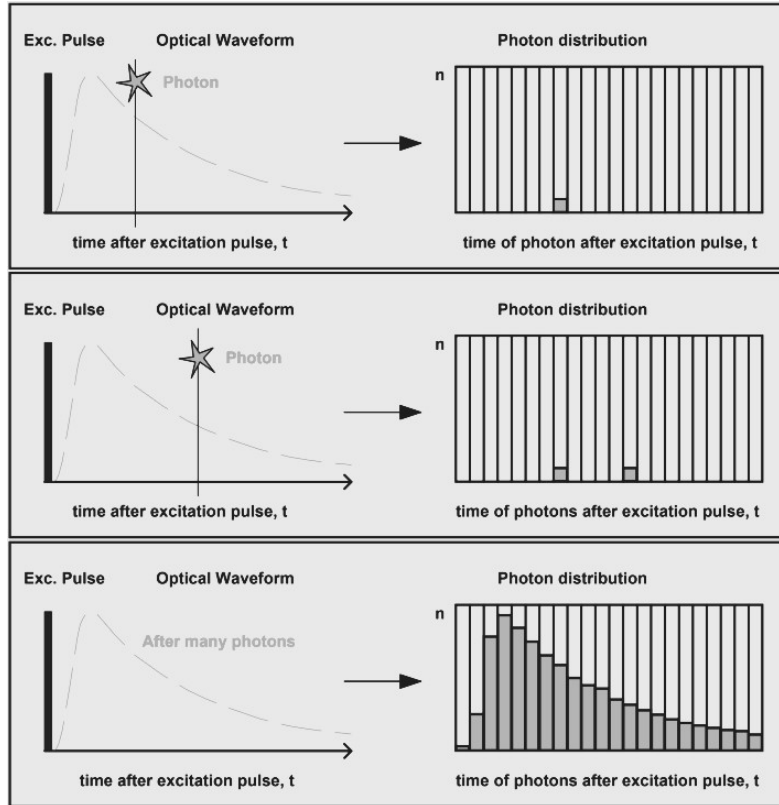


Figure 1.2: Example of an histogram construction [33].

One of the remarkable features of TCSPC is its exceptional temporal resolution, which can reach tens of picoseconds or better.

The detector and associated electronics also exhibit a “dead time” following each photon detection event, during which they are temporarily unable to register subsequent photons. This dead time, typically on the order of 100 nanoseconds in modern systems, limits the maximum count rate achievable (around 10 million counts per second, Mcps). Operating beyond this rate increases the likelihood of missing photons, leading to data loss and distortions.

In summary, TCSPC’s combination of single-photon sensitivity, high temporal precision, and efficient data accumulation makes it an indispensable tool for investigating fast photophysical processes, fluorescence lifetimes, and quantum optical effects with unparalleled accuracy. The following sections will explore the key applications and instrumentation architectures that leverage the unique capabilities of TCSPC technology.

This section provides an overview of the fundamental principles and components of TCSPC systems, starting with an historical perspective, tracing the evolution of photon timing methods from early developments to the sophisticated systems employed today. Then, the main components and functional blocks that constitute a typical TCSPC setup are described, highlighting their roles in accurately detecting and processing single-photon events[4][18].

Particular attention is given to Single Photon Avalanche Diodes (SPADs), which serve as key detectors in modern TCSPC implementations in this Politecnico di Milano laboratory, due to their excellent timing characteristics and detection efficiency. The discussion then extends to the possible configurations of this technique, providing context for the advances in current system architectures.

1.1.1. Historical background

The development of Time-Correlated Single-Photon Counting can be traced back to the mid-20th century, when researchers began exploring ways to detect individual photons. Heinrich Hertz drew the conclusion that to observe a single photon, one requires to convert its signal into an electric one, and that can be done exploiting the photoelectric effect and the secondary emission principle by Austin and Stark. The capability to detect single photons means that it is also possible to timestamp their arrival, similar to timing the arrival of particles. Indeed, the origin of TCSPC lies in particle physics and was applied to nuclear spectroscopy to measure radiation decay times. These investigations laid the

groundwork for the development of single-photon detection techniques [4][10][24].

In the 1970s, advances in electronics and photonics enabled the first practical implementation of TCSPC: early systems used photomultiplier tubes (PMTs) as single-photon detectors and analog electronics for time measurement and were used primarily in fluorescence lifetime studies, where they provided unprecedented insights into the dynamics of excited states in molecules.

The 1980s and 1990s saw significant improvements in TCSPC technology, driven by the development of faster and more sensitive detectors, such as microchannel plate PMTs and, later, Single Photon Avalanche Diodes (SPADs). The, the introduction of digital electronics and microprocessors further enhanced the precision and versatility of TCSPC systems, enabling their use in a wider range of applications, including time-resolved spectroscopy and quantum optics.

In recent decades, the integration of TCSPC with modern technologies, such as Field-Programmable Gate Arrays (FPGAs) and fast data acquisition systems, has revolutionized the research field. These advancements have enabled real-time, high-speed TCSPC with picosecond timing resolution, opening up new possibilities in biomedical imaging, 3D sensing, and quantum communication, as was presented in the first section [4].

1.1.2. Essential Components and Functional Blocks

Every time-correlated single photon counting instrument comprises several essential sub-systems that operate together to achieve precise photon timing measurements [4][33].

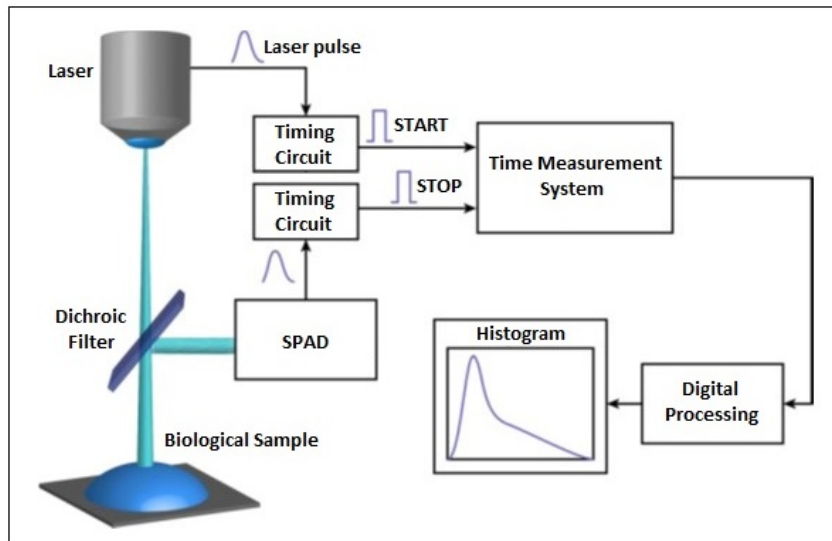


Figure 1.3: Example of a TCSPC conventional setup [33].

- **Excitation Source:** a pulsed laser or light source is used to periodically excite the considered sample. The rate of the laser, synchronized with a timing circuit, determines the time window for photon detection.
- **Single-Photon Detector:** devices such as Single Photon Avalanche Diodes (SPADs) are used to detect individual photons. SPADs are preferred over PMTs and other technologies because of their high sensitivity, fast response, and compatibility with modern integrated circuits.
- **Time Measurement Units:** in the standard configuration, these two components generates two impulses: one is sent contemporarily with the laser pulse (START), the other one is sent when the photon is detected (STOP). The difference between these two signals is then retrieved using a Time to Amplitude Converter (TAC) and transmitted to an Analog to Digital Converter (ADC) which digitizes it. The ADC output provides the digital equivalent of the photon detection time with exceptional precision.
- **Data Acquisition and Processing:** the recorded photon arrival times are processed to build a histogram or a time-tagged dataset, which represents the temporal decay profile of the emitted light. This experimental data is then analyzed to extract key characteristics such as decay constants, lifetimes, or any other relevant parameters, providing insights into the material or system being studied. In this project, the processing part is managed by the FPGA and the instradation to the PC is managed by the FTDI601 and the USB 3.0.

1.1.3. TCSPC Standard Configuration

As mentioned in the precedent bullet point, the standard TCSPC configuration relies on measuring the time delay between a reference signal from the laser pulse and a detection signal from the SPAD. In the classical approach, this delay is recorded for each detected photon and accumulated over many excitation cycles to build an histogram which provides a precise temporal profile of photon arrivals, enabling high-resolution analysis of the optical signal [18][33].

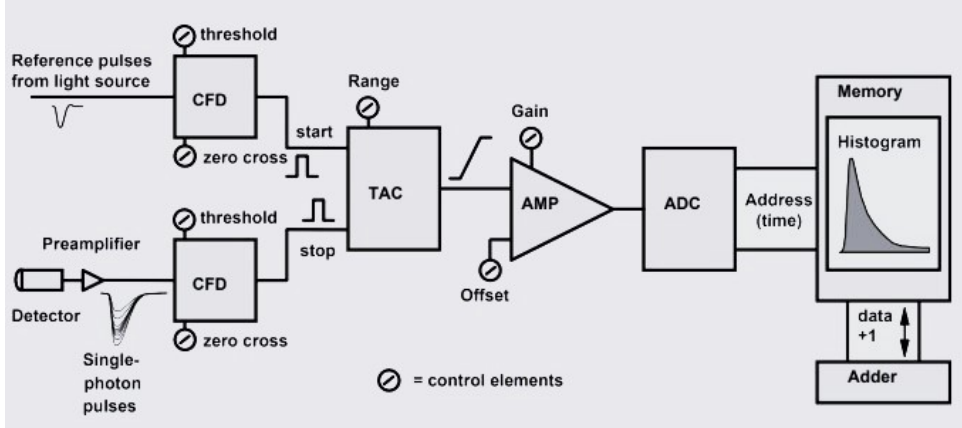


Figure 1.4: Architecture of a classic TCSPC device [5].

CFD = Constant Fraction Discriminator

TAC = Time to Amplitude Converter

ADC = Analog to Digital Converter

1.1.3.1. Single Photon Avalanche Diode

One of the best-performing detectors in modern TCSPC techniques is the Single Photon Avalanche Diode (SPAD), due to its high Photon Detection Efficiency (PDE) coefficient and excellent time resolution. These contemporary devices are characterized by low dark count rate, high efficiency in photon revealing and excellent temporal resolution [4].

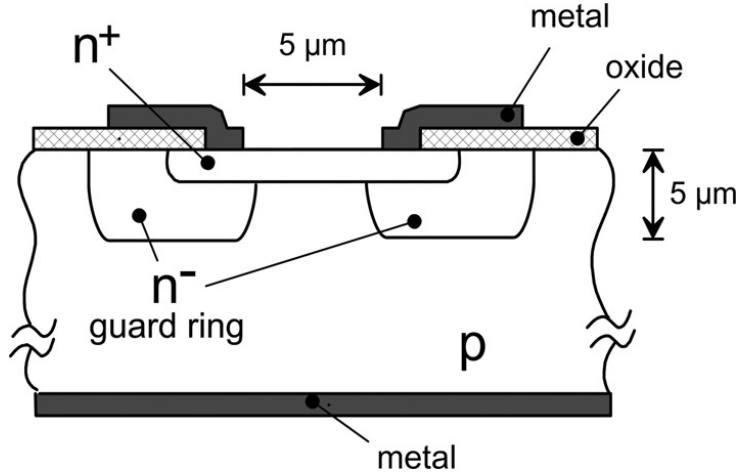


Figure 1.5: Simplified schema of a SPAD [14].

When a photon strikes the active area of the p-n junction, it creates with high probability an electron-hole pair. The electric field accelerates these carriers, triggering impact ionization and initiating a self-sustaining avalanche which produces a macroscopic current pulse, enabling single-photon detection. However, for every avalanche, the SPAD becomes temporarily inactive and, to reset it, the voltage must drop below the breakdown level.

This process, known as quenching (active or passive), defines the SPAD's dead time, during which no new photons can be detected.

- **Passive quenching** uses a high-value resistor in series with the SPAD, which slowly reduces the avalanche current. It has a long, undefined dead time and a high jitter.
- **Active quenching** employs a fast comparator circuit to quickly stop the avalanche and reset the SPAD. This allows for shorter and well-defined dead times, higher count rates, and lower timing jitter [10].

1.1.3.2. Time to Amplitude Converter

In conventional TCSPC systems, a Time-to-Amplitude Converter (TAC) serves as the core component for temporal measurements. When the START signal from the laser and the STOP signal from the photon detection are received, the TAC generates a linearly increasing voltage ramp whose final amplitude is directly proportional to the elapsed time interval between these two events. This analog voltage output is subsequently digitized by a high-precision Analog-to-Digital Converter (ADC) which must be very fast in order to keep the dead time of the system short. The resulting digital value serves as an address pointer to a specific bin in the histogram memory, where each address location corresponds to a discrete time channel. As photons are detected, their respective time bins are incremented, gradually building a statistical representation of the photon arrival time distribution relative to the excitation pulse[4][33].

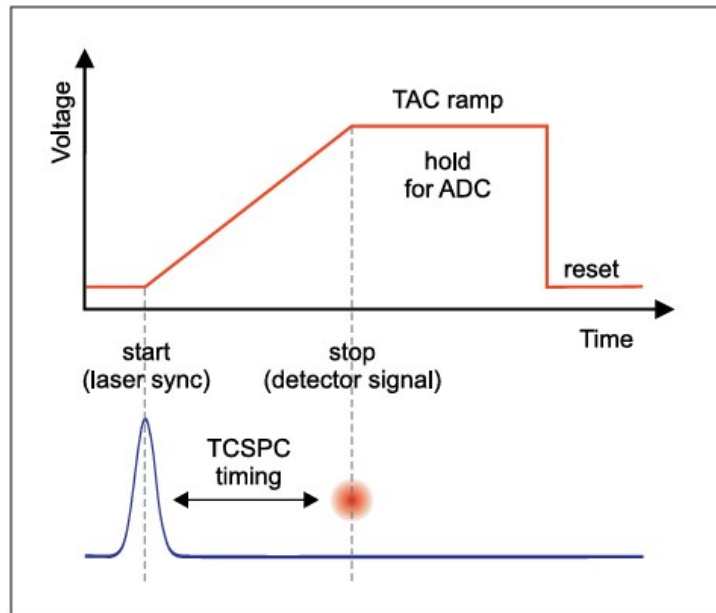


Figure 1.6: Operation principle of a TAC level (red line)[37].

1.1.3.3. Reversed Start-Stop Triggering Configuration

In standard TCSPC, the "Start" signal is triggered by the laser pulse, and the "Stop" is generated by the detected photon. However, at high repetition rates (50-100 MHz), many laser pulses occur without photon detection and this results in frequent TAC resets, making the method inefficient [18]. To overcome this, the inverted Start-Stop configuration is used. Here, the TAC is triggered when a photon is detected (Start) and stopped by the next laser pulse (Stop). This approach significantly reduces TAC resets, as it only operates at the photon detection rate rather than the laser repetition rate[33].

The FPGA's reconfigurable logic implements a time-axis inversion algorithm that can operate through either bitwise complement operations on the ADC output or through memory addressing remapping during histogram readout. This architectural flexibility allows maintaining temporal precision while adapting to different experimental configurations without hardware modification [33].

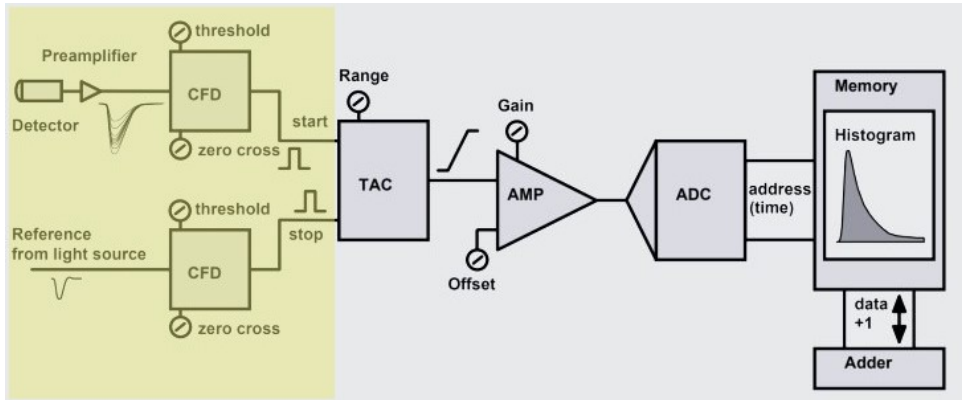


Figure 1.7: Reversed START-STOP configuration [5].

This inverted method is particularly useful for high-speed photon detection applications, ensuring more efficient and precise time measurements while minimizing unnecessary TAC resets.

1.2. Applications of TCSPC

In this section, we present a selection of key applications of TCSPC focussing focus on how this technique is exploited across various domains that benefit from high temporal resolution in photon detection. These include the analysis of fluorescence decay in life sciences, tomographic and three-dimensional imaging techniques, time-of-flight (ToF) systems for distance and depth measurement, and photon coincidence detection in quantum optics and quantum information systems. Each of these applications highlights the versatility and precision of TCSPC in handling low-intensity, time-resolved photon events across distinct scientific and technological contexts.

1.2.1. Time-resolved fluorescence decay analysis

Time-Correlated Single-Photon Counting (TCSPC) is the gold-standard technique for measuring molecular fluorescence in chemical, medical, and biological applications [20]. Figure 1.1 illustrates the main processes involved in fluorescence: upon excitation by an incident photon, a molecule reaches a higher-energy electronic state (S_1) (a), from which it can return to the ground state (S_0) by emitting a longer-wavelength photon (b), converting the energy into heat (c), or transferring it to another molecule (d). The time the molecule remains in the excited state is the excited-state lifetime. Fluorescence refers to the absorption-emission process and is characterized by a decay function, whose $1/e$ time defines the fluorescence lifetime as showed in Figure 1.2.

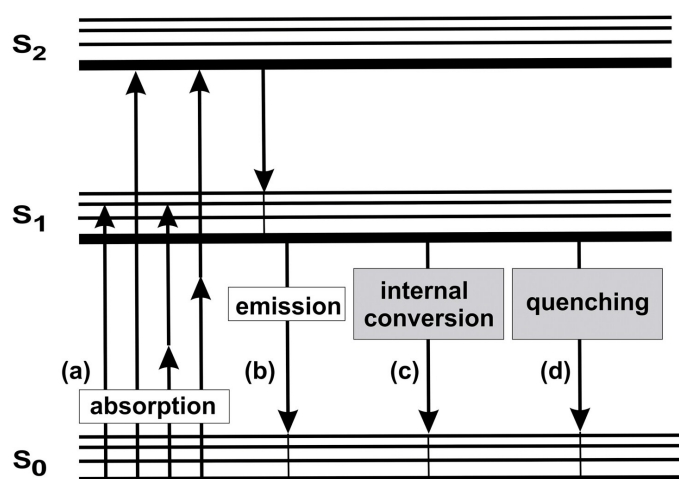


Figure 1.8: Processes involved in the excitation and emission of fluorescence [34].

The emitted fluorescence is then detected by a single-photon-sensitive detector (e.g., a photomultiplier tube or SPAD), which converts each photon into a measurable electrical

pulse. TCSPC electronics precisely timestamp these pulses relative to the excitation laser, enabling the reconstruction of the histogram of photon arrival times [4].

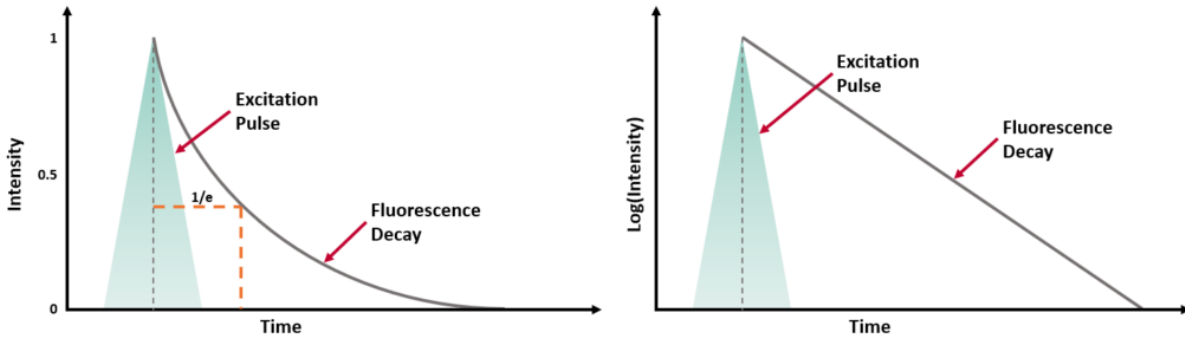


Figure 1.9: Representation of a fluorescence decay following a short excitation pulse on a (a) linear and (b) logarithmic scale [19].

This fluorescence decay curve provides direct insight into molecular properties and chemical processes: by analyzing intensity versus time (or wavelength), they are extracted decay constants revealing molecular interactions and microenvironment sensitivity, quantitative concentration data, and signatures of ongoing chemical reactions or biological processes through characteristic lifetime changes[4].

1.2.2. Photon-Based Tomographic and 3D Imaging

Optical tomography is a medical imaging technique that utilizes near-infrared (NIR) or infrared (IR) light to produce three-dimensional reconstructions mainly of biological tissues. The technique is grounded in the principle that these tissues interact with light in distinct ways depending on their composition and optical absorption properties [22][6].

The process begins with illumination from a low-power NIR/IR light source, which is safe for biological tissues [6]. This light is directed toward the sample, where it undergoes absorption, scattering, and reflection as it interacts with the tissue. An array of detectors then captures the light that emerges from the sample following these interactions and to enhance spatial resolution and depth information, measurements are typically acquired from multiple angles or positions. Computational algorithms subsequently process the collected data to reconstruct a three-dimensional map of the sample's optical properties and the resulting image reveals variations in tissue characteristics such as blood absorption, scattering coefficients, and structural density.

Optical tomography is non-invasive and offers high resolution, making it particularly suitable for imaging superficial and subsurface structures with a penetration depth of up to

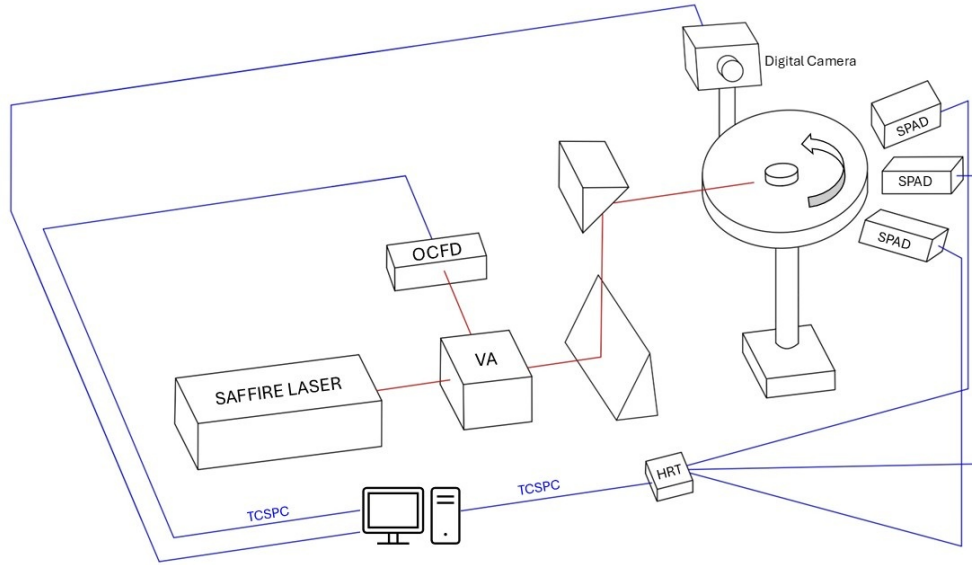


Figure 1.10: Schematic diagram of a SPAD-based DOT imager[30].

It uses a femtosecond Ti:Sapphire laser with variable attenuator, a sample mounted on motorized rotation stage, multiple SPAD detectors on a separate stage, electronic signals routed to a multi-channel TCSPC module for parallel acquisition, and a digital camera capturing sample boundaries during rotation for reconstruction.

a few centimeters. Its medical applications are diverse, including breast and prostate cancer detection via Diffuse Optical Tomography (DOT), functional brain imaging for monitoring cerebral oxygenation, and fetal health monitoring, as well as ophthalmic diagnostics when combined with techniques like optical coherence tomography (OCT) [39][32]. However, the method is constrained by light scattering in biological tissues, which limits its effectiveness for deep-tissue imaging compared to modalities such as MRI (Magnetic Resonance Imaging) or CT (Computed Tomography), using magnetic fields, gamma rays and mechanical waves [6].

SPAD-based Time-Correlated Single Photon Counting (TCSPC) systems are particularly well-suited for optical tomography due to their ultra-high temporal resolution, which enables precise time-of-flight measurements for depth profiling and, additionally, their single-photon sensitivity makes them ideal for detecting weak signals in highly scattering media [6]. Despite these advantages, the requirement for multiple detection channels can impose limitations on the achievable precision, necessitating careful system optimization to balance performance and complexity.

1.2.3. Time-of-Flight Ranging and Depth Estimation

The time-of-flight (ToF) measurement, illustrated in Figure 1.17, constitutes a fundamental technique for determining the duration required for an object or particle to traverse a defined distance [33]. The measurement protocol initiates with the generation of a precisely timed signal, typically in the form of a laser pulse or electrical impulse, which is directionally emitted toward the target object or particle and interacts with the moving target through either reflective or absorptive phenomena. A dedicated sensor system detects the reflected energy pulse and precisely records the temporal interval between emission and reception events. Through knowledge of the time-of-flight and the propagation velocity of the signal medium, one can subsequently derive either the absolute distance between the emission source and target or determine the target's velocity.

This methodology finds extensive application across multiple scientific and technological domains. In metrology, it enables precise distance measurements in laser ranging systems. Biomedical applications include non-invasive blood flow quantification through laser Doppler techniques and, in particle physics, the approach provides critical velocity measurements for charged particles.

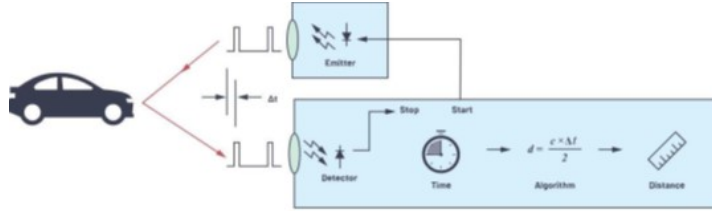


Figure 1.11: Schematic representation of a TOF system using the TCSPC technique [33].

In the end, ToF measurement plays a pivotal role in autonomous vehicle perception systems, particularly in Light Detection and Ranging (LiDAR) sensors, which are essential for real-time environmental mapping and obstacle detection. These systems rely on the precise emission of laser pulses and the accurate real-time measurement of their return time after reflection from surrounding objects.

1.2.4. Photon Coincidence Detection in Quantum Systems

One of the most emblematic quantum optical phenomena is the Hong–Ou–Mandel (HOM) effect, which reveals the fundamental nature of photon indistinguishability. The effect is typically observed when two identical photons are sent into the two input ports of a 50:50 beam splitter. Under ideal conditions, the two photons become indistinguishable and interfere in such a way that they always exit together through the same output port of

the beam splitter, resulting in simultaneous detections at both output ports, known as coincidence events, are strongly reduced or completely suppressed. [8][11][21].

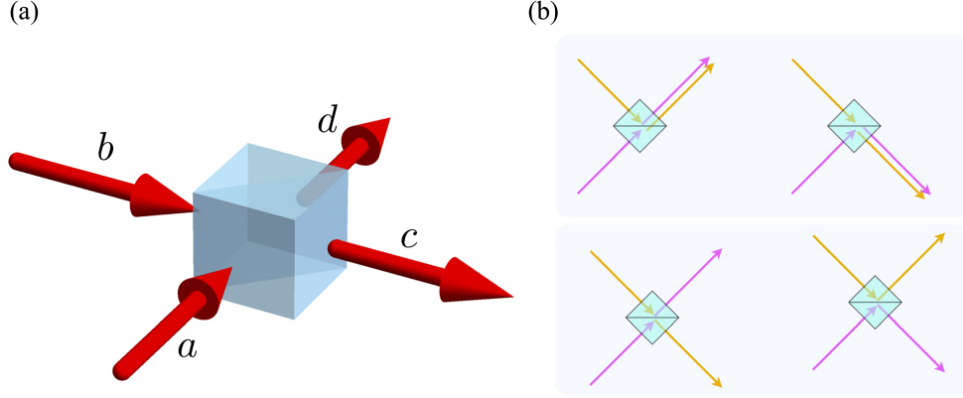


Figure 1.12: (a) Beam splitter with input ports a, b and output ports c, d. (b) Two photons can exit through the same port (top) or different ports (bottom) [7].

Recently, the Italian Space Agency (ASI) has begun actively investigating this phenomenon as part of its research in quantum technologies and optical systems for space applications. In their experimental configuration, a laser beam is split using a beam splitter, and the resulting interference effects are analyzed through coincidence measurements.

For detection, ASI employs two SPADs, and coincidences between photon arrivals on different detectors are measured by comparing their absolute timestamps. This capability is enabled by the time-tagging acquisition mode implemented in the custom multi-channel module object of this thesis.

It also allows researchers to reconstruct the HOM interference pattern and quantify the degree of indistinguishability between the photons generated in the laboratory. The ongoing work by ASI represents a significant step in integrating quantum photonic systems.

1.3. Architecture of the ASI TCSPC implementation

This section provides a comprehensive overview of the ASI TCSPC module. The goal is to outline the architecture and operational flow of the entire module, describing the roles of its key components and how they interact to perform accurate time-resolved photon measurements. By contextualizing the system architecture, this section lays the groundwork for understanding the rationale and requirements behind the software developed in this project.

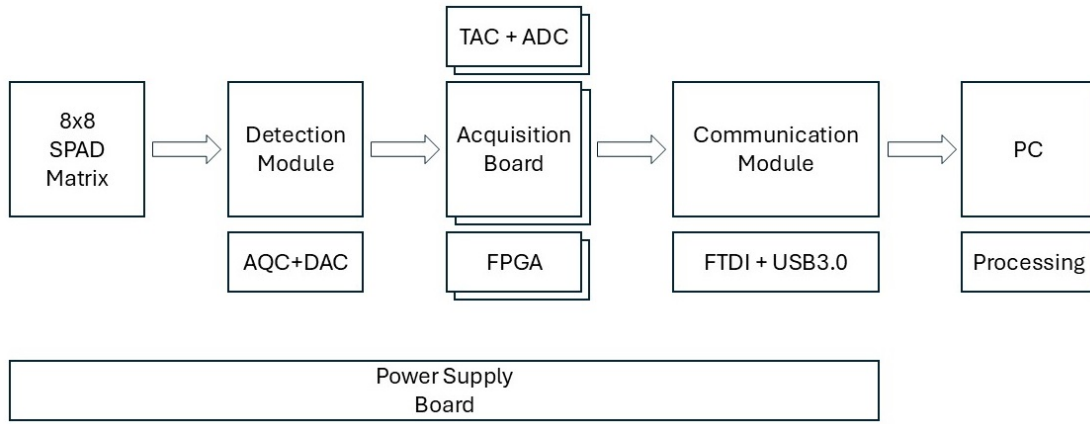


Figure 1.13: Schematic of ASI TCSPC architecture.

As just mentioned, this system includes a 8x8 SPAD matrix, which captures single photons and outputs corresponding electrical pulses. These signals are revealed by the detection board, where the Active Quenching Circuit (AQC) resets the activated SPADs ensuring the mitigation of the dead-time issue, and the Digital-to-Analog Converter (DAC) adjusts control signals and parameters [4][10]. The SPAD generated electrical signals then pass to the two acquisition modules (master-slave relationship), the core of the timing measurement system. Here, the TAC and ADC collaborate to digitize photon arrival times, while the microcontroller and FPGA manage data aggregation, synchronization, and real-time processing. The FTDI interface and USB 3.0 protocol handle high-speed transfer to the host PC for further analysis through MatLab code. A dedicated power supply system ensures stable voltage levels across all components, critical for maintaining timing accuracy and signal integrity. The communication board allows the two FPGAs to communicate through Aurora protocol and to transfer elaborated data to the PC by means of the USB 3.0. Then, the board features specialized circuitry for processing the critical STOP and GATE signals which serve distinct but equally vital functions in the TCSPC measurement process [33].

1.3.1. SPAD Matrix

A separate board was specifically developed to host the custom-designed 8×8 SPAD array, which is directly connected to the detection module. The photodetectors are arranged in an alternating orientation configuration with a precise 150 micrometers pitch, creating an optimized detection area measuring approximately $20 \text{ mm} \times 2 \text{ mm}$. This alternating layout enables efficient interfacing with the timing electronics, connecting to eight AQC Timing modules (16×1 channels each) symmetrically distributed, four positioned on the top side and four on the bottom side of the detection board. This design preserves the system's form factor while maximizing photon detection capability within the constrained spatial envelope. The carefully engineered 150 micrometers pitch represents an optimal balance between detector density and performance characteristics, ensuring reliable operation while minimizing crosstalk between adjacent SPAD elements [33][41].

1.3.2. Detection module

The detection module serves the critical function of converting luminescent signals into electrical pulses and transmitting them to the acquisition board. This core functionality is implemented through AQC that precisely manage the SPAD operation [33].

Each AQC detects the rising edge of the SPAD's avalanche pulse via a high-speed comparator and the output both triggers the rapid bias voltage reduction below breakdown level and generates a synchronous HSI counting pulse (High-Speed Interface single-ended signal). After a carefully controlled waiting period, the circuit restores the operational bias voltage. The research group's custom design enhances this basic functionality by incorporating a Transimpedance Amplifier (TIA) that produces differential LVDS (Low-Voltage Differential Signaling) timing signals for each detector, enabling precise time-stamping operations.

The single-ended HSI counting signals provide a pulse at the instant a photon is detected, while the differential LVDS signals carry precise timing information about the photon's arrival. The counting signals are routed directly to the FPGA inputs on the acquisition boards. To avoid a complex structure and inefficient resource usage, the timing signals are reduced in number through a routing logic before reaching the acquisition boards. This is achieved by combining multiple LVDS signals through OR-LVDS circuits. Finally, an EEPROM is used to initialize the power supplies.

Proceeding further along the detection chain, the Digital-to-Analog Converter (DAC) serves as a critical interface between the digital control systems and analog front-end

electronics. Specifically, the quad-channel DAC employed on the detection board translates digital commands from the microcontroller into precise analog control voltages that govern the behavior of the Active Quenching Circuits. This high-performance converter, operating with 12/16-bit resolution, generates the four essential analog control signals: CtQ (Control Quench to determine the quench duration), CtR (Control Reset to control the SPAD restore time above breaking voltage), CtS (Control Sense to regulate passive quenching resistance), and Vth (Threshold Voltage to adjust the TIA gain).

The combination of precision analog capabilities with robust digital interfacing makes this DAC implementation essential for maintaining optimal SPAD detector performance while enabling software-configurable control of the photon detection timing characteristics throughout the acquisition chain.

1.3.3. Acquisition module

The TCSPC module employs two acquisition boards, master and slave, which work in tandem to process photon detection events with high throughput and precision. Each board integrates the complete signal chain (time-to-amplitude conversion, analog-to-digital conversion, and real-time digital processing via FPGA) enabling parallelized, low-latency timestamping of single-photon events [12][33].

Each board is equipped with its own USB 3.0 interface via an FTDI chip, however, the system is configured to operate in a master-slave mode to optimize data handling and coordination between the two FPGAs. In this case, while both FPGAs process incoming signals from their respective ADCs, only the master board manages the final data aggregation and PC communication via USB 3.0. The slave board synchronizes with the master through a high-speed Aurora protocol link, implemented via the FPGA's GTX transceivers, ensuring seamless coordination between the two processing pipelines [12].

The FTDI interface, which bridges the FPGA and USB 3.0 controller, also facilitates high-bandwidth data transfer to the host PC, while an onboard microcontroller handles critical initialization tasks: handling SPAD resets through AQC and DACs, configuring ADCs, choosing the master board, setting up clock domains, and managing power sequencing.

This hierarchical design ensures robust operation, balancing distributed signal processing with centralized data management for optimal timing resolution and system scalability.

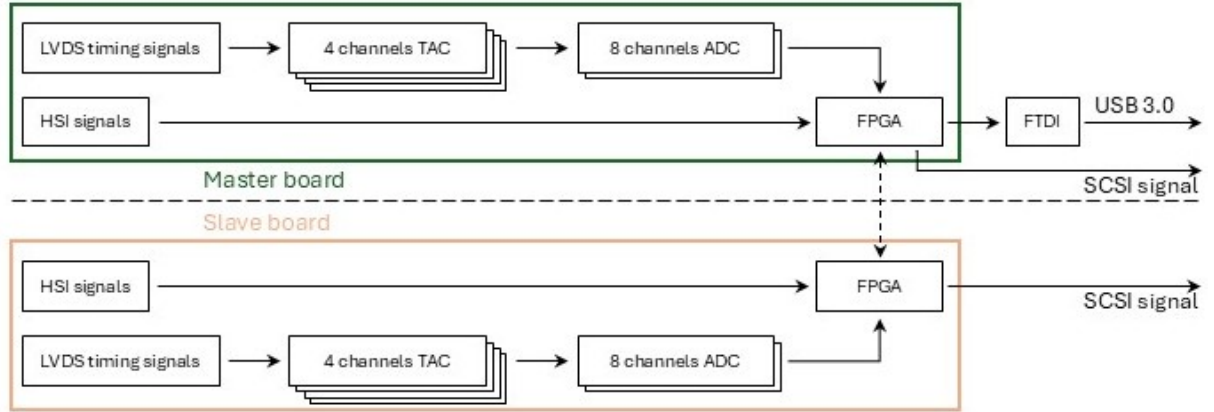


Figure 1.14: Acquisition boards Block Diagram [33].

1.3.3.1. TAC Integration and Analog-to-Digital Conversion

The analog outputs from the TAC stage feed into the AD9653 ADC converters from Analog Devices, which form the critical digitization stage. These 16-bit, 125 MSPS (Mega Samples Per Second) converters employ a dual-lane LVDS interface operating in free-running mode, achieving 1 Gbps/lane throughput through DDR transmission (Double Data Rate, modality in which data are sampled two times for each clock cycle, doubling transmission speed). The ADCs require careful signal integrity management, and have a key role in the whole system since they have a huge impact on the DNL (Differential Non-Linearity, the deviation of an ADC or a DAC from the ideal step-size), on the occupied space, on the resolution and on the dead time. In the worst-case timing scenario, the system experiences a dead time limited to one sampling period (8 ns at 125 MSPS) [33].

The digitized data streams enter the FPGA processing core through LVDS receivers, where several critical operations occur simultaneously. The FPGA synchronizes the DDR data streams, implements dead time minimization through dual-lane operation, and separates the timing data from counting data. The FTDI USB3.0 interface maintains the full 1 Gbps/lane data rate through optimized DMA (Direct Memory Access) controllers and buffer management schemes implemented in the FPGA fabric. This architecture demonstrates careful balance between parallel processing (16 channels), timing precision (jitter-controlled clocking), and power efficiency, while the modular design allows for future sampling rate enhancements through firmware updates.

1.3.3.2. Embedded Microcontroller for Peripheral Coordination

The PIC18F57K42 microcontroller serves as the central configuration and control unit for the acquisition system. This RISC-based device operates at a 16 MHz instruction clock and integrates 128 KB of Flash program memory, 8 KB of SRAM, and 1 KB of EEPROM, all powered by a 3.3V supply. It combines computational efficiency with robust peripheral support, including Universal Asynchronous Receiver Transmitter (UART), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), Direct Memory Access (DMA), Configurable Logic Cells (CLC), Signal Measurement Timers (SMT), a Numerically Controlled Oscillator (NCO), and a Compute-Enabled ADC (ADC2) [33].

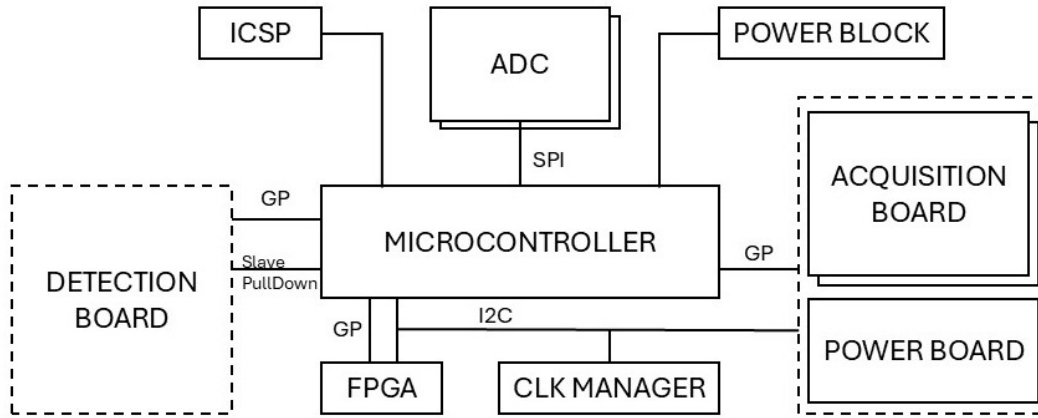


Figure 1.15: Microcontroller schematic [33].

The microcontroller is programmed and debugged through the standard ICSP (In-Circuit Serial Programming) method, which allows to operate without removing it from the board. It uses a small serial interface (few pins) to upload firmware via a programmer connected to an ICSP header.

During initialization, the microcontroller orchestrates critical hardware configurations:

- **Power Management:** activates DC-DC converters via enable pins and monitors their status through power-good signals, ensuring stable voltage delivery to all sub-systems.
- **ADC Configuration:** programs the two ADCs via SPI, setting sampling parameters and calibration registers to optimize time-to-digital conversion.
- **FPGA Coordination:** communicates with the FPGA to synchronize system states, enabling real-time monitoring and control of data acquisition.

The microcontroller also configures the clock manager to generate precise timing references for ADC sampling clocks (125 MHz), FPGA core logic synchronization and Internal FPGA transceivers (GTX interfaces).

The system employs a hardware-software hybrid approach to distinguish master and slave boards. It uses a dedicated `SlavePullDown` signal routed from the detection board to the microcontroller, so the board's role is determined by how this signal line is terminated:

- **Slave boards:** physically connect this line to ground through their connector, creating a low (0V) logic state.
- **Master boards:** leave it floating, with an external pull-up network biasing it to VDD.

The master-slave configuration uses the hardware signal `SlavePullDown` (grounded for slaves, pulled high for master) to automatically assign roles to the two acquisition boards during startup, enabling plug-and-play operation without manual configuration.

In the end, the microcontroller's ADC2 peripheral monitors analog health signals (e.g., supply voltages, temperature), while its SMT modules verify critical timing margins. The NCO provides programmable clock sources for auxiliary components, reducing reliance on external oscillators.

1.3.3.3. FPGA Logic for High-Speed Time Stamping

The data flow architecture of the system was designed to ensure a reliable and efficient transfer of information between the FPGA and the host PC, exploiting the capabilities of the FTDI FT601 USB 3.0 interface. Data produced by the FPGA is stored temporarily in a hierarchical FIFO structure, enabling parallelism and pipelining before being channeled into a final FIFO from which the FT601 performs burst reads.

The Xilinx Kintex-7 XC7K325TFBG900-1 FPGA forms the computational backbone of the acquisition system, integrating real-time signal processing, data routing, and communication protocols. Designed to handle high-throughput time-tagging and photon counting, this FPGA balances I/O bandwidth, memory resources, and power efficiency to meet the stringent requirements of single-photon detection [33].

At the core of its operation, the FPGA manages dynamic signal routing through programmable logic that activates multiplexing when more than 16 detectors are active per acquisition module. This flexibility ensures efficient resource utilization across the system's 64 input channels.

For timing measurements, the FPGA processes 16 differential channels from the TACs, synchronizing them with digitized data from the ADCs. The deserialization of LVDS data streams is carefully managed, with clock domain crossing logic aligning frame clocks (FCO) and data clocks (DCO) to maintain timing integrity in Double Data Rate mode. This means that the system takes high-speed digital data that arrives in pairs of signals (LVDS) and carefully converts it back into a usable format. Since the data is transmitted at double speed (DDR mode, using both rising and falling clock edges), special circuitry makes sure the timing stays perfectly synchronized. Calibration is handled through an integrated dithering system, where the FPGA generates precise sequences to control the TAC-internal DACs via dedicated reset (DAC_RST) and increment (DAC_INC) signals. This careful synchronization ensures no timing errors occur when processing the ultra-fast photon detection data, maintaining the system's precision down to billionths of a second.

The FPGA's interfacing capabilities are equally critical to system operation: high-speed USB 3.0 communication is managed through an FTDI interface using a 32-bit FIFO bus architecture, while inter-board synchronization is maintained via the Aurora protocol implemented on dedicated Gigabit Transceivers (GTX). During system initialization and calibration, the FPGA coordinates with the microcontroller through I2C, exchanging configuration parameters and operational status.

Memory management presents a key design challenge, as histogram generation for 64 channels at full 16-bit ADC resolution would require 67.11 Mb of Block RAM (BRAM) which significantly exceeds the FPGA's 16.02 Mb capacity. This constraint is resolved by operating the ADCs at 13-bit resolution and implementing efficient data encoding schemes, allowing histogram bins to be stored in either 2 or 3 bytes while maintaining full functionality. This optimization reduces memory requirements by a factor of eight without compromising measurement quality.

This implementation demonstrates how modern FPGA technology can be tailored to meet the exacting demands of time-correlated single-photon counting, providing the necessary computational density for real-time processing while maintaining the picosecond-scale timing accuracy required for advanced photonic research applications.

1.3.3.4. FTDI FT601 USB 3.0 Data Bridge

The timing data processed by the FPGA is then transmitted externally via a USB 3.0 SuperSpeed protocol, implemented using the FT601Q-B-T controller from FTDI. This device was selected for its high throughput (theoretical peak of 3.2 Gbps or 400 MB/s in USB 3.0 mode) and backward compatibility (480 Mbps in USB 2.0 mode), making it ideal

for modern TCSPC applications requiring fast and reliable data streaming [25][27][28].

So, the FT601 USB 3.0 controller plays a pivotal role in achieving low-latency transfer of time-tagging data from the FPGA to the host PC. Given the stringent requirements of TCSPC, the FT601Q's architecture is specifically designed to maximize USB 3.0 SuperSpeed bandwidth efficiency. Reaching maximum throughput demands a sophisticated software architecture designed for high throughput data processing, with particular emphasis on bandwidth optimization, the central objective of this research.

To achieve this goal, two main modes of data communication between the host computer and an FPGA have to be analyzed: the Single Channel FIFO mode (245 Synchronous) and the Multi-Channel FIFO mode [28]. The latter is the one specifically designed to fully exploit the USB 3.0 SuperSpeed interface, since the 245 mode is technically compatible with USB 3.0, it originates from earlier generations of FTDI devices and is commonly associated with USB 2.0 communication. As such, it lacks some of the architectural optimizations and flexibility provided by the Multi-Channel FIFO mode, such as multi-endpoint support and more efficient data transfer scheduling [28].

In this project, despite using only a single bidirectional channel, the Multi-Channel mode was adopted to ensure maximum control and compatibility with the high-speed acquisition system based on an FPGA. This choice enables better timing management, easier signal routing, and more robust firmware implementation, while still achieving the same theoretical throughput as the legacy 245 mode [28].

One major bottleneck stems from USB 3.0's packet-based protocol, where acknowledgment handshaking and bulk transfer framing reduce usable throughput. To mitigate this, it was implemented packet-size tuning in the FPGA firmware, balancing smaller packets (to minimize latency) with larger bursts (to improve throughput efficiency). By empirically testing different payload sizes, it was identified an optimal 2 KB burst configuration that maximizes FIFO utilization while keeping USB overhead manageable [33].

To overcome the challenges of reading large volumes of time-tagging data from the FT601Q USB 3.0 controller without loss, in this thesis is developed a high-performance C sharp application that combines low-latency data streaming and efficient buffer management through multi-threading. Given the constraints of USB 3.0 (protocol overhead, host-side bottlenecks, and non-deterministic latency), the software architecture was carefully optimized to ensure gap-free acquisition at sustained throughput exceeding 300 MB/s.

1.3.4. Power delivery and regulation

The power supply board serves as the central hub for voltage regulation and system protection, ensuring stable operation across all modules. It receives a 12V input from the rear connector and implements reverse polarity protection to safeguard the entire system. This board manages multiple critical power rails, including:

- **SPAD biasing voltages:** to maintain optimal detector operations.
- **AQC power supplies:** providing VDDH and VDDH, 3.3V for high-side logic, along with 3.3V and 1.8V for low-side logic.
- **General system voltages:** such as $\pm 5V$, 5.3V, 3.3V, and floating supplies for isolated circuits.

Additionally, the board integrates a microcontroller that reads user-defined voltage settings via UART/USB 2.0 and configures them through onboard DACs, storing these parameters in the detection board's EEPROM for persistence. A PID-controlled (Proportional, Integral and Derivative) Peltier regulator is also implemented to precisely thermostat the SPAD array, ensuring temperature stability for consistent performance [33].

This comprehensive power architecture guarantees reliable operation while supporting configurable voltage adjustments for different experimental conditions.

1.3.5. Communication module

The communication board operates as the system's nerve center, seamlessly integrating three critical functions: high-speed data transfer between acquisition FPGAs, efficient communication with the host computer, and precise management of timing synchronization signals [33].

As detailed in the following chapters, each photon event is encoded into a 32-bit word, structured as: 7-bit SPAD address, 16 bit micro-time and 9 bit macro-time information.

The required throughput T for the classic TCSPC mode is:

$$T = N_c \cdot N_b \cdot f = 16 \cdot 32 \cdot 5 \times 10^6 = 2.56 \text{Gbit/s}$$

Naming N_c the number of channels, N_b the number of bits for each data and f the frequency limit for each TAC to work. So, each channel throughput is limited to 5 MHz due to the TAC constraint.

The communication board interfaces externally through a USB 3.0 connector and its signals originate from the acquisition boards and pass through the communication board, enabling high-speed data transmission to the host computer. The integration of this interface ensures compliance with USB 3.0 standards, while maintaining signal integrity at high transfer rates.

1.4. Considered Approaches in TCSPC Implementation

This section explores the two major strategies for implementing the Time-Correlated Single Photon Counting technique: histogram construction and time-tagging. These methods define how photon arrival events are recorded and encoded, processed, and later analyzed. Each approach presents distinct architectural trade-offs in terms of temporal resolution, data throughput, memory usage, and post-processing flexibility [4] [38].

1.4.1. Histogram Construction

The Time-Correlated Single Photon Counting histogram generation represents a sophisticated data processing challenge that demands careful algorithmic design and hardware optimization. At its core, the system implements a distributed computing architecture where the FPGA serves as a massively parallel temporal accumulator, transforming asynchronous photon arrival events into precisely timestamped histogram bins through a multi-stage computational pipeline [33].

The core measurement chain begins with the TAC, which acts as a precision analog computer that linearly converts the time interval ($\Delta t = t_{STOP} - t_{START}$) into a proportional voltage output. This analog voltage is then digitized by a high-resolution ADC creating a digital word that directly corresponds to the photon arrival time relative to the excitation pulse. The ADC's output serves as a memory address pointer, where each unique digital value maps to a specific time bin in the histogram.

The histogram memory itself is implemented as a dedicated buffer in the FPGA's Block RAM (BRAM), organized as an array of 32-bit counters. Each memory location represents a discrete time channel, with the address space covering the full temporal range of interest (typically several times the fluorescence lifetime being measured). When a photon event occurs, the system executes an atomic read-increment-write operation at the memory location specified by the ADC's output. This operation must be completed within a single clock cycle to maintain throughput at high photon counting rates.

Precisely, in the histogram mode, the data sent to the PC is so built:

$$DataSize = N_{bins} \cdot N_{bit} \cdot N_{ch} = 16384 \cdot 32 \cdot 16 = 1,048MB$$

Naming N_{bins} the number of bins which constitutes the histogram, N_{bit} the number of bits for each data and N_{ch} the number of channels (one histogram per channel).

Practically, the time window defined in the section 1.1, is discretized in 16384 bins and whenever a photon is detected inside it, the ADC increments the corresponding bin of the considered histogram. After many cycles, the distribution of the photons is created as shown in Figure 1.16.

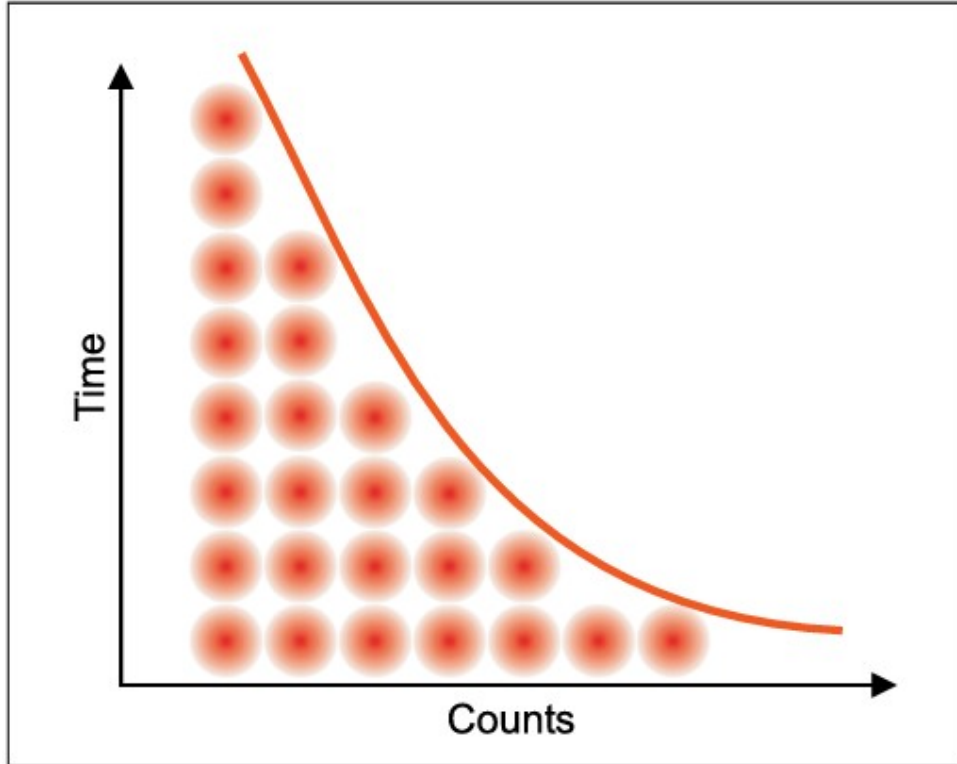


Figure 1.16: Histogram of start-stop times in time-resolved fluorescence measurement with TCSPC [37].

In conclusion, for the histogram mode, the FPGA pre-processes photon arrivals into binned counts and handles all time-critical tasks. So, the software only has to read "finished" histograms at leisure and it uses less than 1% of USB3.0 bandwidth. FPGA also aggregates data, c sharp only reads via polling or interrupts, so it has no timing requirements. Also the consequences of slow reading in histo mode are not so important: it merely results in a loss of statistical precision.

1.4.2. Time-Tagging and Event Logging Mechanism

The time-tagging modality in TCSPC systems represents a paradigm shift from traditional histogram-based approaches, implementing what is essentially a streaming data

architecture for single-photon event processing [4]. In this mode, the system behaves as a high-throughput temporal data generator, where each detected photon triggers the creation of a structured data packet containing complete timing information [38]. This architecture mirrors modern distributed streaming systems, with the FPGA acting as a specialized data producer and the host computer serving as the consumer in a persistent, time-critical data pipeline.

The time-tagging data structure utilizes a tightly packed 32-bit word to encode all critical information about each detected photon event. This format was carefully designed to balance temporal resolution, channel addressing capacity, and system throughput requirements while maintaining compatibility with modern computing architectures.

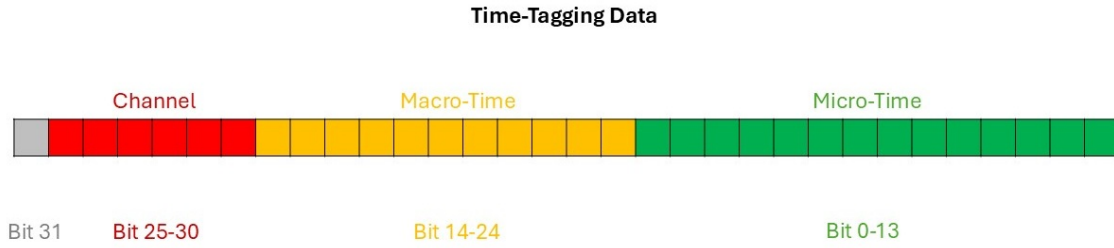


Figure 1.17: Time-Tag Data illustration [33].

- **Micro-Time (14 bits):** Occupying bits [13:0], this field contains the high-resolution temporal measurement of the photon arrival time relative to the laser excitation pulse. The 14-bit precision, sourced directly from the ADC output, provides 16384 discrete time bins. This fine temporal resolution is essential for accurate fluorescence lifetime determination [33].
- **Macro-Time (11 bits):** Utilizing the least significant bits [24:14], this field serves as a coarse timestamp tracking experiment elapsed time. The 9-bit width creates a wrapping counter that increments at a fixed clock rate (typically 10-100 MHz), providing the photon arrival times measured from the beginning of the experiment [33].
- **SPAD Address (7bits):** Positioned in the most significant bits [30:25], this field uniquely identifies which of the 64 SPAD detectors registered the photon event. The 7-bit width allows addressing up to 128 discrete channels ($2^7 = 128$), which matches the system's maximum channel count. This field enables post-processing correlation analysis between different detectors [33].

As just analyzed in the Communication Module section, the throughput requirements for the time-tagging format are substantial [38]. In a 16-channel system operating at each TAC's maximum rate of 5 MHz, the total data rate reaches 2.56 Gbps, in contrast to histogram mode where each photon is associated with only a small amount of data.

Note that, from time-tagging data, the Matlab program can obviously extract and reconstruct the same histogram reproduced with the histogram mode.

1.5. Data Throughput and Real-Time Processing Challenges

The need to process and store large volumes of timestamped data instantly is a significant challenge and it's the main focus of this thesis. As was presented in the preceding section, photon detection systems can generate data at rates of gigabits per second, and efficiently handling this high throughput is critical[38].

The histogram acquisition mode in TCSPC systems employs a decoupled processing architecture that fundamentally separates time-critical operations from host-side data handling. At its core, the FPGA implements a hard-real-time processing pipeline that performs photon arrival time quantization and histogram bin updates with deterministic latency, typically under 5 nanoseconds per event [33][40]. This dedicated hardware acceleration transforms raw photon events into statistically stable histogram representations while completely isolating the temporal precision requirements from the host computer.

The host software interacts with this system through an asynchronous, non-blocking interface designed for minimal bandwidth utilization, typically consuming less than 1% of the available USB 3.0 throughput [33]. This efficiency stems from the FPGA's role as a statistical aggregator, which outputs complete histogram snapshots at configurable intervals rather than streaming individual photon events. The C# application can retrieve these pre-processed histograms through either polling or interrupt-driven mechanisms, with neither approach imposing strict timing requirements on the host side.

Summarizing, the main challenges are:

- **High Data Rates:** as already introduced, photon detection systems generate vast amounts of timestamped data at rates reaching gigabits per second. Efficiently managing this high throughput requires a combination of optimized hardware and software strategies. Traditional storage and processing architectures may struggle to keep up with these demands, leading to potential data loss or increased latency.
- **Scalability and Flexibility:** as experimental setups grow in complexity, there is a need for systems that can scale both in terms of data acquisition capacity and flexibility to accommodate a wide range of measurement scenarios. Modular systems and software frameworks that allow for easy integration of data processing units can enhance the scalability of time-resolved photon counting setups.
- **Integration of the two modalities:** unifying the two data acquisition modalities in the same software isn't banal to implement. It requires the fully comprehension

of object-oriented paradigm to exploit the Single Responsibility Principle, which allows to avoid interferences in data interpretation.

2 | Software of the ASI module

The development of the data acquisition software presented in this thesis, carried out within the framework of a scientific collaboration involving the Italian Space Agency, required a programming platform capable of handling high-speed data streams, offering modular architecture, and enabling a seamless integration between user interface and hardware-level communication. To fulfill these requirements, the C# programming language and the Visual Studio Integrated Development Environment (IDE) were chosen. C# is a modern, object-oriented language developed by Microsoft, widely adopted in both industrial and scientific contexts for its expressiveness, safety, and ease of use. Its fully object-oriented paradigm allows for the implementation of clean, maintainable, and scalable software architectures, a crucial requirement for research environments where evolving hardware and experimental protocols demand frequent software adaptations and extensions.

A key factor in the language selection was the necessity of interfacing with a FTDI FT601 USB 3.0 Superspeed bridge, responsible for transferring 32-bit encoded histogram and time-tagging data from the FPGA to the host computer. C# offers native interoperability with .NET-based libraries, including the FTD3XX.NET library provided by FTDI, which supports low-level communication with the FT601 chip. Indeed, this allows direct access to critical USB functions such as ReadPipe and WritePipe, while ensuring high throughput and stability [26].

Furthermore, Visual Studio provides a comprehensive development ecosystem that integrates code editing, graphical interface design, debugging, and version control into a single environment. Its Windows Forms and WPF (Windows Presentation Foundation) frameworks enable rapid and flexible creation of user interfaces, allowing the software to expose configuration panels, real-time data visualizations, and acquisition controls within a consistent and user-friendly front-end.

The combined use of C# and Visual Studio enables the unification of hardware-level communication, data processing logic, and user interface design within the same project structure, greatly simplifying the development workflow and increasing maintainability.

The resulting software is not only performant and reliable, but also future-proof, being easily extensible to new acquisition modes, hardware channels, or visualization tools.

In conclusion, the C# .NET platform and Visual Studio 2022 IDE were selected for their unmatched support for object-oriented design, compatibility with the FT601 interface via FTD3XX, and their integrated tools for building robust scientific applications with both front-end and back-end components.

2.1. Object-Oriented Programming Paradigm: Principles and Methodologies

Object-Oriented Programming (OOP) has emerged as a predominant paradigm in software engineering, aiming to increase code modularity, reusability, and maintainability. Unlike procedural programming, which organizes code as a sequence of instructions and functions, OOP structures software into entities known as objects, encapsulating both data and behavior. This chapter introduces the fundamental concepts of OOP, its core principles, and discusses its relevance to the software system developed within this thesis [17].

2.1.1. Conceptual foundations of OOP

The underlying philosophy of OOP is based on the abstraction of software components: what matters is not how a functionality is implemented internally, but rather what characteristics and actions a software component is capable of performing and exposing to the outside world. This **black-box** approach mirrors principles in hardware design, where a user interacts with a device based on its visible functionalities, not its internal electronic circuitry [17][29].

At the core of OOP lies the concept of the **class**, which serves as a formal specification of a data type, defining both its internal structure and the operations that can be performed on it. A class acts as a blueprint from which **objects**, concrete instances of that class, are created; each object encapsulates a set of **attributes** (also referred to as fields or properties) that represent its internal state, and a set of **methods** (also called member functions) that define its behavior. Through these mechanisms, objects communicate and collaborate, enabling complex system behaviors to emerge from simpler components.

According to Bertrand Meyer [29], one of the fundamental advantages of the object-oriented paradigm is its capacity to manage software complexity by allowing developers

to structure systems into modular, reusable, and self-contained components. By encapsulating both data and behavior, classes promote a high level of cohesion and information hiding, which not only reduces the cognitive load on developers but also enhances code maintainability and scalability.

In practice, the consistent application of object-oriented principles, through the careful definition of classes, the instantiation of meaningful objects, and the implementation of well-scoped methods and attributes, leads to robust and adaptable software architectures that can evolve alongside the requirements of scientific and industrial applications.

2.1.2. Properties of OOP

The object-oriented paradigm provides several key advantages in software engineering. These benefits derive from core principles such as encapsulation, inheritance, and polymorphism, which together promote reusability, scalability, and robustness of code.

In the context of scientific software, such as photon detection systems interfacing with FPGA-based hardware, these features are critical for managing architectural complexity.

2.1.2.1. Encapsulation

Encapsulation is the practice of organizing data and the operations on that data within a single module, typically a class, while exposing only a well-defined external interface. This limits dependencies between components and enables internal changes without affecting external code, as long as the interface remains stable. Information hiding is a key aspect of encapsulation: it ensures that internal implementation details are not accessible to clients, promoting modularity, maintainability, and safe evolution of the software. To maximize the advantages of encapsulation, the exposure of implementation details has to be reduced to the bare minimum [35].

In the presented software, for example, encapsulation is evident the classes `DataReader` `TimeTag` and `DataReaderHisto`, which are responsible for data reading and expose only necessary public methods like `Run()` and `Stop()` while hiding the complexities of buffer management, file operations, and USB communication. This clear separation of concerns preserves data integrity, prevents accidental misuse, and makes the system easier to maintain, debug, and extend.

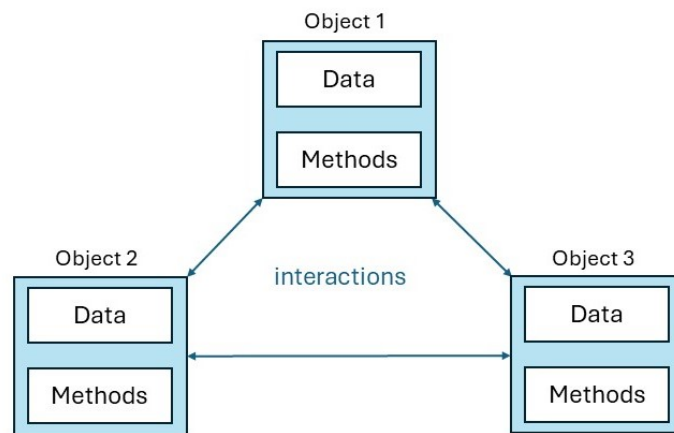


Figure 2.1: Objects interacting through encapsulation.

2.1.2.2. Inheritance

Inheritance is a feature that enables a class, known as a subclass or derived class, to acquire the attributes and methods of another class, referred to as the superclass or base class. This promotes code reuse and supports the creation of hierarchical class structures. By defining general functionality in the base class, developers can extend and specialize it in derived classes without repeating code, since subclasses inherit the properties and methods of the superclass [29][31][35].

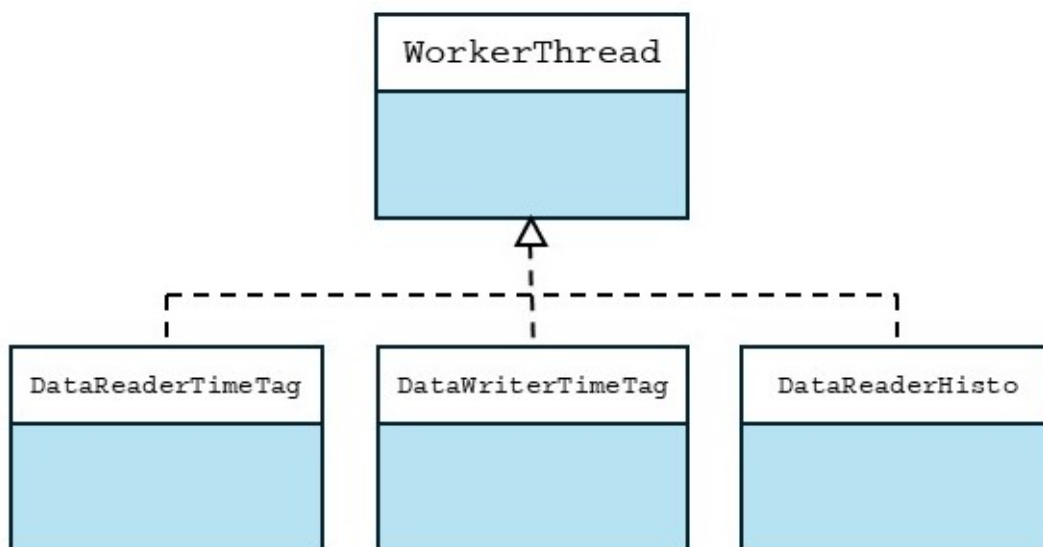


Figure 2.2: WorkerThread is the superclass, the other are the subclasses.

In the context of the presented software, a general abstract class `WorkerThread` defines a common interface and shared behavior for threaded operations, including methods such as `Run()`, `Stop()`, and the abstract method `AsyncTask()`. Specific modules such as `DataReaderTimeTag` and `DataReaderHisto` extend this base class, implementing custom acquisition logic tailored to each modality (time-tagging and histogram). This inheritance-based approach ensures consistency in thread management while allowing for modular and scalable specialization of the data acquisition process. All the three subclasses shares the `AsyncTask` method but it performs different tasks based on who called it.

2.1.2.3. Polymorphism

Building on inheritance, polymorphism enables objects of different classes to be treated uniformly if they share a common superclass or interface. This behavior manifests most commonly in the ability to call the same method on different objects and have each one respond in its own way, a principle known as **dynamic dispatch** [29].

In object-oriented programming, understanding the difference between compile-time and runtime is essential to grasp polymorphism. Compile-time refers to the moment when code is translated into executable form, while runtime is when the program is actually executed. This distinction is key, as polymorphism can occur at either stage, with static polymorphism resolved during compilation and dynamic polymorphism handled during execution [29].

A concrete implementation of polymorphism is evident in the provided code through the use of the `WorkerThread` abstract class. This class defines an abstract method `AsyncTask()`, which serves as a polymorphic interface to asynchronous operations performed by derived thread classes. As just underlined in the preceding section, both `DataReaderTimeTag` and `DataReaderHisto` inherit from `WorkerThread` and provide their own distinct implementations of `AsyncTask()` via the `override` keyword:

- `DataReaderTimeTag.AsyncTask()` reads time-tagging data from an FTDI device and places them in an intermediate buffer.
- `DataReaderHisto.AsyncTask()` continuously reads histogram data asynchronously and dispatches it to a graphical callback for real-time visualization.

The polymorphic behavior is triggered when the `Run()` method of `WorkerThread` is invoked: a new thread which executes `AsyncTask()` is internally created and started, without knowing which specific subclass it belongs to.

```
public void Run()
{
    if (isRunning)
    {
        return;
    }
    isRunning = true;

    thread = new Thread(AsyncTask);

    thread.Start();
}
```

This is the essence of **runtime polymorphism**: at runtime, the actual version of `AsyncTask()` executed depends on the concrete subclass.

Through this design, new thread behaviors can be introduced simply by extending `WorkerThread` and overriding `AsyncTask()`, without altering the existing threading infrastructure.

2.1.2.4. Abstraction

Abstraction is one of the core principles of object-oriented programming. It involves exposing only the essential features of an object through a simplified interface, while hiding the underlying complexity. This allows developers to interact with objects at a higher level, focusing on *what* they do rather than *how* they do it [3][29]. An abstract class is a base class that cannot be instantiated directly and is intended to provide a common definition for its derived classes. It may include both fully implemented members and abstract members, methods or properties declared without implementation, which must be explicitly overridden by any concrete subclass. This approach enforces a consistent interface across related classes while delegating the responsibility of specific behavior to the subclasses [3].

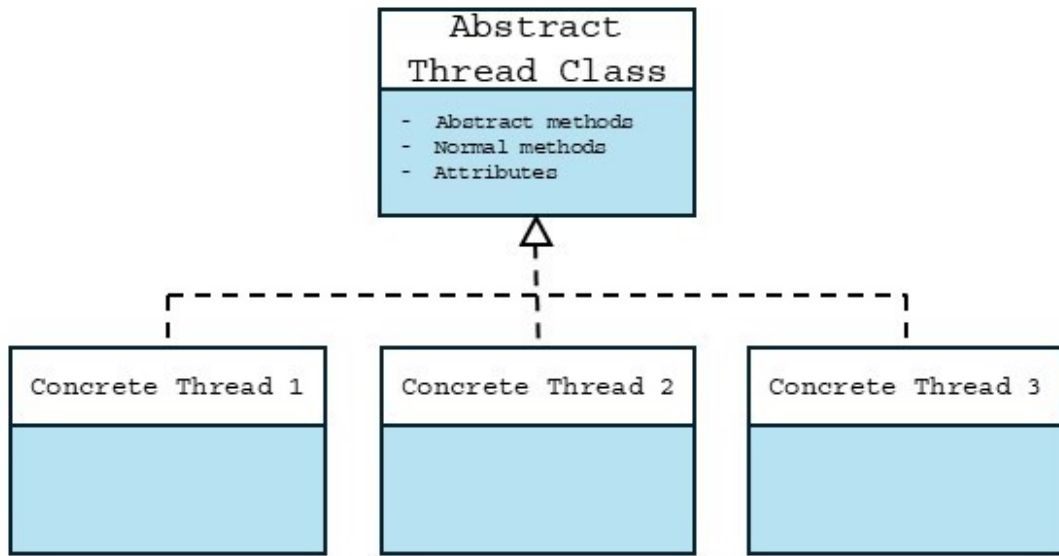


Figure 2.3: Abstraction example: Abstract Thread Class acts as a blueprint.

In this project, the concept of abstraction is concretely realized through the implementation of the `WorkerThread` abstract class, which defines the common structure and behavior of any background thread, encapsulating thread management, lifecycle control (start, stop, and status), and standardized logging mechanisms.

This abstraction allows for the creation of specific thread classes, such as `DataReaderTimeTag` and `DataReaderHisto`, that inherit from `WorkerThread` and focus exclusively on their domain logic: reading and writing time-tagged data, or continuously updating histogram data.

By abstracting the threading mechanism, the code achieves modularity, ease of maintenance and reliability. Thus, abstraction here is not just a theoretical principle, it plays a practical role in maintaining a scalable and robust multithreaded system.

2.1.3. Advantages of OOP

As previously stated, Object-Oriented Programming introduces a paradigm that organizes software design around data, or objects, rather than functions and logic. This approach offers several advantages that directly impact the development lifecycle of complex systems, particularly in terms of modularity, scalability, and maintainability.

2.1.3.1. Modularity

One of the primary advantages of OOP is its intrinsic support for modularity, indeed, in OOP, software is decomposed into discrete classes and objects, each encapsulating a specific functionality. This separation of concerns allows software engineers to isolate and independently develop, test, and debug different parts of the application. Each class operates as a self-contained module with well-defined interfaces, promoting code reuse and enhancing the clarity of the overall system architecture.

This modular structure is especially beneficial in large-scale projects, where teams can work concurrently on different components without generating code conflicts. Furthermore, the encapsulation principle ensures that internal implementations remain hidden between units, reducing interdependencies and fostering robustness in software systems [15].

Accordingly to Meyer [29], to be considered modular, a design method has to satisfy five fundamental requirements:

- **Decomposability:** a software construction method satisfies Modular Decomposability if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them.
- **Composability:** a method satisfies Modular Composability if it favors the production of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed.
- **Understandability:** a method favors Modular Understandability if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others.
- **Continuity:** a method satisfies Modular Continuity if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules.
- **Protection:** a method satisfies Modular Protection if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules.

2.1.3.2. Scalability

OOP facilitates both horizontal and vertical scalability of software systems: horizontal scalability is achieved through the reuse and extension of classes via inheritance and polymorphism, enabling the incremental addition of new features with minimal code duplication; vertical scalability, on the other hand, is supported by the hierarchical structuring of class relationships, allowing for the abstraction of complex behaviors into higher-level constructs.

By leveraging principles such as abstraction and generalization, OOP enables the design of frameworks and libraries that can be extended or customized according to specific application requirements. This flexibility is a cornerstone in the development of modern enterprise systems, where requirements evolve rapidly and extensibility is a key factor for long-term sustainability [29].

2.1.3.3. Maintainability

Maintainability is significantly improved in OOP due to the combination of modularity, encapsulation, and clear class hierarchies. When changes or bug fixes are required, developers can locate and update the relevant class or method without affecting the rest of the system, since the modules are independent from each other.

Combined with the good habit of commenting the code, this localized modification reduces the risk of regression and enhances the reliability of software evolution, making life easier for colleagues and future programmers.

Additionally, OOP promotes the use of design patterns and standardized architectures, such as *producer-consumer* one, which further improve the readability and organization of codebases [13]. These structured approaches are essential in maintaining complex systems over time, particularly in collaborative development environments where code quality and documentation play a pivotal role.

2.2. Concurrent Programming and Thread-Based Architectures

Modern software systems frequently demand high-performance, especially in data-intensive applications such as instrumentation, signal processing, and real-time monitoring. Concurrent programming, and more specifically, thread-based architectures, provide the foundation to meet these performance requirements by enabling parallelism and efficient CPU utilization.

2.2.1. Fundamentals of concurrency in software systems

Concurrency refers to the ability of a system to execute multiple tasks logically simultaneously and, in the context of software engineering, it involves structuring programs in a way that multiple sequences of operations, can be executed independently and in parallel. This paradigm is crucial when applications must remain responsive (e.g., GUI applications), or when multiple I/O or computational tasks must be performed without bottlenecks [1][2][16].

Threads are the basic unit of concurrency, each one has its own execution context but shares the memory space with other threads within the same process, enabling fast communication at the cost of increased complexity in resource sharing and synchronization.

2.2.2. Comparative analysis: sequential vs. parallel execution

In sequential execution, tasks are processed one after the other, which can lead to inefficiencies when dealing with tasks that involve waiting. In contrast, parallel execution enables multiple tasks to be performed simultaneously, either on multi-core CPUs or across distributed systems, thereby improving throughput and latency [16].

- **Sequential Execution:** Simpler to implement and debug; however, it doesn't exploit modern hardware at their maximum potential and may introduce latency in I/O-bound processes.
- **Parallel Execution:** Requires a more complex program design to handle synchronization, locks, race conditions, and shared resource contention, but can greatly increase performance and responsiveness [36].

The choice between sequential and concurrent models depends on application requirements, system architecture, and desired scalability. Figure 2.4 illustrates a hypothetical scenario where threading can be beneficial for dividing a workload. However, if a thread-safe structure is not used, threads may interfere with one another, potentially leading to unpredictable results such as "CEBAFD" or "FBEADC", and even data being overwritten or corrupted.

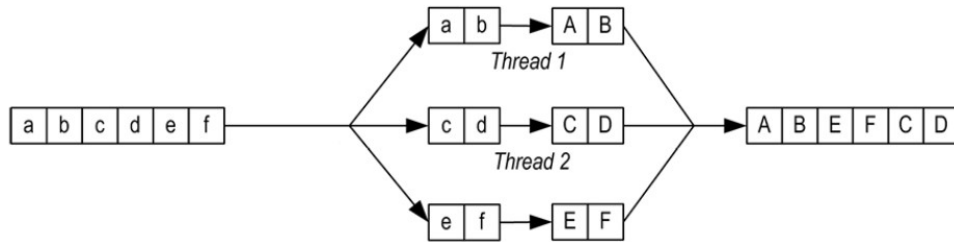


Figure 2.4: Parallelization principle [1].

2.2.3. Thread management and synchronization in C#

C# provides a comprehensive framework for thread-based programming via the namespace `System.Threading` and the higher-level `Task Parallel Library (TPL)`. Threads can be instantiated explicitly using the `Thread` class or managed more abstractly using `Task`, `async/await`, and the `Parallel` class [1][2][31].

As noted in the preceding section, proper synchronization is critical to avoid race conditions, deadlocks, and inconsistent states. Common synchronization primitives in C# include:

- `lock` keyword
- `Mutex` and `Semaphore`
- `ConcurrentQueue<T>`, `BlockingCollection<T>` for thread-safe data structures

The current version of the project adopts a *producer-consumer* model built around the `BufferManager` class, which leverages a thread-safe `BlockingCollection<byte[]>` to manage the flow of data from the FTDI device to disk. This approach streamlines synchronization by eliminating the need for explicit locks or complex concurrency mechanisms, resulting in a more robust, modular, and maintainable system.

2.2.4. Practical implementation: the `DataReaderTimeTag` thread

In the current architecture, the `DataReaderTimeTag` class is responsible for acquiring raw data blocks from the FTDI interface and passing them to a shared buffer as shown in Figure 2.5. This is part of a *producer-consumer* architecture where data acquisition and file writing are performed by two separate threads. The reader thread, implemented in `DataReaderTimeTag`, performs a single `ReadPipe()` call to retrieve the full data block and then inserts the buffer into a thread-safe queue, implemented via a `BlockingCollection<byte[]>` in the `BufferManager` class.

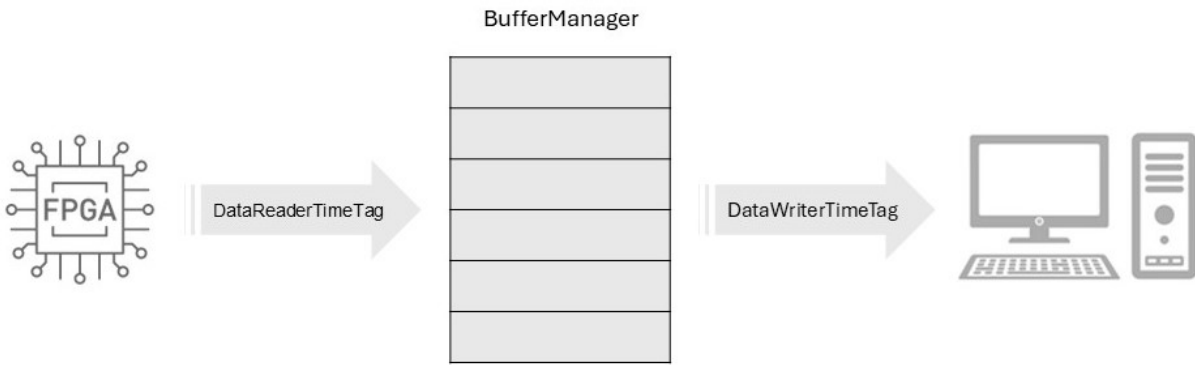


Figure 2.5: Implementation of the Producer-Consumer pattern.

This design replaces an earlier version in which both reading and writing operations were performed within a single thread. While that approach was simpler in terms of control flow and synchronization, it tightly coupled data acquisition with disk I/O, leading to reduced responsiveness and longer latency between acquisitions.

By decoupling the two phases and adopting the *producer-consumer* pattern, the system now offers several practical benefits: the reader thread can return immediately after inserting the data into the buffer, making it available for subsequent acquisitions without waiting for disk operations to complete; it enhances system scalability, making it easier to handle higher and higher data rates; and increases robustness, as temporary slowdowns in disk I/O do not block data collection. Additionally, it facilitates better resource utilization, since CPU and I/O operations are no longer tightly synchronized.. Concurrently, a dedicated consumer thread (`DataWriterTimeTag`) retrieves the buffered data and writes it to disk, splitting it into multiple binary files based on the number of files and their selected size.

A more in-depth technical analysis of the `DataReaderTimeTag` and `DataWriterTimeTag` thread, along with a comparative discussion of the histogram-oriented counterpart `DataReaderHisto`, will be presented in section 2.5.1.4.

2.3. Design and Engineering of the New Acquisition Software

This section presents the design strategy and engineering choices adopted to meet the project's functional and performance requirements with emphasis on a modular, maintainable, and scalable architecture, capable of supporting current needs while remaining adaptable to future enhancements. The following subsections describe the specification of software requirements, the rationale behind key architectural decisions, and the adoption of technologies enabling efficient data acquisition and processing.

2.3.1. Software Requirements and Performance Constraints

The development of the new acquisition software was guided by a set of high-level requirements focused on maximizing throughput, ensuring stable communication with the FPGA, and maintaining flexibility and usability. These requirements are summarized below:

- **High Throughput via USB 3.0:** the system must leverage the full practical bandwidth of the FTDI FT601 interface. Although USB 3.0 provides a theoretical link speed of 5 Gbps, practical throughput is limited by system components and software overhead and the FT601 chip, with a 32-bit FIFO interface operating at 100 MHz, supports a theoretical maximum of 400 MB/s. In this project, a stable and sustained data acquisition rate of 320 MB/s (2,56 Gbps) was achieved, representing a realistic upper bound for this interface.
- **Read-Centric Architecture:** since data is analyzed offline (using an ad hoc MATLAB application), the software architecture prioritizes read throughput from the FPGA over write operations. This ensures that data can be captured at maximum rate without bottlenecks and that subsequent reads does not need to wait for the write to end.
- **Uninterrupted Data Reading:** the acquisition software must guarantee a continuous and lossless data stream between the PC and the FPGA. This implies efficient memory buffering, non-blocking I/O operations, and robust error handling to prevent data overflow or communication stalls.

- **User Interface Usability:** the graphical interface must expose all acquisition controls in a clear, accessible, and unambiguous manner. Commands for interacting with each system component should be logically grouped and ergonomically designed to avoid ASI operators errors.
- **Modular and Maintainable Design:** the software must be organized in well-defined, decoupled components to facilitate future extensions and simplify debugging and maintenance.
- **Scalability and Parallelism:** the architecture must accommodate future expansion toward multi-threaded or multi-stream data acquisition, so, thread management must be designed to scale with hardware improvements and increased data flow complexity.

2.3.2. Architectural Decisions

The design and implementation of the new acquisition software were guided by a set of fundamental architectural decisions aimed at meeting the just mentioned requirements. This section presents the rationale and principles behind these decisions, providing a conceptual foundation for the class-level analysis that follows.

2.3.2.1. Adoption of Object-Oriented Design Principles

At the core of the design lies a modular structure in which each class encapsulates a well-defined responsibility. This ensures high cohesion within components and low coupling between them, facilitating future extensions and code evolution. Critical functionalities, such as data acquisition, file writing, communication and connection control, and user interface management, were logically separated into dedicated classes, thereby simplifying both the reasoning about the codebase and the debugging process.

Abstraction was another central principle: hardware-specific operations (e.g., direct interaction with the FT601 chip or the underlying USB 3.0 transport layer) were encapsulated in clearly delimited modules. This abstraction layer decouples low-level mechanisms from high-level application logic, allowing for better testability and enabling possible future hardware upgrades with minimal impact on the rest of the system.

A complete analysis of each class is deferred to a later section.

2.3.2.2. FTDI FT601 Interface and Data Transmission Libraries

The integration of the FTDI FT601 SuperSpeed USB 3.0 to FIFO bridge chip was a pivotal aspect of the acquisition software's architecture to facilitate seamless communication between the PC application and the FT601 hardware. The FTDI D3XX driver and its associated .NET wrapper were utilized to provide a high-performance API designed specifically for the FT60x series, enabling efficient data transfer and device control.

All FTDI FT601 communication functionalities were modularly encapsulated in the `FTDIController` class.

The FTDI D3XX .NET library was used to directly control the USB3.0 data stream, the key methods employed are:

- **CreateDeviceInfoList** and **GetDeviceInfoList**: these two methods are used together to detect connected FTDI devices. The first builds a device information list and returns the number of D3XX devices connected to the system, while the second retrieves a list of detailed information about each plugged-in D3XX device, such as its serial number, description, and index. This separation allows for dynamic enumeration of USB devices, which is essential in environments with multiple FTDI interfaces.
- **OpenByIndex**: opens a connection to a specific FTDI device based on its index in the enumerated list. This method is often preferred when the device order is known or user-driven (this case, since the user chooses), and it avoids reliance on string-based identifiers like serial numbers or descriptions.
- **ReadPipe**: provides direct access to the USB endpoint for high-throughput read operations. It is particularly well-suited to streaming data from devices like the FT601, as it enables low-latency, large-block data transfers directly from the specified endpoint (0x82).
- **SetPipeTimeout**: allows the programmer to configure a timeout for read and write operations on a given pipe. The default value is set to 5000 ms and this means that if no byte arrives within this time gap, the ReadPipe returns the `FT_IO_TIMEOUT` error. Within the scope of this project, setting the timeout to 0 disables it entirely, which is appropriate for continuous acquisition scenarios where operations must block until data is available.

- **WritePipe:** used to send data or commands to the USB device via a specific output endpoint (0x02); in applications involving FPGA control, this function enables the transmission of control words or parameters in real-time.
- **AbortPipe:** this function is essential for implementing user-interruptible data acquisition loops and ensuring responsive UI behavior during long measurement sessions. Indeed, it offers a mechanism to interrupt ongoing pipe operations, which in this project is employed to terminate a blocking ReadPipe call.

These methods, provided by the FTDI D3XX .NET library, constitute the foundation for direct and efficient communication with the FT601 chip, supporting both control and high-throughput data transfer operations in a flexible and modular way.

2.3.2.3. Scalable threading model for future parallelism

To support potential future extensions such as multi-channel acquisition or real-time data processing, a scalable threading model was incorporated into the software design. Rather than hard-coding a fixed number of worker threads, the architecture allows for dynamic allocation based on runtime needs. The implementation of this model, long with the responsibilities and behavior of the individual threads, is detailed in the section 2.5.1.4, where the `WorkerThread` class is discussed.

2.3.3. Class architecture and system overview

To provide a comprehensive overview of the software structure, this section presents a UML class diagram representing the main components of the application and their relationships. The diagram highlights the core classes involved in device communication, data acquisition, buffering, and user interaction.

Although each class will be described in detail in the following sections, the diagram in Figure 2.6 and 2.7 serves as a high-level map of the system, illustrating dependencies, data flow, and interaction points. This includes the user interface forms, the controller responsible for USB communication, background threads for data processing, and configuration classes used to parameterize acquisition behavior.

The diagram below anticipates the overall class relationships that will be discussed throughout the chapter.

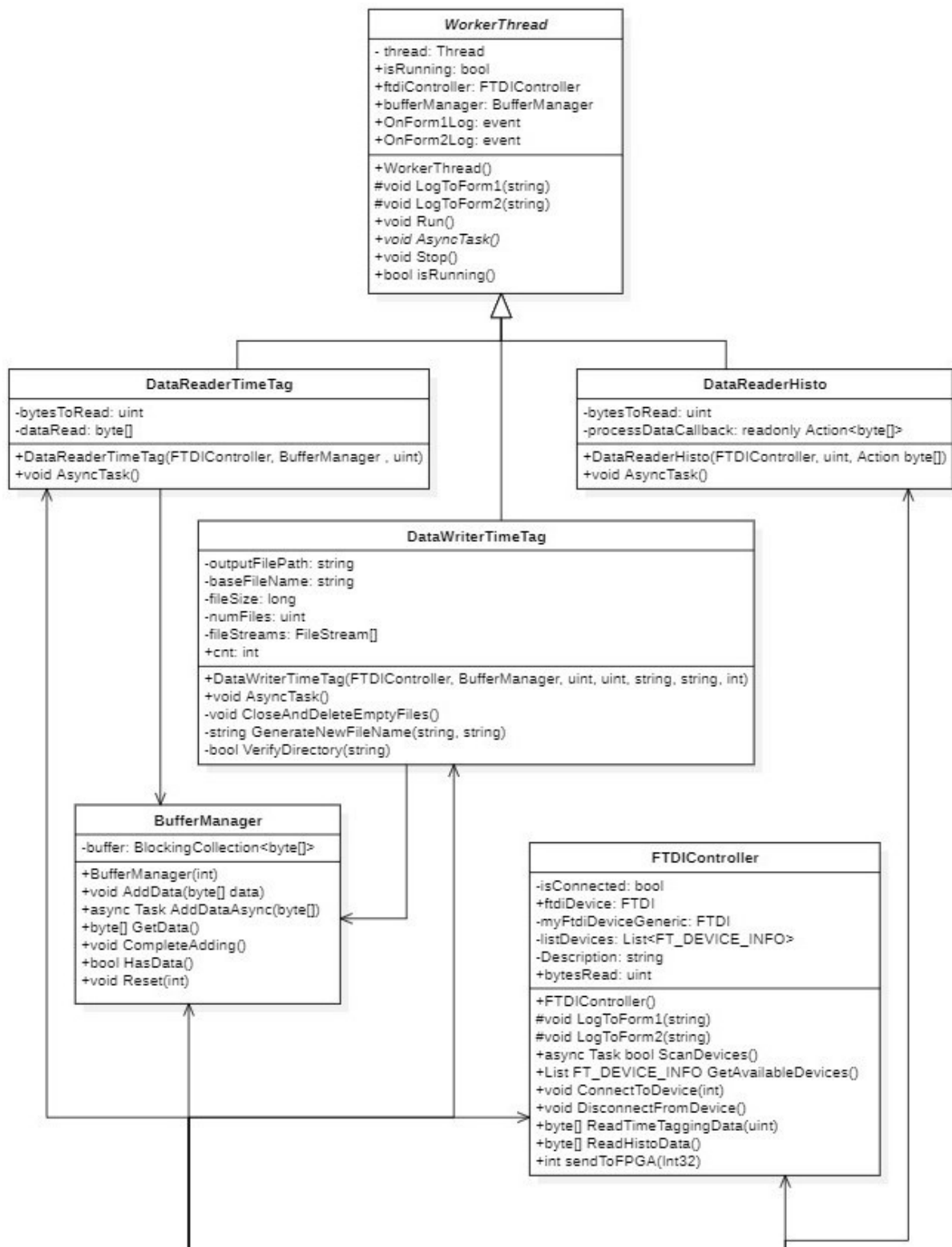


Figure 2.6

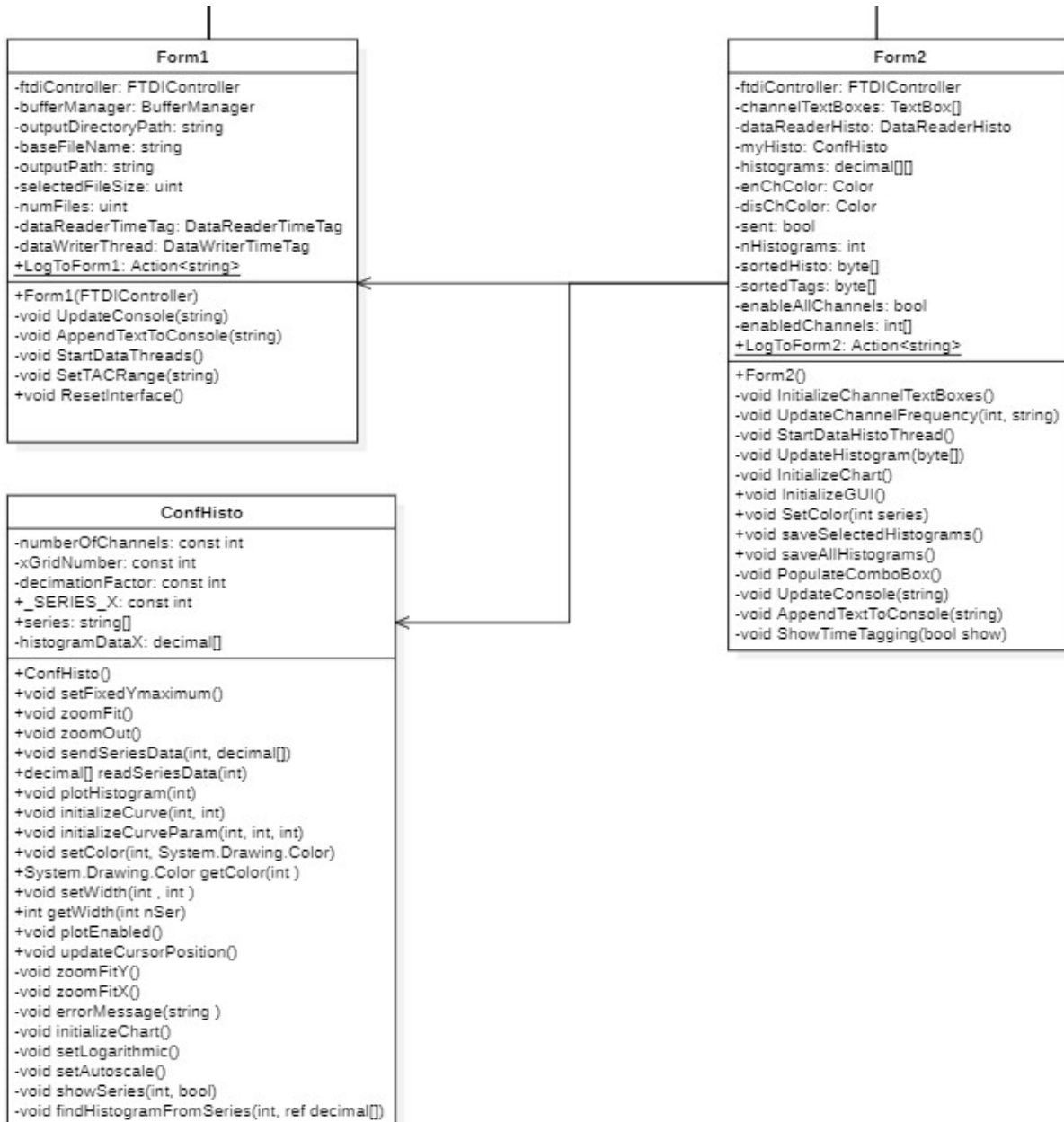


Figure 2.7: UML diagram of the application.

2.4. Implementation Details

The implementation presented in this chapter corresponds to the final, stable version of the software system developed for high-throughput data acquisition from an FPGA board via the FTDI FT601 interface. It is the result of a progressive refinement process involving multiple design iterations, systematic debugging, and extensive runtime testing under realistic data loads.

The following sections offer a detailed breakdown of each class involved in the architecture, analyze thread communication and buffer management strategies, and highlight implementation choices made to guarantee interface responsiveness, fault tolerance, and maintainability in a complex data acquisition pipeline.

2.4.1. Performance-oriented implementation choices

The primary challenge addressed in this work concerns achieving high performance in the time-tagging acquisition mode, where data throughput can reach up to 2.56 Gbps. Ensuring reliable and efficient handling of such a massive data rate represents the key objective of this thesis. In contrast, the histogram acquisition mode demands significantly less bandwidth and thus poses fewer performance constraints. Consequently, most of the design choices, optimizations, and architectural decisions were focussed on meeting the stringent requirements imposed by time-tagging, ensuring data integrity and sustained throughput at near-maximum USB 3.0 bandwidth.

At the hardware level, the use of a pipelined FIFO pyramid within the FPGA reduces congestion and ensures sustained data delivery to the FT601, even under high data rates. This strategy guarantees that data produced by the sensors is not lost due to temporary stalls in the USB interface.

On the software side, the system adopts a multithreaded design that separates data acquisition and file writing into distinct, specialized threads. This structure is coordinated through the use of an intermediate buffer managed by the `BufferManager`, which decouples the data flow between acquisition and storage. The `DataReaderTimeTag` thread performs high-throughput reads from the FTDI device and immediately deposits the data into the buffer, enabling prompt initiation of the next acquisition cycle without delay. Meanwhile, the `DataWriterTimeTag` thread asynchronously retrieves buffered data and writes it to disk, maintaining continuous throughput.

Overall, the implemented design ensures efficient utilization of the USB 3.0 interface by

minimizing communication overhead through block-wise data transfers. This approach supports the sustained high data rates required by the time-tagging acquisition mode, maintaining system responsiveness and throughput without compromising stability.

2.4.2. Class-level design and implementation

The following subsections provide a detailed description of each core class involved in the application. The decision to structure them as indexed entries, rather than using a flat or enumerated list, aims to facilitate readability and quick reference throughout the thesis.

2.4.2.1. Form1

The class `Form1.cs` implements the Graphical User Interface (GUI) and control logic for conducting time-tagging measurements of photon arrivals from a Single-Photon Avalanche Diode detector. This class constitutes the main operational interface of the application in time-tagging mode and is responsible for managing the acquisition workflow from the FTDI FT601 interface, which is connected to a custom FPGA-based backend.

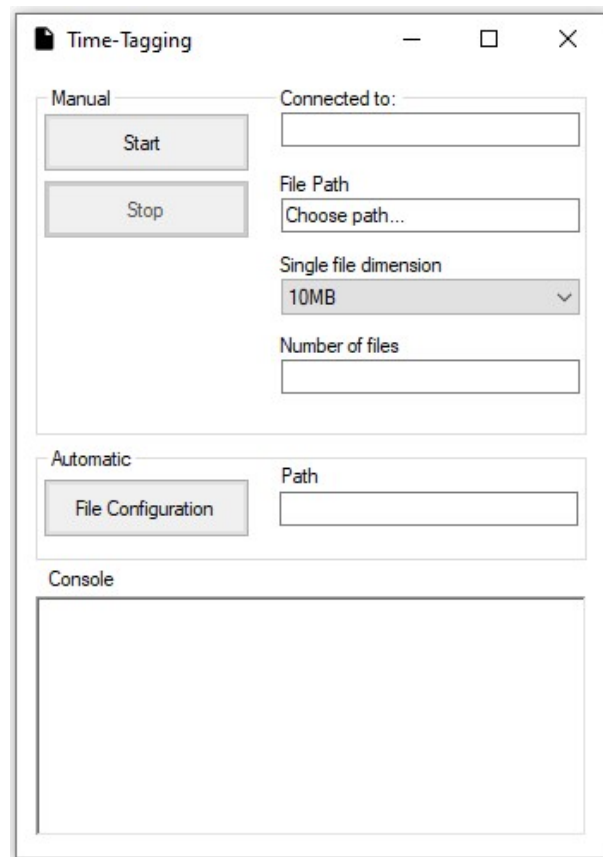


Figure 2.8: Form1 GUI.

Figure 2.8 shows the main interface of the time tagging mode, which easily allows the user to start and stop data acquisition, monitor system status, and manage output files.

Typically, `Form1.cs` is instantiated from `Form2.cs`, which provides an initial environment for signal testing and histogram acquisition. By the time `Form1` is launched, the FTDI device has already been detected, selected, and initialized within `Form2`, and its reference is passed to `Form1` for use in the acquisition process. The connected device is displayed to the user within a read-only textbox and is subsequently used for all data exchanges during the measurement.

The class supports two distinct modes for configuring the acquisition parameters:

- **Manual Configuration Mode:** in this mode, the user is required to manually define all relevant acquisition parameters via dedicated GUI controls:
 - The destination folder for saving output data files, selected through a standard directory browser dialog opened clicking a text box.
 - The data size format, chosen from three predefined options inside a combo box, which determines how the time-tagged data is read, buffered and interpreted.
 - The number of data files to be generated, allowing segmentation of the acquisition stream for long-duration measurements.
- **Automatic Configuration Mode:** This mode allows the user to provide a predefined configuration file containing all acquisition parameters in a fixed, ordered format. Upon selecting the file through a dialog box, the parameters are parsed and automatically applied to the internal configuration.

This functionality was introduced mid-development in response to specific requirements from the Agenzia Spaziale Italiana. Its integration was straightforward thanks to the modular structure of the software, which allowed new features to be added with minimal impact on existing components.

In their laboratories, the researchers employed a dynamic setup involving spatially scanned laser beams for calibration and measurement purposes. In their configuration, the laser beam is automatically repositioned across different spatial coordinates, and a separate time-tagged acquisition must be performed at each position. Given that these experiments are conducted overnight and without human supervision, it was essential to implement a fully automated acquisition routine to avoid an user intervention each time. The automatic configuration mode enables this workflow by allowing the system to load predefined settings, start data acquisition,

and proceed seamlessly from one measurement to the next as the laser position changes through an external control software. This ensures unattended and reliable operations during long acquisition sessions.

Once the configuration has been defined, either manually or automatically, the user can initiate the measurement by pressing the *Start* button. The application then handles real-time data transfer from the FPGA, supporting mid-acquisition termination via the *Stop* button, and writes the acquired time-tagged events directly to file in a structured format. In case of early termination, the data acquired up to that point is preserved and written.

In the following, the methods implemented within the `Form1.cs` class are presented and discussed in detail in terms of functionality, role within the acquisition workflow, and interactions with both the user interface and the hardware backend.

- **Form1(FTDIController ftdiController)**: constructor of the class which initializes components, delegates and event handlers to log messages on the console, configures also UI elements, and the buffer manager. It saves the reference of the FTDI controller passed by Form2.
- **Form1_Load(object sender, EventArgs e)**: event handler executed when the form loads, before it is visible from the user; disables stop button, makes some textboxes read-only, and sets combobox style.
- **Form1_Shown(object sender, EventArgs e)**: event handler executed right after the form is shown for the first time; it deselects any active control.
- **Form1_Activated(object sender, EventArgs e)**: event handler executed everytime the form is reactivated (e.g. restored from taskbar); deselects any active control.
- **UpdateConsole(string message)**: checks if the call comes from the UI thread or from another thread and updates the console with the given message safely, through a *publisher-subscriber* pattern later analyzed in FTDIController class section.
- **AppendTextToConsole(string message)**: helper method that appends a text line to the console and scrolls to the end, enhance the separation of the logic principle.
- **StartDataThreads()**: it validates the output path and the number of files before launching the `DataReaderTimeTag` and `DataWriterTimeTag` threads, both

connected to the user interface via the `OnForm1Log` event for real-time feedback. The control logic ensures a consistent and safe startup sequence, whether triggered by `startButton_Click` or `startFileConfig_Click`.

A counter variable `cnt` is also maintained to track the number of generated files across multiple acquisitions. If the base file name is not changed between measurements, subsequent acquisitions will automatically continue the file numbering (e.g., `file1.dat`, `file2.dat`, `file3.dat`, then `file4.dat`, `file5.dat`, etc.), ensuring no file overwriting occurs and that each dataset remains uniquely indexed.

- **`SetTACRange(string tacRange)`**: sends the command to FPGA to set the TAC Full-Scale range based on the input string and logs success or error.
- **`startButton_Click(object sender, EventArgs e)`**: tries to start data threads, logging the result of this operation.
- **`startFileConfig_Click(object sender, EventArgs e)`**: handler for configuration file selection button; opens a dialog to select a set up .txt file, parses it while setting parameters, and starts acquisition threads.

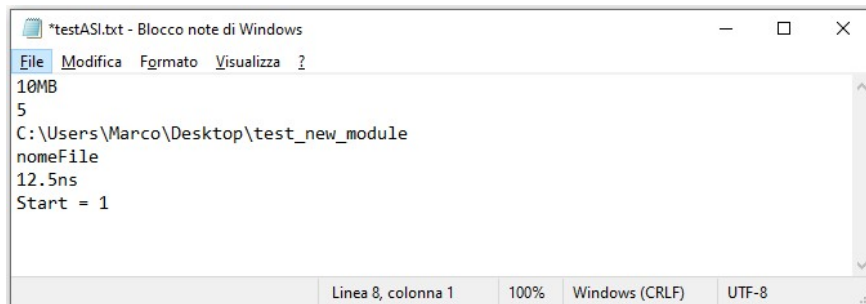


Figure 2.9: Example of .txt file for automatic mode.

This file is trimmed as follows:

1. File size
 2. Number of files to write
 3. Output directory
 4. File name
 5. TAC FSR
 6. Start or Wait
- **`stopButton_Click(object sender, EventArgs e)`**: async handler for the Stop button click which stops data reader thread, aborts FTDI read pipe, resets buffer and interface. After the stoppage, the partial data read is written on the predefined number of files and empty ones are deleted.

- **ResetInterface()**: clears UI elements related to file paths and resets configuration parameters, preparing interface for a new acquisition session. It is called both after the conclusion of a measurement and after an early stoppage.
- **pathBox_Click(object sender, EventArgs e)**: handler for clicking the path textbox which opens a SaveFileDialog to select output path and base file name. The directory location will be shown in the box and, everytime the user changes it, the counter is reset.
- **numFilesBox_TextChanged(object sender, EventArgs e)**: parses the number of files to save.
- **comboBox1_SelectedIndexChanged(object sender, EventArgs e)**: handler for file size combobox changes; sets the selected file size and logs it to console.

2.4.2.2. Form2

The `Form2.cs` class serves as the main interface for histogram-mode data acquisition in this thesis project. It is the first form that appears upon launching the application and provides comprehensive control and monitoring of the histogram acquisition process through a user-friendly graphical interface. Operators can interact with this form to scan and select FTDI-connected devices from a dropdown combo box menu, configure acquisition parameters such as the TAC Full Scale Range (FSR), and choose between linear or logarithmic chart scaling. The interface also includes options to enable or disable autoscaling of the plotted histograms, allowing for tailored visualization depending on the data characteristics.

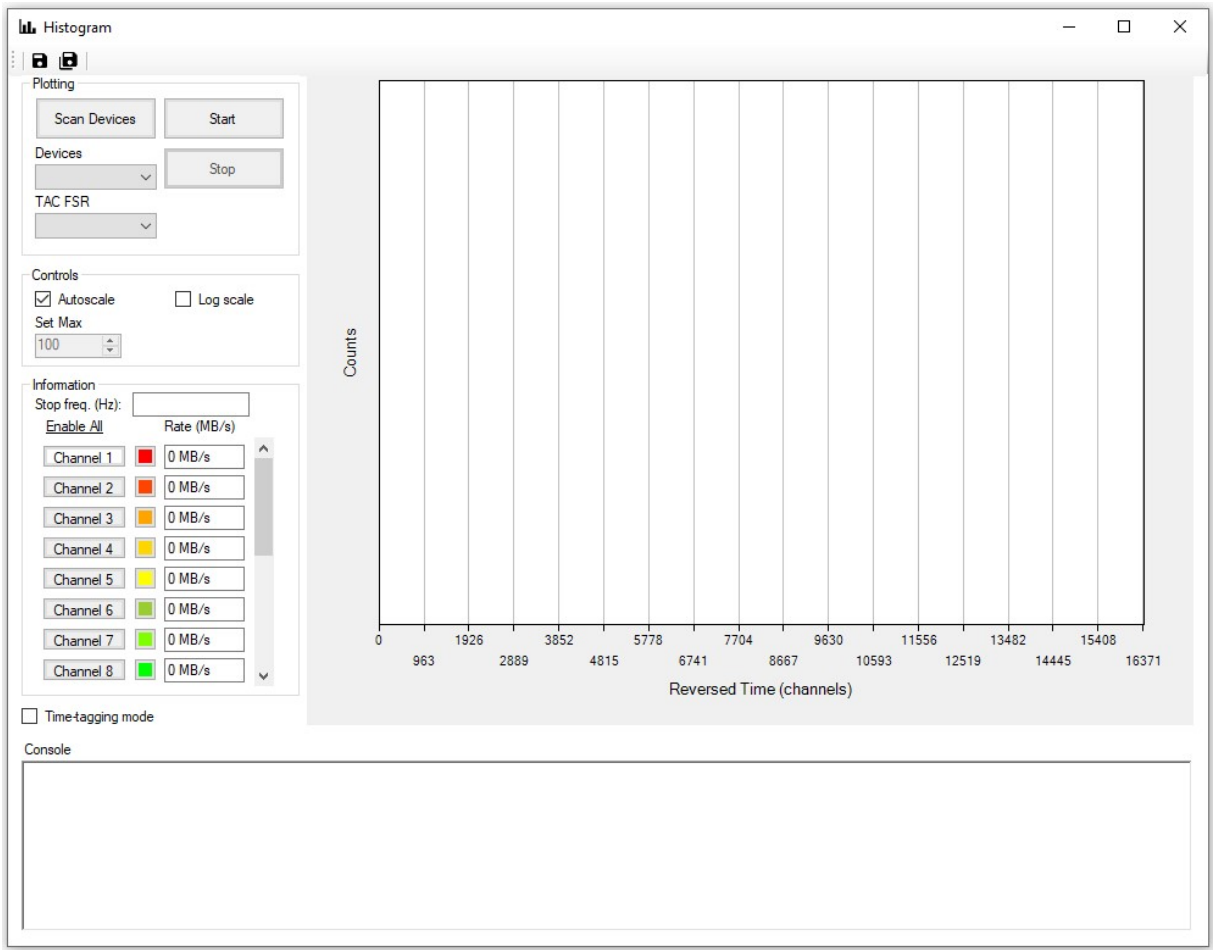


Figure 2.10: Form2 GUI.

A central component of the form is `chart1`, which displays live histograms for up to 16 acquisition channels. Real-time data acquisition rates for each channel are shown in an organized table layout and can be saved, while a dedicated textbox provides feedback on the overall stop frequency. Additionally, users can switch to time-tagging mode by

checking a designated checkbox, which triggers the opening of `Form1`.

Behind the scenes, the form manages the FTDI connection and configures the data acquisition thread via the `DataReaderHisto` class. It ensures that incoming data is processed and visualized correctly, while also maintaining a log of internal messages for debugging and user feedback. The form is responsive to user interaction, dynamically enabling or disabling GUI elements based on the current acquisition state, thereby enhancing both usability and robustness of the application.

- **`Form2()`**: constructor of the class which initializes components, delegates and event handlers to log messages on the console; it configures also UI elements such as the chart and the console.
- **`Form2_Load()`**: load event handler that sets the two drop-down lists named as `deviceComboBox` and `tacFsrComboBox` to read-only mode and disables the stop measurement button until a reading starts.
- **`Form2_Shown(object sender, EventArgs e)`**: event handler that deselects all controls when the form first appears, and sets the initial enabled state of channel 1 (`buttonEnCh1`) with the enabled channel color, in order to display it.
- **`Form2_Activated(object sender, EventArgs e)`**: event handler that deselects all controls everytime the form is reactivated (e.g. restored from taskbar).
- **`InitializeChannelTextBoxes()`**: initializes the dictionary array of 16 text boxes (`textBoxChX`) used to display channel rates and sets their default text to "0 MB/s".
- **`UpdateChannelFrequency(int channel, string frequency)`**: thread-safe method (to avoid to block the UI) that updates the rate display for a specific channel (0-16) using the `channelTextBoxes` array.
- **`StartDataHistoThread()`**: initiates the histogram acquisition thread by first validating the connection and then clearing the FPGA memory. It proceeds to initialize `DataReaderHisto` with a 1MB buffer and sets up the data processing callback, where `processData` defines the actions to take when new data arrives. In each reading cycle, `DataReaderHisto` fills the `bytesToRead` buffer and passes it to `processDataCallback`, which updates the histogram accordingly. Finally, the stop button is enabled once the process has started.
- **`UpdateHistogram(byte[] histoData)`**: this method is responsible for parsing and processing a raw byte array, `histoData`, which contains multiplexed his-

togram data from multiple acquisition channels, along with associated metadata such as flags, tags, and frequency counters.

The method starts by scanning the array to identify the correct starting offset of the actual histogram data skipping over non-relevant data segments such as headers or frequency metadata. It does this by searching for a specific marker byte (e.g., `0x6F` in the fourth byte of a bin), which indicates the beginning of valid histogram content.

Once the correct starting point is located, the method extracts two key data blocks:

- **Histogram Data (`sortedHisto`)**: a fixed-size array that stores the raw histogram bin values for each acquisition channel, each of which encoded using 4 bytes, where the first three bytes represent a 24-bit count value. This method reconstructs the photon count by bit-shifting and aggregating these bytes. In the end as a preprocessing step, the first 4 bytes (first bin) of each channel are zeroed to remove any potential initial peak artifacts.
- **Tag Data (`sortedTags`)**: the 1024 bytes following the histogram data, including metadata such as the stop frequency counter and the strobe frequency counters for each channel, are then copied into this array.

The **stop frequency** is calculated from three specific bytes within `sortedTags`, combined using bitwise operations, scaled appropriately and displayed in a UI textbox using a thread-safe `Invoke` call. Then, for each acquisition channel, the method computes the **strobe frequency** from a corresponding 3-byte sequence within `sortedTags` and displayed through the helper method `UpdateChannelFrequency`.

The histogram data for each channel is stored in an intermediate array structure `histograms[i]`, which is populated by decoding 3-byte groups from `sortedHisto`. This array holds the count values for each bin and is passed to the method `sendSeriesData`, which internally maps the data to the appropriate series in the histogram handler object `myHisto`. To visualize the data, the method invokes `plotHistogram(i)` within a UI-thread-safe block for each channel, ensuring proper rendering without cross-threading issues.

Finally, if the `sent` flag is still `false`, the method sends two stop commands to the FPGA using `ftdiController.sendToFPGA(...)` to halt acquisition and data transmission. It then sets `sent` to `true`.

- **InitializeChart()**: configures the chart by allocating memory for 17 histograms, initializing `ConfHisto` with parametric ramp settings, and setting up default visualization properties for all series.
- **InitializeGUI()**: sets up the initial GUI state by configuring all channel enable buttons (`buttonEnChX`), color buttons (`buttonColorX`), and chart display options (autoscale enabled by default). Links UI elements to their corresponding `ConfHisto` properties.
- **SetColor(int series)**: opens a color dialog to change the display color for a specific channel, it updates both the color button appearance and the corresponding chart series color.
- **saveSelectedHistograms()**: saves enabled channels to a user-selected text file. Writes 16384 lines per enabled channel with header markers and uses the `enabledChannels` array to determine which channels to save.
- **saveAllHistograms()**: saves all 16 standard channels to a text file regardless of enabled state. Uses the same format as `saveSelectedHistograms` but includes all channels sequentially.
- **PopulateComboBox()**: scans available FTDI devices using `FTDIController` and populates `comboBox1` with device descriptions. Clears existing items and handles cases where no devices are found.
- **UpdateConsole(string message)**: thread-safe logging method that appends messages to `richTextBox1`. Uses `Invoke/BeginInvoke` pattern for cross-thread operations and automatically scrolls to the newest message.
- **AppendTextToConsole(string message)**: helper method that actually performs the text append operation on `richTextBox1` which is called by `UpdateConsole` after thread safety checks.
- **ShowTimeTagging(bool show)**: manages the `Form1` lifecycle: it opens and positions `Form1` to the right of `Form2`, sets `form1alive` flag to keep track of its life and handles proper cleanup when closing.
- **Form1Closure(object sender, EventArgs e)**: callback method (method activated by an event) that resets the time-tagging `checkBox1` when `Form1` is closed, maintaining UI state consistency.
- **scanButton_Click(object sender, EventArgs e)**: asynchronous event handler that scans for FTDI devices when the scan button is clicked. Updates

comboBox1 with the devices found and provides feedback through the console log.

- **deviceComboBox_SelectedIndexChanged(object sender, EventArgs e):** handles device selection changes by disconnecting any existing connection and establishing a new connection to the selected FTDI device index.
- **tacFsrComboBox_SelectedIndexChanged(object sender, EventArgs e):** configures the TAC FSR setting by sending the appropriate FPGA command (0xF1–F4) based on the selected time range.
- **startButton_Click(object sender, EventArgs e):** starts the thread which acquires the histogram data and enables the stopButton to stop the measurement.
- **stopButton_Click(object sender, EventArgs e):** disables the stopButton and stops the histogram acquisition by setting the isRunning flag to false.
- **timeTagCheckBox_CheckedChanged(object sender, EventArgs e):** toggles time-tagging mode by showing/hiding Form1. Stops any ongoing histogram measurements and maintains thread safety by checking dataReaderHisto null state.
- **logarithmicCheckBox_CheckedChanged(object sender, EventArgs e):** toggles logarithmic Y-axis scaling on the chart by updating the flag isYlogarithmic and providing an immediate visual feedback.
- **autoScaleCheckBox_CheckedChanged(object sender, EventArgs e):** controls autoscaling behavior by setting isYautoscale and enabling the manual maximum control numericUpDown1 accordingly.
- **numericUpDown1_ValueChanged(object sender, EventArgs e):** updates the fixed Y-axis maximum value in ConfHisto when the user changes the numeric up/down control. This control is active only when autoscaling is disabled.
- **enableAllLinkedLabel_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e):** toggles all channels on/off when the "Enable All"/"Disable All" link is clicked. Updates both the ConfHisto visibility flags and button background colors (enChColor/disChColor).
- **buttonColorX_Click(object sender, EventArgs e):** individual color change handlers for each channel that call SetColor() with the corresponding series index.

- **buttonEnChX_Click(object sender, EventArgs e)**: handles channel enable/disable toggles and updates both the `ConfHisto` visibility flags and the button background colors for their respective channels.

2.4.2.3. FTDIController

The `FTDIController` class constitutes the fundamental communication layer for FTDI device management within the `ASI_module_FT601` system. As the architectural centerpiece bridging hardware interactions and software components, it implements an efficient *publisher-subscriber* pattern that enables decentralized event logging across the application's UI components (specifically `Form1` and `Form2`) and worker threads. This design provides robust functionality spanning several critical domains: device discovery and connection lifecycle management, dual-mode data acquisition (supporting both time-tag and histogram operational modes through specialized read operations), synchronous FPGA command transmission, comprehensive status monitoring, and cross-component event propagation.

Built upon the FTDI D3XX library's native capabilities, the controller abstracts low-level USB protocol details while maintaining thread-safe operation characteristics essential for modern instrumentation systems. This abstraction enables seamless integration with both foreground UI interactions and background processing tasks, particularly crucial for high throughput data acquisition scenarios. The implementation notably employs two dedicated event channels (`OnForm1Log` and `OnForm2Log`) that implement a decoupled logging mechanism, following established *publisher-subscriber* patterns. This architectural choice significantly reduces coupling between hardware control logic and presentation layers, being a design approach that aligns with contemporary best practices for scientific instrumentation software architectures. The event-driven paradigm proves particularly advantageous in distributed systems where multiple consumers may require concurrent access to status updates and diagnostic information, without creating direct dependencies on the communication controller's implementation details.

- **ScanDevices()**: asynchronously scans for available devices and creates the `list Devices`, which will be used for choosing the device to connect and to populate the UI combo box.
- **GetAvailableDevices()**: getter method which retrieves the cached list of detected devices.
- **ConnectToDevice(int deviceIndex)**: establishes connection to a specific FTDI device by index, after checking the list obtained by the scansion.

- **DisconnectFromDevice()**: safely terminates the active FTDI connection. Used when switching from a device to another.
- **ReadTimeTaggingData(uint bytesToRead)**: synchronous read operation for **time-tag** mode exploiting ReadPipe FTDI method.
- **ReadHistoData(uint numBytesToRead)**: synchronous read operation for **histogram** mode exploiting ReadPipe FTDI method.
- **OnForm1Log** and **OnForm2Log**: public event delegates of type `Action<string>` that implement the publisher-subscriber pattern, allowing the two Form classes to subscribe for log messages respectively. Initialized with empty delegates to ensure null safety.
- **LogToForm1(string message)**: protected method that safely invokes the `OnForm1Log` event using the null-conditional operator, delivering messages to Form1's console without class name prefixes for cleaner output.
- **LogToForm2(string message)**: protected method that similarly invokes the `OnForm2Log` event, providing dedicated message routing to Form2's console while maintaining the same invocation safety features.
- **sendToFPGA(Int32 number)**: Writes 4-byte commands to FPGA as shown in the table 2.1.

Command	Function
0xF4	Set TAC range to 100 ns
0xF3	Set TAC range to 50 ns
0xF2	Set TAC range to 25 ns
0xF1	Set TAC range to 12.5 ns
0xD0	Clear RAM from previous measurement
0xD1	Enable measurement
102	Stop measurement
45	Stop download

Table 2.1: Command reference for `sendToFPGA`.

2.4.2.4. WorkerThread

The `WorkerThread` abstract class serves as a foundational framework for creating specialized thread implementations in this thesis project. This architectural design provides a standardized approach to thread management while enabling specific functionalities through inheritance and encapsulating common thread operations while delegating task-specific behavior to derived classes through the `AsyncTask()` abstract method.

The class establishes several critical components for thread management:

- A `Thread` object that represents the actual execution thread.
- A boolean `isRunning` flag for thread state management.
- Shared resources including an `FTDIDController` for device communication.
- A `BufferManager` for efficient data handling.
- Comprehensive logging infrastructure with dual output channels (`Form1` and `Form2`).

The inheritance mechanism allows three specialized thread types to extend this base functionality:

- `DataReaderTimeTag` for high-speed time-tag data acquisition, written on the intermediate buffer.
- `DataReaderHisto` for histogram data collection and processing.
- `DataWriterTimeTag` for efficient file-based data storage, got from the buffer.

As already specified in the preceding sections, each derived class implements the `AsyncTask()` method with specific behavior while inheriting the robust thread management capabilities from the base class. The design follows the principle of *separation of concerns*, where the base class handles thread lifecycle management while derived classes focus on their specialized data processing tasks.

In **time-tagging mode**, the data flow begins with `DataReaderTimeTag`, which reads raw data directly from the FPGA and immediately writes it into an intermediate buffer managed by `BufferManager`. This operation is designed to be non-blocking, allowing `DataReaderTimeTag` to promptly initiate a new acquisition cycle without waiting for the data to be written to disk. Concurrently, `DataWriterTimeTag` monitors the buffer: as soon as new data becomes available, the thread retrieves it, segments it into the appropriate number of output files, and performs the disk write operations accordingly.

In contrast, in **histogram mode**, the acquisition is handled by `DataReaderHisto`, which also reads a different type of data from the FPGA in a similar way. However, instead of writing to disk, it forwards each block of acquired data to a predefined callback function which is responsible for processing the incoming data and updating the histogram visualization in real-time. This decoupled architecture ensures continuous data acquisition with minimal latency while maintaining responsiveness.

The logging system deserves special attention from thread perspective, as it implements a *publisher-subscriber* pattern through C# events which allows UI components to subscribe to log messages without creating tight coupling between the worker threads and the presentation layer. The `OnForm1Log` and `OnForm2Log` events provide flexible communication channels for status updates and debugging information.

DataReaderTimeTag Thread

- **AsyncTask()**: the core time-tagging data acquisition engine handles sustained 2.56 Gbps (320 MB/s) transfers from the FTDI device through direct calls to `ReadTimeTaggingData()`, which, thanks to the intermediate buffer, don't need to wait for precedings writing on disk. This synchronous implementation ensures deterministic timing for high-speed operations while maintaining data integrity through a two-stage error handling system, first capturing device-level communication faults during the FTDI read operation, then managing buffer-related exceptions during the subsequent transfer to the shared `BufferManager`. The method guarantees an explicit error messages management, routed to both console and Form1's logging interface.

```
// Read from FTDI and writes on file
2 riferimenti | 0 modifiche | 0 autori, 0 modifiche
public override void AsyncTask()
{
    try
    {
        dataRead = ftdiController.ReadTimeTaggingData((UInt32)bytesToRead); // Reading function
    }
    catch (Exception ex)
    {
        ftdiController.ftdiDevice.AbortPipe(0x82); // Abort reading
        Console.WriteLine($"Error in ReadTimeTaggingData: {ex.Message}"); // Hidden debug message (for the developer)
    }

    try
    {
        bufferManager.AddData(dataRead); // Add data to the intermediate buffer
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error in AddData: {ex.Message}"); // Hidden debug message (for the developer)
    }

    LogToForm1("DataReaderTimeTag thread terminated."); // Console debug message
}
```

Figure 2.11: AsyncTask method for DataReaderTimeTag

DataReaderHisto Thread

- **AsyncTask()**: the histogram acquisition engine operates in a continuous polling loop during thread execution, maintaining a steady 1 Hz refresh rate to balance UI responsiveness with processing overhead. Data flows through a callback pipeline to Form2's visualization system, where the processDataCallback delegate handles real-time histogram rendering. The implementation enforces robust termination protocols, catching both cancellation requests and hardware exceptions while maintaining isolated logging through Form2's dedicated event channel. Error states trigger immediate loop termination with diagnostic messages while preserving system stability. As a final note it should be remembered that, if the ReadPipe in DataReaderTimeTag is interrupted by the AbortPipe coming from the Stop button, the thread will write the partial data correctly without any error, erasing empty files.

```
// Read histo data from FTDI
2 riferimenti | 0 modifiche | 0 autori, 0 modifiche
public override void AsyncTask()
{
    byte[] histoDataRead = new byte[bytesToRead]; // Reading dimension

    LogToForm2("Starting a new measurement..."); // Console debug message

    // Cyclic update
    while (IsRunning())
    {
        try
        {
            histoDataRead = ftdiController.ReadHistoData(bytesToRead); // Reading function

            processDataCallback(histoDataRead); // Pass the data to the callback for processing
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("Histo data reading task canceled."); // Hidden debug message (for the developer)
            break;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error in DataReaderHisto: {ex.Message}"); // Hidden debug message (for the developer)
            break;
        }
    }
}
```

Figure 2.12: AsyncTask method for DataReaderHisto

DataWriterTimeTag Thread

- **AsyncTask()**: the file management system employs an advanced pre-allocation strategy to handle large datasets, segmenting data streams into fixed-size files for efficient storage management and analysis through the MatLab software. Whenever data becomes available for writing, the Array.Copy method is employed to perform low-overhead memory-to-disk transfers. Resource management is handled via a finally block, which guarantees the proper release of file handles and associated

system resources, ensuring robustness and preventing resource leaks. Comprehensive logging provides real-time feedback including exact byte counts written and file creation events, with automatic detection and removal of empty files to maintain storage efficiency.

```
// Writing task of DataWriter thread
2 references | 0 modifications | 0 authors, 0 modifications
public override void AsyncTask()
{
    VerifyDirectory(outputFilePath); // Check if the path exists

    try
    {
        long totalBytesWritten = 0; // Bytes written counter

        byte[] dataRead = bufferManager.GetData(); // Gets data from the buffer (different from zero)

        // Creation of all the files, done here to consent subsequent reads
        FileStream[] fileStreams = new FileStream[numFiles];
        for (int i = 0; i < numFiles; i++)
        {
            string newFilePath = GenerateNewFileName(outputFilePath, baseFileName);
            cnt++;
            fileStreams[i] = new FileStream(newFilePath, FileMode.Create, FileAccess.Write, FileShare.ReadWrite);
            Console.WriteLine($"Created new file: {newFilePath}"); // Hidden debug message (for the developer)
        }

        for (int fileIndex = 0; fileIndex < numFiles; fileIndex++)
        {
            long offset = fileIndex * fileSize; // Offset computation for each file (offset points to the beginning of a file)

            long remainingBytes = ftdiController.bytesRead - offset; // Stop button pressed?

            if (remainingBytes <= 0)
                break; // If no more data to write

            // Choose the currentChunkSize: how many bytes are available after X writes? If there are less data than the fileSize, take only the one we have
            int currentChunkSize = (int)Math.Min(fileSize, remainingBytes);

            // Extract the corresponding chunk from the buffer
            byte[] chunk = new byte[currentChunkSize];
            Array.Copy(dataRead, offset, chunk, 0, currentChunkSize);

            // Write the chunk in the corresponding file
            fileStreams[fileIndex].Write(chunk, 0, chunk.Length);
            fileStreams[fileIndex].Flush(); //Ensure data are written on disk

            // Check how many bytes has been written
            totalBytesWritten += chunk.Length;
        }

        LogToForm1($"Wrote total {totalBytesWritten} bytes to files."); // Console debug message
    }
    catch (Exception ex)
    {
        LogToForm1($"Error writing files: {ex.Message}"); // Console debug message
    }
    finally
    {
        CloseAndDeleteEmptyFiles();
        LogToForm1("DataWriter thread terminated."); // Divide the console for subsequent readings
        LogToForm1("-----");
    }
}
```

Figure 2.13: AsyncTask method for DataWriterTimeTag

- **CloseAndDeleteEmptyFiles()**: performs comprehensive file resource cleanup identifying and removing empty files to conserve storage.
- **GenerateNewFileName(string directoryPath, string baseFilePath)**: creates sequentially numbered filenames combining directory path with base filename and counter.
- **VerifyDirectory(string path)**: validates directory existence and write permissions, creating and immediately deleting a test file to verify filesystem access. Handles various exception cases including unauthorized accesses.

2.4.2.5. BufferManager

The `BufferManager` class serves as a critical **intermediary component** in a *producer-consumer* scenario between two threads: `DataReaderTimeTag` (producer) and `DataWriterTimeTag` (consumer). Its primary role is to decouple the reading and writing operations, allowing them to operate asynchronously and independently. This architectural decision brings several key benefits that enhance the system's performance, reliability, and maintainability.

First and foremost, the `BufferManager` improves the **responsiveness** of the user interface: by introducing an intermediate buffer, if the user wants to do some sequential reads, every `DataReaderTimeTag` next thread can start reading data blocks without being blocked by the file-writing operations performed by `DataWriterTimeTag`. This capability becomes especially crucial in scenarios where the writing process might encounter delays due to disk I/O bottlenecks, massive data transfers, or other system constraints. In the absence of this buffering mechanism, the data acquisition pipeline would enforce strict synchronization between read and write operations, resulting in UI thread blocking during write operations and consequently degrading the application's perceived performance and interactivity.

Another significant advantage is **fault isolation**: if the writing process fails (e.g., due to disk full errors or permission issues), the reading process can continue unaffected, with the buffered data remaining available for potential recovery or retry scenarios. This makes the system more robust compared to a tightly coupled implementation where a write failure would immediately disrupt the entire data acquisition pipeline.

The choice of using a `BlockingCollection` as the underlying buffer implementation was driven by the thread safety property it provides, eliminating the need for manual synchronization mechanisms when multiple threads access the buffer concurrently. This **thread-safe** nature ensures that the producer and consumer can work simultaneously without risking data corruption or race conditions, while also handling the complexities of thread signaling and blocking automatically.

Considering a software design perspective, the `BufferManager` class adheres to the **Single Responsibility Principle** by focusing exclusively on buffer management and by splitting the duties of the reading and writing threads. This separation of concerns makes the system more maintainable and easier to modify, for instance, if the buffering strategy needs to change (perhaps to implement priority queues, circular structures or different push-pull policies), the modification can be localized to this class without affecting the reading or writing components.

The buffer also serves as a shock absorber between components that may operate at different speeds, in fact, the `DataReader` might produce data in bursts, while the `DataWriter` might have more consistent but slower write speeds. The buffer smooths out these timing differences, preventing data loss during temporary speed mismatches.

Additionally, the `BufferManager` provides valuable flexibility in system configuration: the capacity parameter in the constructor allows tuning the system's memory footprint versus its tolerance for speed variations. Indeed, a larger buffer can accommodate longer periods where the producer outpaces the consumer, while a smaller buffer might be preferred in memory-constrained environments.

- **`BufferManager(int capacity)`**: the constructor initializes the underlying `BlockingCollection` with a specified capacity which determines how many byte arrays the buffer can hold before the producer threads will block when trying to add new items.
- **`AddData(byte[] data)`**: this synchronous method adds a byte array to the buffer after first checking for null or empty arrays, which are ignored. It also implements blocking semantics when the buffer reaches maximum capacity, entering a wait state until consumer threads free space through dequeuing operations. This flow control mechanism enforces automatic rate limiting when consumer throughput is insufficient to maintain pipeline equilibrium and guarantees that memory usage remains within deterministic limits, preventing uncontrolled heap expansion. Adding a debug-level console output here is fundamental for monitoring buffer saturation states during development and testing phases.
- **`GetData()`**: this is the basic consumption method that retrieves and removes a byte array from the buffer and, if it's empty, the call will block until data becomes available. This simple blocking behavior is ideal when the consumer can afford to wait indefinitely for new data. The method throws an `InvalidOperationException` if `CompleteAdding` has been called and the buffer is empty, signaling that no more data will arrive.
- **`CompleteAdding()`**: this method signals that no more items will be added to the buffer. It's crucial for proper pipeline shutdown, allowing consumers to know when they've processed all available data rather than waiting indefinitely for more and, after this call, attempts to add more data will throw an exception, while consumers can continue to retrieve remaining items until the buffer is empty.

- **IsAddingCompleted (Property)**: this read-only property exposes whether `CompleteAdding` has been called on the buffer. It's useful for consumers to check whether they should expect more data or can terminate their processing loops.
- **HasData()**: a convenience method that checks both whether the buffer has items available and whether adding has been completed. This provides a more complete picture of the buffer's state than just checking the count, as it considers both current contents and future expectations.
- **Reset(int capacity)**: reinitializes the buffer with a new capacity, effectively clearing all existing items and state. This allows for buffer reuse in scenarios where the processing pipeline needs to be restarted (e.g. after a stop interruption).

2.4.2.6. ConfHisto

The `ConfHisto` class represents an histogram visualization and analysis tool that was reverse-engineered and adapted for use in a signal testing and setup configuration environment. Then it was specifically connected to `Form2`, which serves as a preliminary testing phase before entering the time-tagging mode, allowing users to verify the quality and characteristics of incoming signals. Without any formal documentation or datasheet about the histogram data structure, it was employed a **black-box** reverse engineering approach to understand and utilize this class effectively, demonstrating how modularity and practical engineering often involves working with imperfect information and adapting existing solutions to new contexts.

The class is designed to handle multiple histogram data series (up to 17 distinct series) with a fixed X-axis length of 16384 channels, tailored for high-resolution spectral analysis. Each series can be individually controlled for visibility, color, and line width, providing flexibility in visualizing multiple data streams simultaneously.

The milestone of the class is the `myChart` object, which is an instance of the Windows Forms Chart Control (`System.Windows.Forms.DataVisualization.Charting.Chart`), a versatile tool for rendering dynamic data visualizations. In this application, it serves as the graphical engine for displaying histograms, with its properties and methods algorithmic configured to adapt to varying data densities and zoom levels.

The control's `ChartArea` defines the plotting region, where axis scaling, grid intervals, and data decimation are dynamically adjusted, ensuring optimal representation even with large datasets. The `plotHistogram` method leverages `myChart` to clear, rescale, and redraw histogram bars, handling edge cases like logarithmic scaling and decimation-driven

peak detection. Its integration demonstrates a balance between automated rendering and manual fine-tuning, particularly in managing axis intervals and data point insertion.

The use case in this thesis project involves printing these histograms every second during the testing phase, allowing real-time monitoring of signal characteristics to adjust the hardware setup. The class properties like `isYlogarithmic` and `isYautoscale` are used to dynamically adjust the visualization based on signal properties, while methods like `plotEnabled()` ensure only active series are rendered.

The implementation shows careful consideration of performance in the plotting methods, especially when handling the large 16384-channel datasets, and includes error handling for edge cases like zoom-induced index out-of-range exceptions. The cursor position tracking and zoom management features provide interactive analysis capabilities directly on the chart. Through this class, it has been created an effective signal diagnostic tool that, despite its origins in a different context, now serves as a critical component in the application’s signal verification workflow, demonstrating the value of adaptable, reverse-engineered solutions in specialized engineering applications.

The adjusted and used methods are the following:

- **`sendSeriesData(int nSer, decimal[] histogramData)`**: this function is designed to store the retrieved histogram data of one specific channel into a dedicated internal data structure based on their serial number, i.e. `histogramDataX`. Each array is then used for reading and searching purposes in following methods.

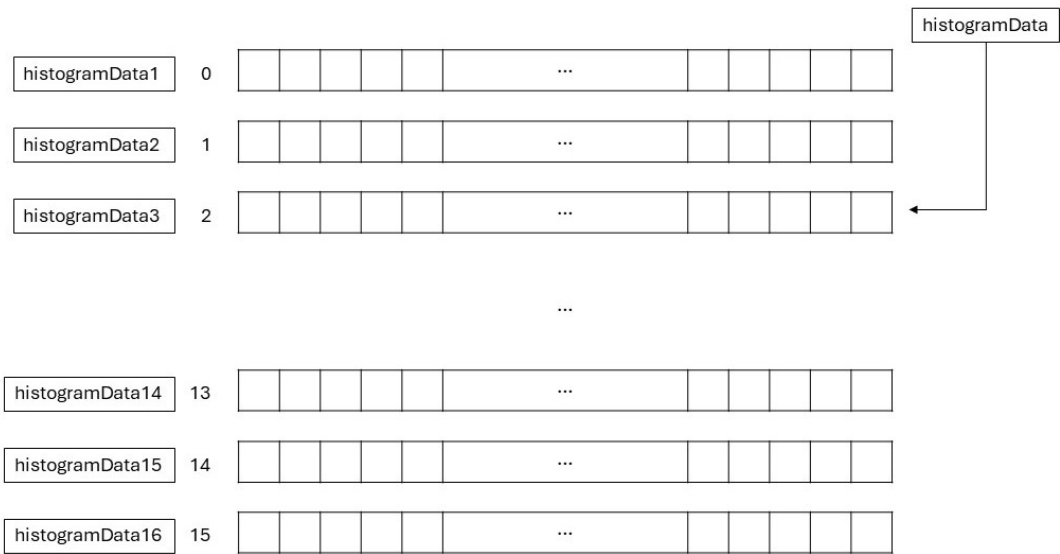


Figure 2.14: Insertion of the `histoData` of a specific channel in the **internal structure**.

- **readSeriesData(int nSer)**: switch method which returns the histogram Data of a specific data series, filled by the preceding method. This is crucial when saving all or selected histograms.
- **findHistogramFromSeries(int nSer, ref decimal[] histogramData)**: switch method which allows to retrieve the histogram data of a specific data series from the histogramDataX array structure, returning it inside the passed parameter histogramData, ready for being plotted.
- **plotHistogram(int nSer)**: after every auxiliary data structure has been filled through sendSeriesData, this method selects the histogram to plot by means of findHistogramFromSeries. This method was used as a **black-box** since it doesn't exist any documentation about the type of data: it employs functions from the Windows Forms Chart Control.
- **initializeCurveParam(int nSer, int curve, int param)**: this method generates and displays a **default data series** for each channel based on a specified curve type and parameter. It creates a local decimal array sized to the number of channels, then fills it either with a ramp or a constant value depending on the curve type chosen. After building the data, it calls sendSeriesData to update the internal histogram data for the specified series and then immediately calls plotHistogram to visualize the result.
- **showSeries(int nSer, bool value)**: this helper method enables or disables the visualization of a specific data serie, plotting it.
- **public bool showSeriesX**: this property controls whether a histogram series is displayed. When set, it updates a backing field and calls showSeries() to plot and refresh the UI. In Form2, these properties are checked in InitializeGUI() to set corresponding button colors (enChColor for enabled, disChColor for disabled), linking the data state directly to the visual presentation also by means of the relative buttons.
- **setColor(int nSer, System.Drawing.Color color)**: helper method which exploits the Color property of the Chart type namespace.

2.4.3. Robustness: exception handling and error recovery

Robustness has been a central focus throughout the development of the application: the system anticipates and manages potential points of failure both at the communication and file-handling level, providing debug messages both for the developer (back-end console) and for the user (front-end console). Exception handling is implemented around all critical operations, including device initialization, data reading, and file writing, ensuring that meaningful error messages are logged and propagated to the user when needed.

If an error occurs during the data transfer, the system logs the failure and ensures that any data read up to that point is preserved and written, thus preventing the loss of valuable information. Although the `ReadPipe` function may fail abruptly due to a device or communication issue, the modular structure of the software helps isolate faults by decoupling acquisition, buffering, and file writing. If necessary, the `AbortPipe` function can be invoked explicitly to cancel ongoing transfers, but this action renders the pipe unusable until a subsequent `ResetPipe` is performed. This architecture facilitates controlled recovery and helps maintain the FTDI channel in a consistent state.

Moreover, the interface is designed to detect and report device disconnections, file access issues, and unexpected states in the communication channel. This increases the reliability of the system during prolonged acquisitions, where the integrity of the data and the stability of the software are critical.

3 | System Validation Framework and Future Directions

This chapter presents the experimental setup and the validation process for the photon counting software developed in this thesis. The first section offers a detailed overview of the measurement configuration, describing the instruments and hardware modules used to test the acquisition system, including the specific key components analyzed in the previous chapters, such as the FTDI FT601 interface, the Xilinx FPGA board, and auxiliary equipment such as an oscilloscope and an impulse generator. After this, the system's functional behavior is analyzed in both histogram and time-tagging acquisition modes, with particular emphasis on the software validation methodology comprising data integrity checks and the utilization of a MATLAB script to interpret and verify time-tagging results. Finally, the chapter discusses potential improvements to the system, focusing on FPGA multi-boot capabilities and software enhancements aimed at increasing flexibility and automation.

3.1. Hardware Measurement Setup

This section describes the hardware configuration employed for the measurement and testing phases of the system. The hardware setup is constituted to generate and provide the START and STOP input signals required by the system under test. All components are configured to ensure precise timing, synchronization, and signal integrity, thereby enabling accurate data acquisition and subsequent analysis. Particular attention is given to the reproducibility and stability of the input signals, which are essential for validating the system's performance.

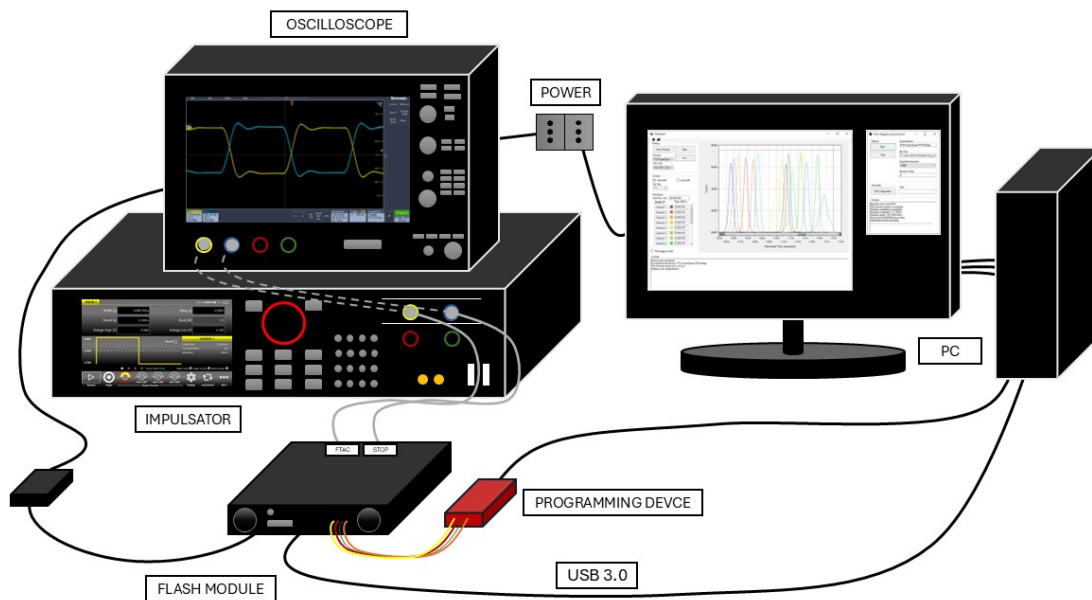


Figure 3.1: Complete hardware testing setup.

The first step in configuring the hardware measurement setup involves powering on and connecting the FLASH acquisition module to the PC via a USB 3.0 interface [12]. This system consists of two modules: the detection head, which houses the detector and its dedicated readout electronics; and the acquisition (timing) module, which is employed in the present research work for the development of the firmware for time-tagging mode. Since the system is used exclusively in its acquisition component, the analysis will focus solely on this one.

The FLASH system integrates both the timing conversion logic and the data transmission functionality. The timing conversion section is responsible for measuring the temporal delay between photon detection events and a reference signal, following the reversed START-

STOP scheme typical of TCSPC setups. This structure supports advanced modes such as the Fast-TAC (F-TAC), where multiple TACs operate in parallel to handle high event rates without introducing distortion due to dead time. The FPGA within the module processes the digitized timing information, generating either histograms or time-tagged event data depending on the configuration done with the Vivado firmware. The processed data are then transferred to the PC via the FTDI601 using the USB 3.0 link.

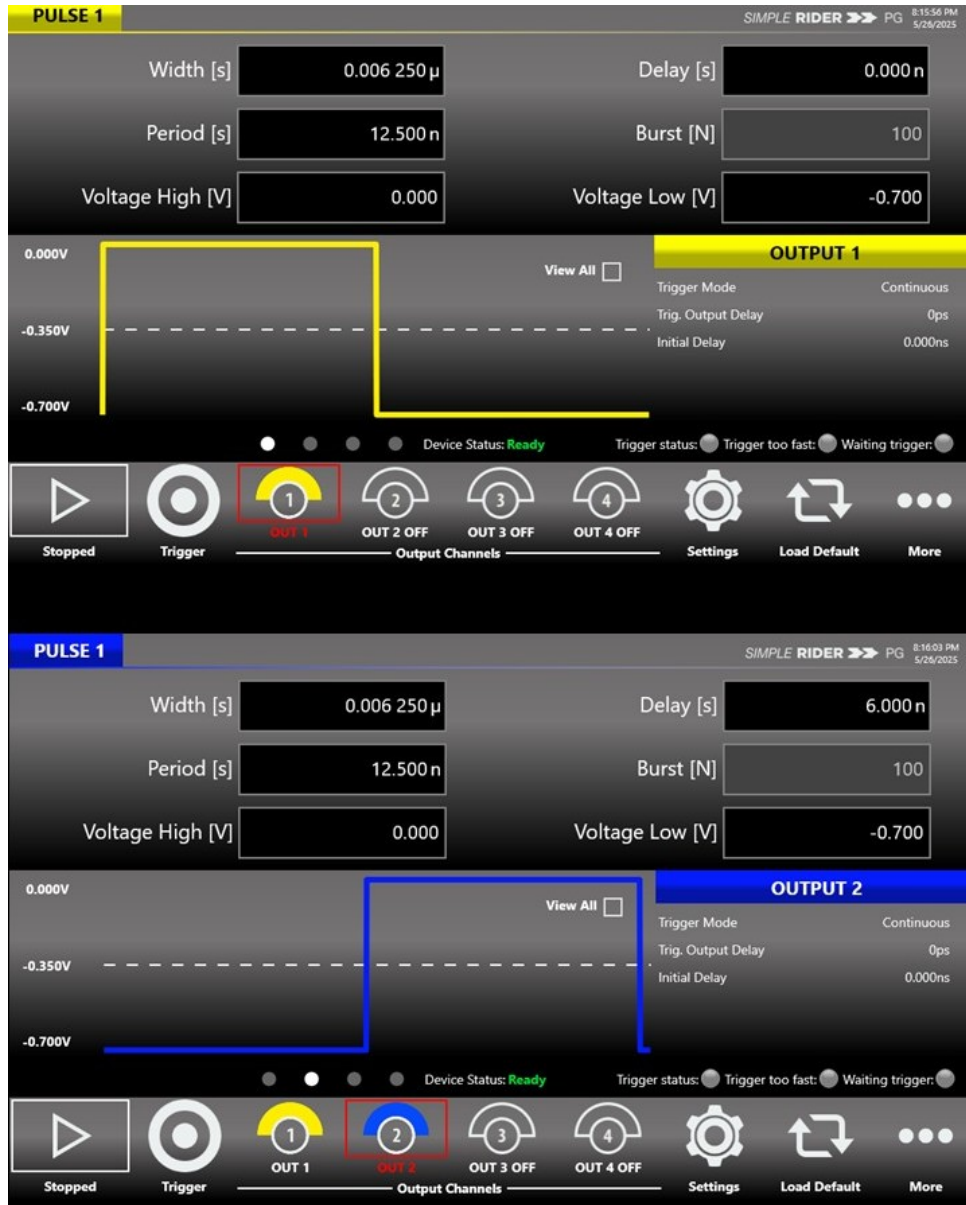


Figure 3.2: START and STOP configuration.

Once the FLASH module is connected, the next step involves configuring the signal generation system through the impulsator Pulse Rider PG-1074 from Active Technologies. The aim is to produce a START pulse and a STOP pulse, where the second is delayed

relative to the first in order to simulate photon arrivals, as shown in Figure 3.1. In this inverted START-STOP configuration, the STOP signal scans the time axis while the START signal represents a photon detection event. This means that in each time window between two STOP pulses, a photon event will occur approximately at the same temporal position (or bin) since the two signals are correlated, and so, the resulting histogram will not exhibit a classical fluorescence decay curve, rather a Gaussian-like distribution centered around the simulated photon arrival time. In this setup, the START and STOP signals are generated at 80 MHz, since the goal is to test the maximum performance of the time-tagging method.

These signals are initially verified using the MDO34 3-BW-200 oscilloscope to ensure their correctness in terms of shape, timing, and amplitude as shown in Figure 3.3. The START-STOP timing interval is determined by measuring the time difference between the falling edges of the two signals.

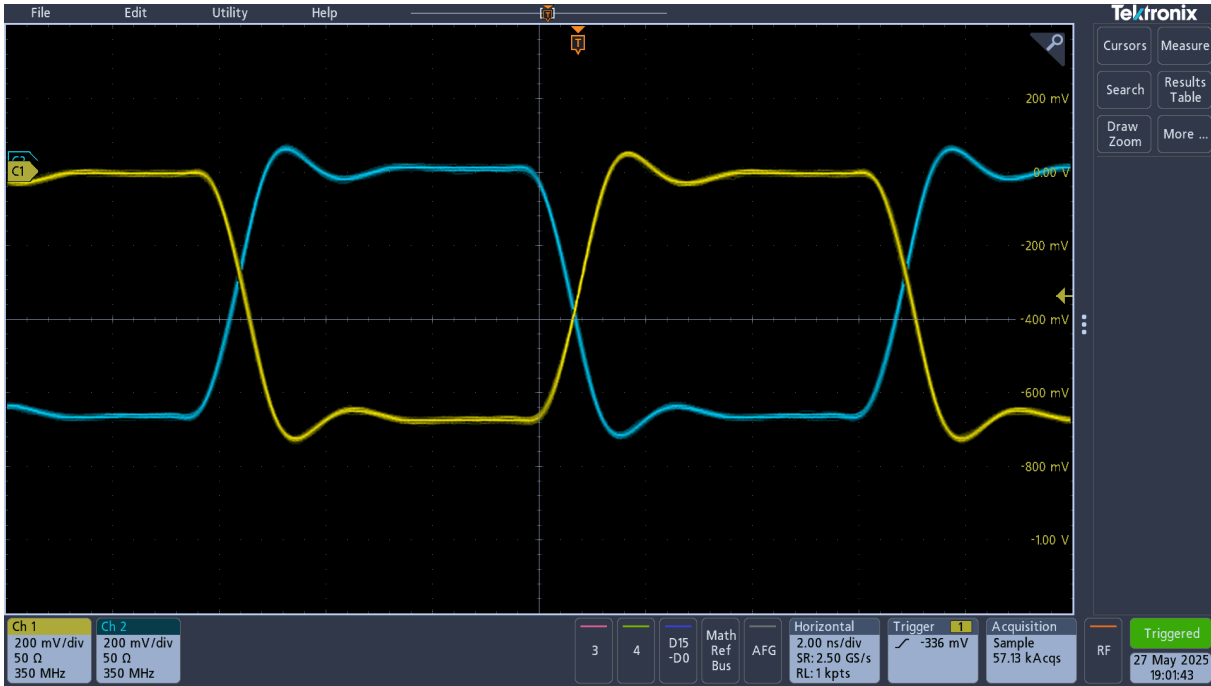


Figure 3.3: Oscilloscope screen representing START (yellow) and STOP (blue) signals.

After validation, the START signal is routed to the F-TAC input of the FLASH module, while the STOP signal is connected to the corresponding STOP input. With the hardware fully configured and the signals properly delivered to the module, the system is ready to initiate the measurement phase.

3.2. Software Verification Procedure

This section outlines the methodology adopted for the verification and validation of the data acquisition system, detailing the configuration procedures, the multi-tool software testing workflow, and the subsequent data postprocessing. The validation workflow typically begins by launching four main development environments: Vivado 2019.2, Visual Studio 2022, MATLAB R2021b, and NI LabVIEW 2021. Upon powering the setup up, the FPGA automatically reprograms itself with the default bitstream stored in the flash memory, which corresponds to the histogram acquisition mode.

The testing phase begins with the verification of the signals previously configured using both the pulse generator and the oscilloscope, in combination with the LabVIEW software developed by the researcher Serena Farina to check if the PC is receiving data correctly. The process starts by selecting the appropriate device within the LabVIEW interface and, once the device is selected, the connection is established and the measurement parameters are configured according to the required specifications. After completing the setup, the signal acquisition is initiated by pressing the START button and the selected channels are dynamically displayed.

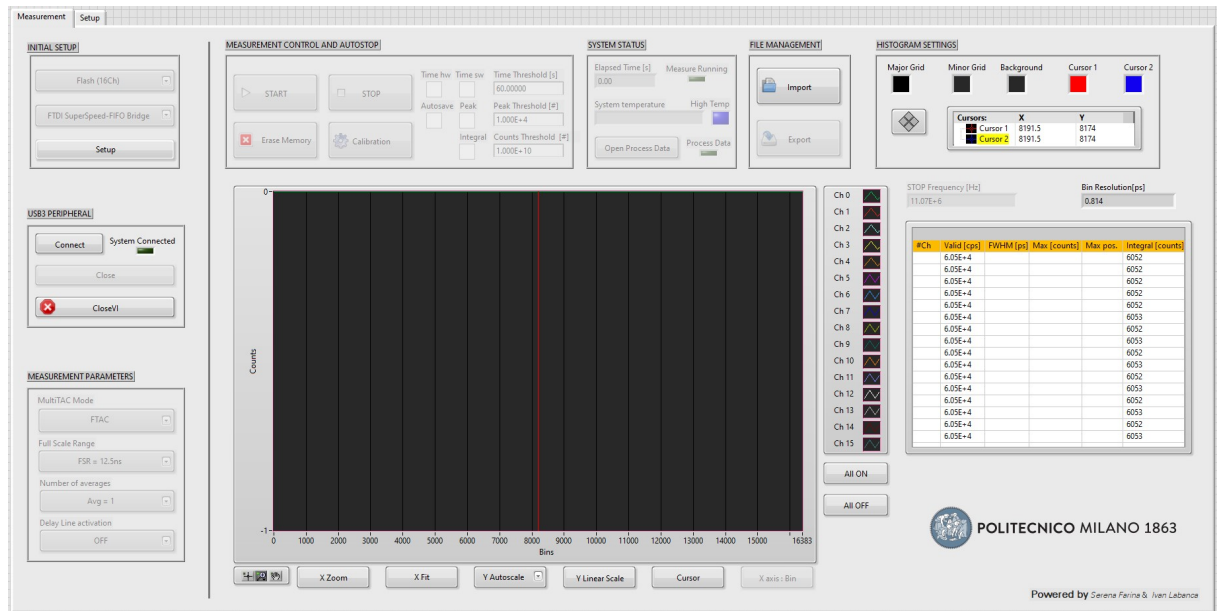


Figure 3.4: Serena Farina's verification software [12].

Then, the process continues switching to the C# application developed in Visual Studio and opening the histogram mode Form2 GUI to test all sixteen channels. This process starts with a device scan, followed by the establishment of a connection clicking the chosen device from the displayed ones in the drop-down list, and configuring the TAC Full Scale Range parameter. Once the configuration is complete, data acquisition begins by pressing the START button and the generated charts can then be observed in real-time, with the possibility to zoom in and out to highlight the behavior of each channel.

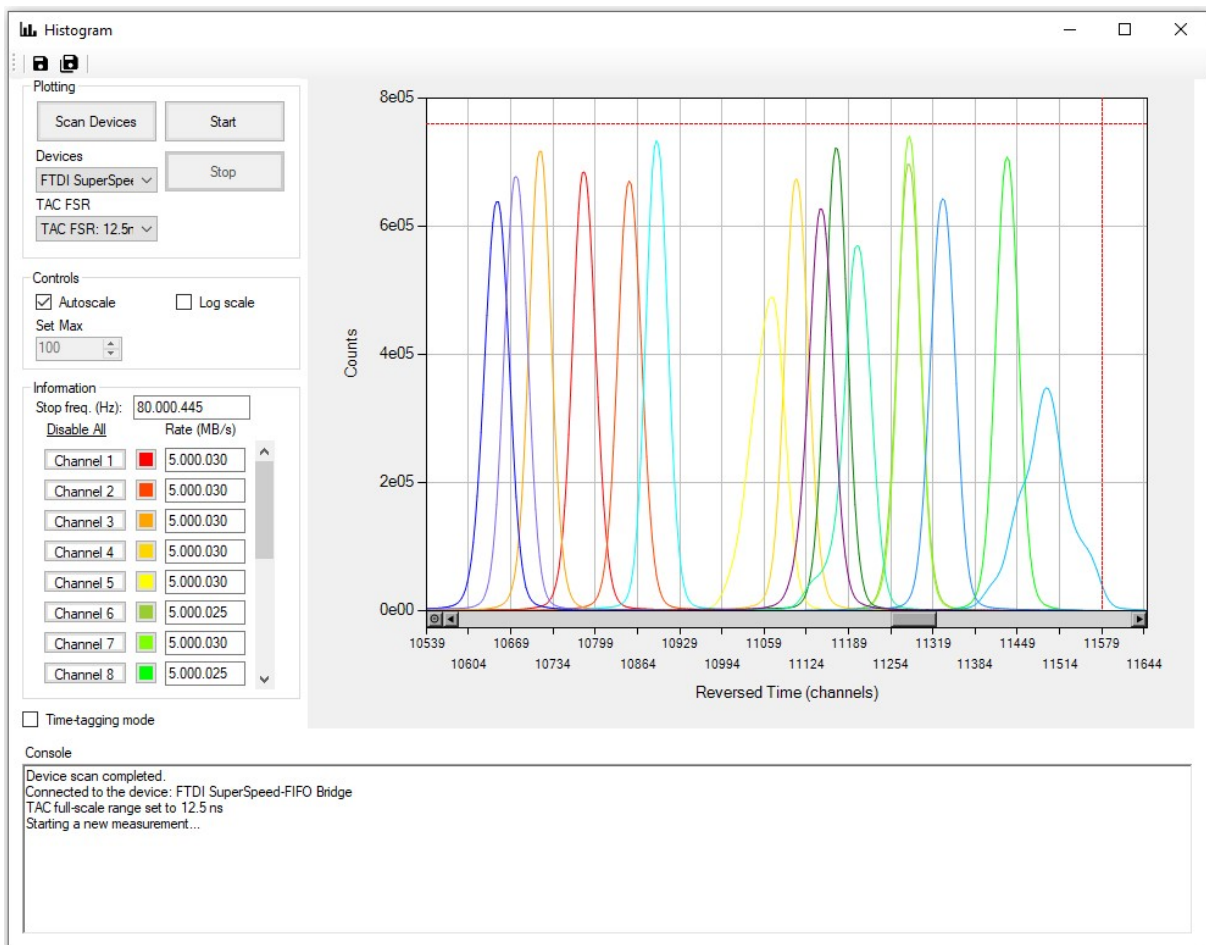


Figure 3.5: Histogram mode on.

As presented in Figure 3.2, the signal under analysis has a stop frequency of 80 MHz, which is distributed across 16 channels, resulting in an effective frequency of 5 MHz per channel. The START signal, generated by the same pulse generator, is also characterized by a frequency of 80 MHz and is temporally correlated with the stop signal, with the stop pulse delayed by approximately 6 ns. The resulting histogram displays a Gaussian distribution for each channel. This characteristic shape is due to the fixed time delay between the correlated start and stop signals, combined with the temporal jitter and

finite timing resolution of the system. This temporal noise introduces small variations in the measured time intervals, causing the data to spread around the true delay value and forming the Gaussian peak observed in the measurement.

Stated that, the tester can proceed to STOP and SAVE DATA from all or selected channels. At this point, it is expected to see a characteristic distribution of data in the output file: this file includes 16384 rows corresponding to each bin, matching the resolution of the histogram. The file initially contains a sequence of zeros, followed by a gradual increase in values up to a peak, after which the values decrease again. This distribution confirms the proper functionality of the histogram mode and the expected behavior of the input testing signals. Additional tests are conducted by enabling the logarithmic scale and adjusting minimum and maximum bounds to analyze the dynamic range.

After validating the histogram mode, the application is closed and the FPGA is reprogrammed for time-tagging operation using Vivado. This procedure involves launching Vivado, accessing the Hardware Manager, selecting the target device, and loading the time-tagging bitstream. During programming, the red LED on the FPGA board remains illuminated to indicate that the process is in progress, and it switches off automatically upon successful completion.

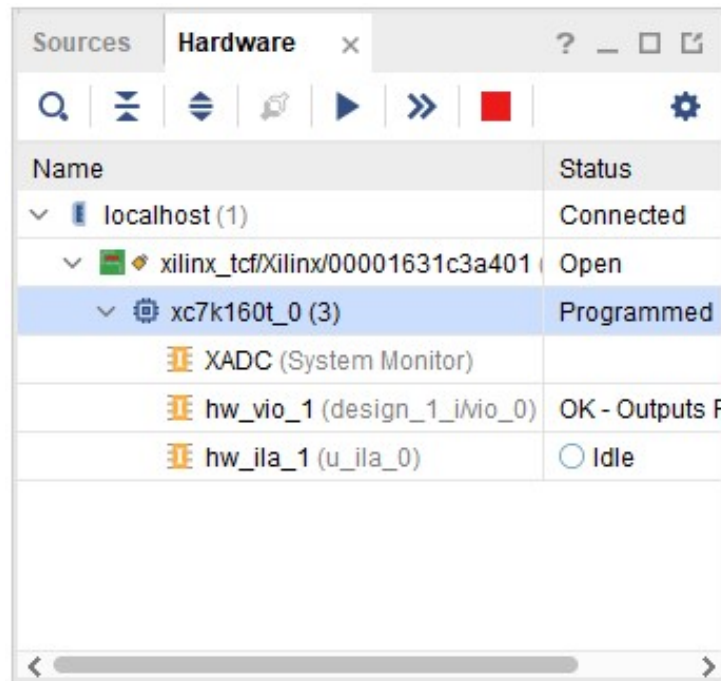


Figure 3.6: Vivado interface [33].

Once reprogrammed, the C# application is reopened and, after doing the same `Form2` configuration routine, `Form1` is launched to handle the time-tagging interface. As shown in Figure 3.3, two types of acquisition configurations are supported: manual (user-defined) and automatic (based on a precompiled configuration file) and in both cases, the system confirms the device status and readiness within the interface.

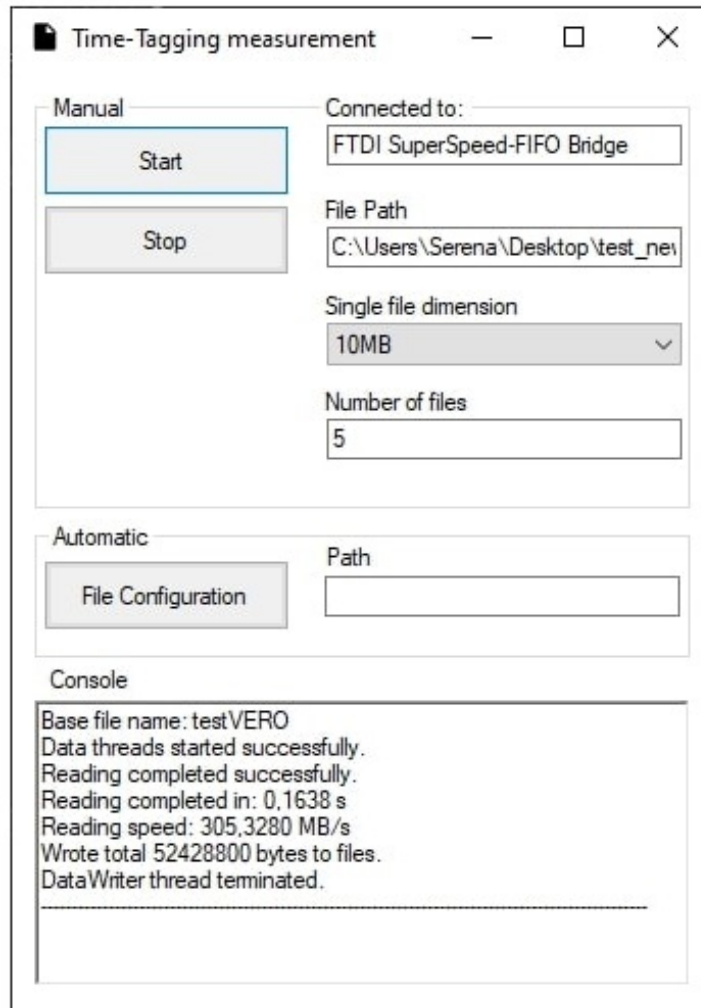


Figure 3.7: Time-Tagging mode on.

For **manual configuration**, the user must specify the *base name* of the files where data will be written and the *output directory* within a Save File dialog opened with a click, then I have to select the *file size* (1 MB, 10 MB, or 100 MB), and the *number of files* to be generated. Upon pressing the START button, the data transfer begins, and the console provides real-time feedback for debugging and monitoring. Note that, after the stoppage or the end of the measurement, if additional acquisitions are performed without changing the base filename, the system automatically continues the sequence by incrementing file indices (e.g., from files 1–5 to files 6–10), ensuring no overwriting. In contrast, using a

different filename restarts the numbering from 1, again without affecting previously saved data.

The **file configuration** feature was introduced to replicate the manual measurement setup process in an automated way when the presence of an operator is not feasible, but it has to be tested with an interface button (which will be enabled by an external control software in the real world experiments). The file is built as shown in section 2.4.2.1 and is then selected pressing the the FILE CONFIGURATION button. Once selected and confirmed the right .txt file, the measurement starts as if it was done with manual configuration.

After the acquisition, the generated files are analyzed using a MATLAB script which plots the data and verifies consistency with the expected photon arrival patterns. It also flags any anomalies, such as incomplete data blocks or missing FIFO markers, which indicates data loss.

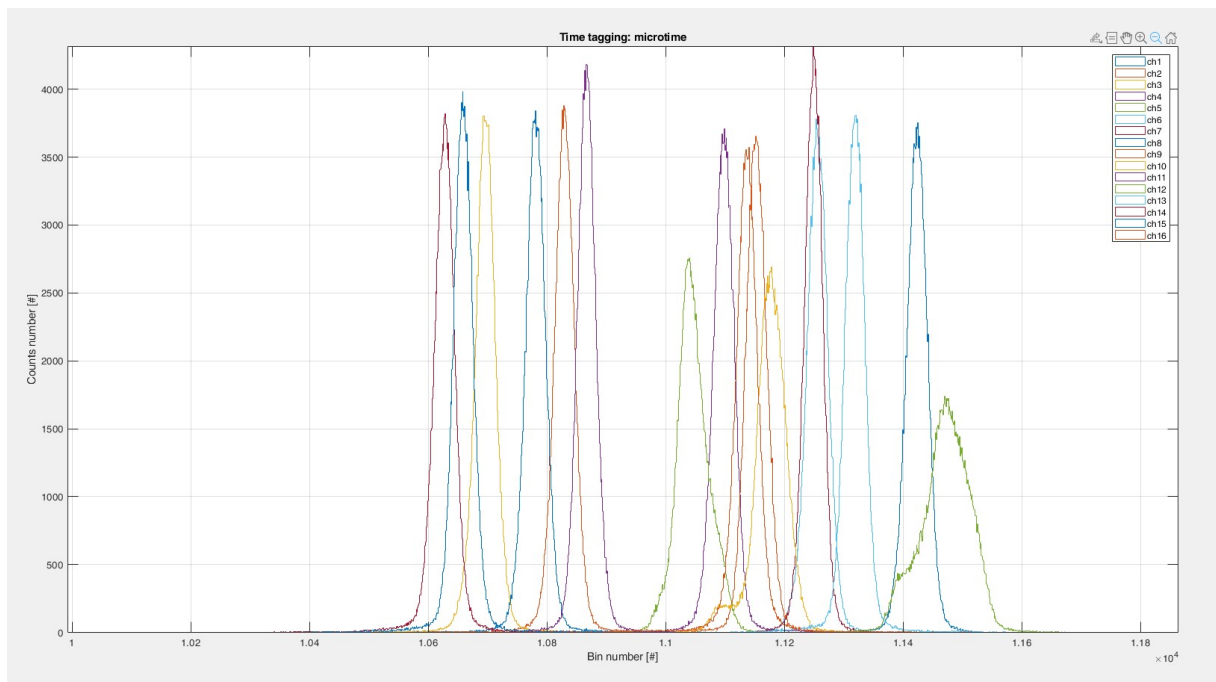


Figure 3.8: MATLAB reconstruction.

By comparing Figure 3.2 with Figure 3.5, it is evident that the resulting histograms are consistent. This correspondence arises from the fact that, in both acquisition modes, the photon events originate from the same data source and micro-time measurements. In histogram mode, the FPGA directly processes photon arrival times into binned counts, aggregating data in real-time to build and transmit complete histograms for each channel. The host computer simply retrieves these precomputed histograms with minimal band-

width requirements. Conversely, in time-tagging mode, the FPGA streams raw photon event data at high rates, encoding each event in a compact 32-bit word that includes micro-time, which is fundamental to histogram reconstruction through MatLab. Despite the differences in architecture, with the FPGA handling aggregation in histogram mode and the host performing it in time-tagging mode, the final histograms derived from both approaches align, as they are ultimately built from the same fundamental photon event information.

3.3. Prospects for Evolution and Performance Improvements

The developed system successfully achieves a data transmission speed of 2.56 Gbps in time-tagging mode, fully exploiting the capabilities of the USB 3.0 Superspeed interface. Moreover, all functionalities, including both time-tagging and histogram modes, have been integrated within a single, modular software solution based on object-oriented principles. This architecture lays the foundation for a flexible and scalable tool, which can easily adapt to future improvements in both hardware and data processing methodologies.

Looking ahead, one of the most promising directions for the evolution of this system is the integration of the FPGA multiboot capability. This feature would allow the FPGA to store multiple bitstreams in its non-volatile flash memory and dynamically switch between them upon receiving specific commands from the C# software. In practice, this would mean that the operator could change the operational mode of the FPGA (between histogram and time-tagging configurations) without needing to physically reload the bitstream via external tools such as Vivado. Instead, the switching could be managed entirely through the software interface, with the inclusion of features like a loading bar to inform the user about the progress of the reconfiguration process. Such an enhancement would not only improve usability but also reduce downtime between mode switches, increasing the efficiency of experimental workflows. The adoption of multiboot strategies in FPGA-based instrumentation has already shown potential in similar contexts, as reported in studies focused on reconfigurable systems for space and imaging applications [23].

In a MultiBoot configuration, the FPGA is capable of dynamically reloading its configuration by selecting and loading a different bitstream stored in its SPI flash memory. Typically, at power-up, the FPGA loads a golden image, a pre-loaded bitstream, stored at the base address. From this initial configuration, the system can trigger a switch to a MultiBoot image located at a higher memory address, corresponding to an alternative

design. When a MultiBoot event is triggered, the FPGA clears its configuration memory while preserving specific control registers and then restarts the configuration process, fetching the new bitstream from the designated flash memory address. In the event that the MultiBoot image fails to configure correctly, the fallback mechanism ensures that the golden image is automatically reloaded, providing system robustness against configuration errors. This architecture allows multiple MultiBoot images to coexist, with the possibility for any configured design to trigger the loading of another image as needed [23].

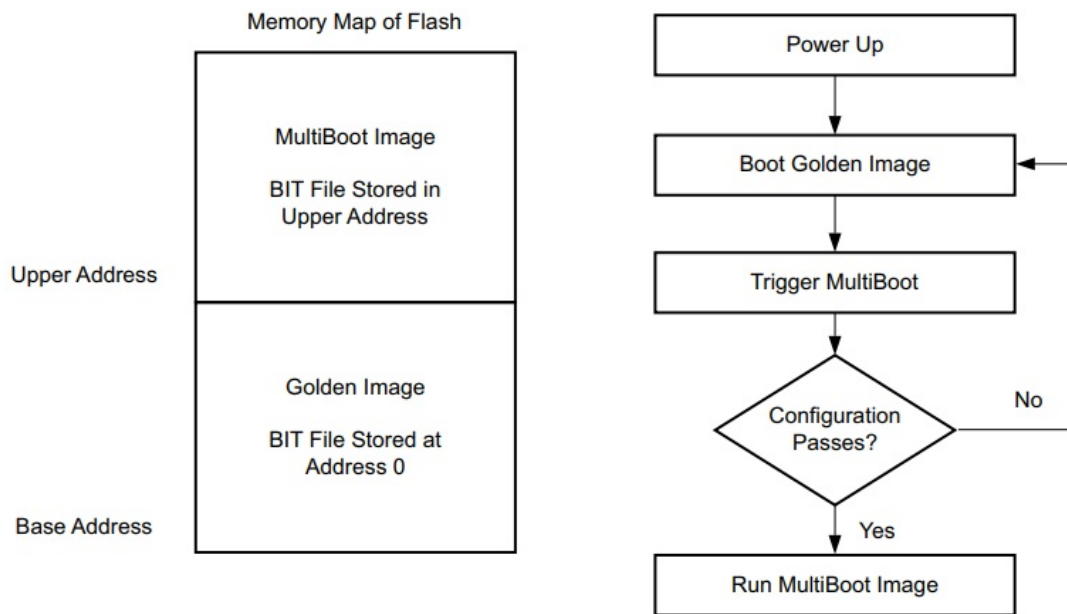


Figure 3.9: MultiBoot flow diagram.

Another promising development could be the integration of the MATLAB post-processing routines directly within the C# application, enabling a unified software environment for both data acquisition and analysis. This would consolidate multiple functionalities within a single tool (as was successfully achieved in this work with the implementation of the histogram mode) thereby enhancing usability and streamlining the user workflow.

Then, the modular structure of the software and the choice of a modern object-oriented design open the door to further enhancements. These could include the addition of new data analysis modules for online event filtering or preliminary statistical processing, or the implementation of advanced visualization tools capable of rendering photon arrival distributions in real time. Furthermore, the scalable nature of the architecture ensures that the system can easily be adapted to future hardware upgrades, such as the transition to faster interfaces like USB 3.1 or PCI Express, or the integration of more sophisticated FPGA devices with built-in data compression or signal processing capabilities.

In conclusion, the work presented in this thesis not only achieves its initial objectives, but also establishes a solid foundation for the continuous evolution of both software and hardware components. The future developments outlined here are not merely speculative; they are well-aligned with current trends in the field and supported by the results reported in recent literature. By pursuing these directions, it will be possible to further increase the efficiency, flexibility, and scientific value of photon-counting acquisition systems in advanced sensing and imaging applications.

Bibliography

- [1] J. Albahari. Threading in c. *URL:< http://www. albahari. com/threading*, 2006.
- [2] J. Albahari. *C# 12 in a Nutshell: The Definitive Reference*. " O'Reilly Media, Inc.", 2023.
- [3] J. Albahari and B. Albahari. *in a nutshell*. 2003.
- [4] W. Becker. *Advanced Time-Correlated Single Photon Counting Techniques*. Springer, 2005.
- [5] W. Becker. *The bh TCSPC handbook*. Becker & Hickl GmbH, 2021.
- [6] Y. Bérubé-Lauzière, M. Crotti, S. Boucher, S. Ettehad, J. Pichette, and I. Rech. Prospects on time-domain diffuse optical tomography based on time-correlated single photon counting for small animal imaging. *Journal of Spectroscopy* <https://onlinelibrary.wiley.com/doi/full/10.1155/2016/1947613>, 2016(1): 1947613, 2016.
- [7] F. Bouchard, A. Sit, Y. Zhang, R. Fickler, F. M. Miatto, Y. Yao, F. Sciarrino, and E. Karimi. Two-photon interference: the hong-ou-mandel effect. *Reports on Progress in Physics*, 84(1):012402, 2020.
- [8] F. Bouchard, A. Sit, Y. Zhang, R. Fickler, F. M. Miatto, Y. Yao, F. Sciarrino, and E. Karimi. Two-photon interference: the hong-ou-mandel effect. *Reports on Progress in Physics*, 84(1):012402, 2020.
- [9] F. Ceccarelli, G. Acconcia, A. Gulinatti, M. Ghioni, I. Rech, and R. Osellame. Recent advances and future perspectives of single-photon avalanche diodes for quantum photonics applications. *Advanced Quantum Technologies*, 4(2):2000102, 2021.
- [10] S. Cova, M. Ghioni, A. Lacaita, C. Samori, and F. Zappa. Avalanche photodiodes and quenching circuits for single-photon detection. *Applied optics*, 35(12):1956–1976, 1996.
- [11] G. Di Martino, Y. Sonnefraud, M. S. Tame, S. Kéna-Cohen, F. Dieleman, Ş. Özdemir,

- M. Kim, and S. A. Maier. Observation of quantum interference in the plasmonic hong-ou-mandel effect. *Physical Review Applied*, 1(3):034004, 2014.
- [12] S. Farina. Oltre i limiti del pile-up : realizzazione e test di un sistema per tcspc ad elevati rate e bassa distorsione. *URL* <https://www.politesi.polimi.it/handle/10589/153909>, 2018-2019.
- [13] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, 2012.
- [14] A. Gallivanoni, I. Rech, and M. Ghioni. Progress in quenching circuits for single photon avalanche diodes. *IEEE Transactions on nuclear science*, 57(6):3815–3826, 2010.
- [15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns. Elements of reusable object-oriented software. *Design Patterns*, 1995.
- [16] B. Goetz. *Java concurrency in practice*. Pearson Education, 2006.
- [17] HTML.it. Guida alla programmazione orientata agli oggetti, s.d. *URL* <https://www.html.it/guide/guida-programmazione-orientata-agli-oggetti/>.
- [18] E. Instruments. Tcspc-what is time-correlated single photon counting. *URL* <https://www.edinst.com/resource/tcspc-what-is-time-correlated-single-photon-counting/>, 2013.
- [19] E. Instruments. Determining fluorescence lifetimes. *URL* <https://www.edinst.com/resource/determining-fluorescence-lifetimes/>, 2023.
- [20] H. C. Ishikawa-Ankerhold, R. Ankerhold, and G. P. Drummen. Advanced fluorescence microscopy techniques—frap, flip, flap, fret and flim. *URL* <https://pubmed.ncbi.nlm.nih.gov/22469598/>, 17(4):4047–4132, 2012.
- [21] M. Jachura. Coalescing photon pairs caught on camera.
- [22] A. Kalyanov, J. Jiang, S. Lindner, L. Ahnen, A. di Costanzo, J. M. Pavia, S. S. Majos, and M. Wolf. Time domain near-infrared optical tomography with time-of-flight spad camera: The new generation. In *Optical Tomography and Spectroscopy*, pages OF4D–5. Optica Publishing Group, 2018.
- [23] G. Kempahommaiah. Multiboot and fallback using icap in ultrascale+ fpgas application note (xapp1296). Technical report, Xilinx, 2017.
- [24] J. R. Lakowicz. *Principles of fluorescence spectroscopy*. Springer, 2006.

- [25] F. Ltd. An386 ftdi ft600 - maximize performance. Technical report, Future Technology Devices International, 2016. Application Note AN_386.
- [26] F. Ltd. An407 d3xx .net programmer’s guide. Technical report, Future Technology Devices International, 2016. Application Note AN_407.
- [27] F. Ltd. An412 ft600/ft601 usb bridge chips integration. Technical report, Future Technology Devices International, 2017. Application Note AN_412.
- [28] F. Ltd. Ft600q/ft601q usb 3.0 superspeed ic datasheet. Technical report, Future Technology Devices International, 2021. Datasheet DS_FT600Q/FT601Q.
- [29] B. Meyer. *Object-oriented software construction*, volume 2. Prentice hall Englewood Cliffs, 1997.
- [30] Y. Mu and M. Niedre. Fast single photon avalanche photodiode-based time-resolved diffuse optical tomography scanner. *Biomedical optics express*, 6(9):3596–3609, 2015.
- [31] Oracle. Java tutorials. URL <https://docs.oracle.com/javase/tutorial/java/concepts/>.
- [32] D. Piao, K. E. Bartels, Z. Jiang, G. R. Holyoak, J. W. Ritchey, G. Xu, C. F. Bunting, and G. Slobodov. Alternative transrectal prostate imaging: a diffuse optical tomography method. *IEEE Journal of Selected Topics in Quantum Electronics*, 16(4):715–729, 2009.
- [33] G. F. Piergiorgio Daniele. Progettazione e caratterizzazione di un sistema multicanale ad alte prestazioni per applicazioni tcspc: il modulo asi. 2022-2023.
- [34] V. Shcheslavskiy, M. Shirmanova, K. Yashin, A. Rück, M. Skala, and W. Becker. Fluorescence lifetime imaging techniques—a review on principles, applications and clinical relevance. *Journal of Biophotonics*, page e202400450, 2025.
- [35] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, 1986.
- [36] A. Tanenbaum. *Modern operating systems*. Pearson Education, Inc., 2009.
- [37] M. Wahl. Time-correlated single photon counting. URL: https://www.picoquant.com/images/uploads/page/files/7253/technote_tcspc.pdf, 2014.
- [38] M. Wahl, T. Röhlicke, S. Kulisch, S. Rohilla, B. Krämer, and A. C. Hocke. Photon arrival time tagging with many channels, sub-nanosecond deadtime, very high

- throughput, and fiber optic remote synchronization. *Review of Scientific Instruments*, 91(1), 2020.
- [39] J. Wang, S. Jiang, Z. Li, R. M. diFlorio Alexander, R. J. Barth, P. A. Kaufman, B. W. Pogue, and K. D. Paulsen. In vivo quantitative imaging of normal and cancerous breast tissue using broadband diffuse optical tomography. *Medical physics*, 37(7Part1):3715–3724, 2010.
- [40] Xilinx. Kintex-7 fpga family: Product table and product selection guide, Mar. 2021. URL https://docs.xilinx.com/v/u/en-US/ds182_Kintex_7_Data_Sheet.
- [41] F. Zappa, S. Tisa, A. Tosi, and S. Cova. Principles and features of single-photon avalanche diode arrays. *Sensors and Actuators A: Physical*, 140(1):103–112, 2007.

List of Figures

1.1	Measurement of start-stop times in time-resolved fluorescence measurement with TCSPC. [37].	5
1.2	Example of an histogram construction [33].	6
1.3	Example of a TCSPC conventional setup [33].	8
1.4	Architecture of a classic TCSPC device [5].	10
1.5	Simplified schema of a SPAD [14].	10
1.6	Operation principle of a TAC level (red line)[37].	11
1.7	Reversed START-STOP configuration [5].	12
1.8	Processes involved in the excitation and emission of fluorescence [34]. . . .	13
1.9	Representation of a fluorescence decay following a short excitation pulse on a (a) linear and (b) logarithmic scale [19].	14
1.10	Schematic diagram of a SPAD-based DOT imager[30].	15
1.11	Schematic representation of a TOF system using the TCSPC technique [33].	16
1.12	(a) Beam splitter with input ports a, b and output ports c, d. (b) Two photons can exit through the same port (top) or different ports (bottom) [7].	17
1.13	Schematic of ASI TCSPC architecture.	18
1.14	Acquisition boards Block Diagram [33].	21
1.15	Microcontroller schematic [33].	22
1.16	Histogram of start-stop times in time-resolved fluorescence measurement with TCSPC [37].	29
1.17	Time-Tag Data illustration [33].	30
2.1	Objects interacting through encapsulation.	38
2.2	WorkerThread is the superclass, the other are the subclasses.	38
2.3	Abstraction example: Abstract Thread Class acts as a blueprint.	41
2.4	Parallelization principle [1].	45
2.5	Implementation of the Producer-Consumer pattern.	46
2.6		51
2.7	UML diagram of the application.	52
2.8	Form1 GUI.	54

2.9	Example of .txt file for automatic mode.	57
2.10	Form2 GUI.	59
2.11	AsyncTask method for DataReaderTimeTag	67
2.12	AsyncTask method for DataReaderHisto	68
2.13	AsyncTask method for DataWriterTimeTag	69
2.14	Insertion of the histoData of a specific channel in the internal structure . 73	
3.1	Complete hardware testing setup.	78
3.2	START and STOP configuration.	79
3.3	Oscilloscope screen representing START (yellow) and STOP (blue) signals. 80	
3.4	Serena Farina's verification software [12].	81
3.5	Histogram mode on.	82
3.6	Vivado interface [33].	83
3.7	Time-Tagging mode on.	84
3.8	MATLAB reconstruction.	85
3.9	MultiBoot flow diagram.	87

List of Tables

2.1	Command reference for <code>sendToFPGA</code>	65
-----	---	----

