Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Software y Sistemas

Trabajo Fin de Máster

C++ Implementation of IRIO Software Tools for the Management of FPGA Reconfigurable Input/Output Devices

Autor: Víctor Costa Pérez
Tutor: Jose María Barambones Ramírez
Co-Tutor: Mariano Ruiz Gonzalez

Madrid, Julio 2024

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster
Máster Universitario en Software y Sistemas

Título: C++ Implementation of IRIO Software Tools for the Management of FPGA Reconfigurable Input/Output Devices

Julio 2024

Autor: Víctor Costa Pérez
Tutor: Jose María Barambones Ramírez
Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid
Co-Tutor: Mariano Ruiz Gonzalez
Ingeniería Telemática y Electrónica
ETS de Ingeniería y Sistemas de Telecomunicación
Universidad Politécnica de Madrid

Resumen

Los requisitos en experimentos de Big Science suelen demandar hardware específico y muy especializado, capaz de adquirir o generar datos a altas velocidades, además de muchas veces requerir operar a tiempo real. Una solución a estos problemas son las FPGAs, que permiten implementar aplicaciones muy especializadas con recursos lógicos, generando hardware específico para cada aplicación.

La familia de dispositivos RIO ofrece FPGAs con capacidades de entrada y salida en una gran variedad de factores de forma, lo que las convierte en una opción muy versátil. Otra de sus grandes ventajas es que pueden ser configuradas a través de LabVIEW FPGA, simplificando el desarrollo. Sin embargo, para interactuar con estas aplicaciones desde un host sigue siendo necesario conocer en profundidad la aplicación, ya que puede existir un número prácticamente infinito de diferentes implementaciones.

Para simplificar este problema, en 2015 se propusieron las IRIO Design Rules, un conjunto de normas a la hora de nombrar recursos y su división según su funcionalidad. Al establecer un conjunto de normas es posible crear uan API más simplificada para interactuar con el host. Este es el caso de irioCore, una libreria en C lanzada junto con las reglas de diseño. Sin embargo, se identificaron posibles problemas relacionados con la gestión de memoria y la seguridad. Además, su DevOps es muy básico, ofreciendo únicamente pruebas funcionales, las cuales requieren los dispositivos RIO, además de otro hardware adicional.

Para abordar estos problemas, se ha decidido reimplementar la librería en C++, aprovechando el paradigma OOP y otras características avanzadas propias de un lenguaje más moderno. También se añadirá soporte para la R Series, anteriormente no soportado. También se implementará un mecanismo de retrocompatibilidad en la forma de un C wrapper, para permitir que aplciaciones ya desarrolladas sigan funcionando con las nuevas ventajas.

Para mejorar la calidad del software y su desarrollo, se implementarán varias estrategias de DevOps, como la definición de diferentes pipelines, procesos para comprobar la calidad del código, así como la simplificación de futuras operaciones de mantenibilidad y expansión. Estas estrategias incluirán procesos CI/CD y pruebas automatizadas. Además, se utilizarán herramientas de análisis estático del código para obtener medidas detalladas de calidad del código. Con estos resultados, se comparará la biblioteca anterior con la nueva, buscando cuantificar las mejoras en términos de eficiencia, seguridad y mantenibilidad. De este modo, se garantizará que la nueva biblioteca no solo cumpla con los requisitos funcionales, sino que también ofrezca una mejora de calidad y facilite el trabajo al desarrollador.

Abstract

Big Science experiments typically require highly specialized hardware capable of acquiring or generating data at high speeds and often need to operate in real-time. One solution to these challenges is FPGAs, which allow for the implementation of highly specialized applications with logical resources, creating hardware specific to each application.

The RIO family of devices offers FPGAs with input and output capabilities in a wide variety of form factors, making them a very versatile option. Another significant advantage is that they can be configured through LabVIEW FPGA, simplifying development. However, to interact with these applications from a host, it is still necessary to have an in-depth understanding of the application, as there can be virtually infinite different implementations.

To simplify this problem, the IRIO Design Rules were proposed in 2015, a set of norms for naming resources and dividing them according to their functionality. By establishing a set of rules, it is possible to create a more simplified API to interact with the host. This is the case of irioCore, a C library launched along with the design rules. However, potential issues related to memory management and security were identified. Moreover, its DevOps is very basic, offering only functional tests, requiring the actual RIO devices on top of additional hardware.

To address these issues, the decision has been made to reimplement the library in C++, taking advantage of the OOP paradigm and other advanced features of a more modern language. Support for the R Series, previously unsupported, will also be added. A mechanism for backward compatibility in the form of a C wrapper will be implemented to allow already developed applications to continue functioning with the new advantages.

To improve software quality and development, several DevOps strategies will be implemented, such as defining different pipelines and processes to check code quality, as well as simplifying future maintainability and expansion operations. These strategies will include CI/CD processes and automated tests. Additionally, static code analysis tools will be used to obtain detailed code quality metrics. With these results, the previous library will be compared with the new one, seeking to quantify improvements in terms of efficiency, security, and maintainability. In this way, it will be ensured that the new library not only meets functional requirements but also offers quality improvements and facilitates the developer's work.

Table of contents

Acronyms

ABI Application Binary Interface. 24, 51, 96

ADC Analog to Digital Converter. 31

ALM Adaptive Logic Module. 16

API Application Programming Interface. i, iii, 2, 24, 51, 96

ASIC Application Specific Integrated Circuit. 1, 16

BFP BitFile Parser. 26, 28, 30, 31, 48, 65–67, 70–72

CD Continuous Delivery. 7

CI Continuous Integration. 7, 13, 15, 56, 59, 72

CI/CD Continuous Integration and Continuous Delivery. i, iii, 2, 3, 6, 7, 14, 16, 25, 60, 61, 84, 86, 96, 99

CL CameraLink. 45, 46, 53–56

CLI Command-Line Interface. 13

COTS Commercial Off-The-Shelf. 1

CPU Central Processing Unit. 17

cRIO Compact RIO. 19, 20, 22, 26, 31, 42, 83, 88

CVCS Centralized Version Control System. 11

DAC Digital to Analog Converter. 31

DAQ Data Acquisition. 22, 27, 46, 55

DevOps Development Operations. i, iii, 3, 5, 6, 12, 14, 59, 60, 96

DMA Direct Access Memory. 21, 22, 28–31, 42–45, 53, 57

DSP Digital Signal Processing. 16, 28

DVCS Distributed Version Control System. 11, 14, 61

fff Fake Function Framework. 13, 72

FIFO First In, First Out. 43

FlexRIO Flexible RIO. 20, 22, 26, 31, 41, 55, 56

FPGA Field Programmable Gate Array. i, iii, 1–3, 16–22, 26–29, 31–33, 35–43, 48, 49, 51, 53, 57, 58, 72, 74–78, 95, 99

HDL Hardware Description Language. 1, 16

HtT Host to Target. 28

I/O Input/Output. 1, 18–20, 22, 28, 37, 39

I2A2 Investigación en Instrumentación y Acústica Aplicada. 2, 25

IC Integrated Circuits. 1

IDE Integrated Development Environment. 10, 54

IMAQ Image Acquisition. 22, 27, 45, 46, 53, 55

LabVIEW Laboratory Virtual Instrument Engineering Workbench. i, iii, 1, 19, 21, 25, 28, 31, 36, 95

LOC Lines Of Code. 83, 90, 91, 96

LUT Lookup Table. 28

MFP Master's Final Project. 3

MUSS Master Universitario en Software y Sistemas. 3

MXI Multi-System Extension Interface. 18, 19, 55

NI National Instruments. 1, 18, 22, 48, 55, 56, 99

OOP Object-Oriented Programming. i, iii, 22–24, 26, 90, 96

OS Operating System. 15, 60

PBP Point By Point. 22, 27, 40, 46

PCI Peripheral Component Interconnect. 18, 20, 99

PCIe Peripheral Component Interconnect Express. 18, 20, 56

PLC Programmable Logic Controller. 19

PXI PCI eXtensions for Instrumentation. 18, 20, 99

PXIe PCI eXtensions for Instrumentation Express. 18, 20, 55

RAII Resource Acquisition Is Initialization. 23, 26, 95

RAM Random Accesss Memory. 16, 17

RIO Reconfigurable Input/Ouput. i, iii, 1–3, 18, 19, 21, 22, 25, 26, 31, 48, 49, 51, 53, 56, 57, 68, 72–74, 76–78, 88, 95, 96, 99

SoC System-on-Chip. 17

STL Standard Template Library. 22

TDD Test-Driven Development. 7

TtH Target to Host. 28

UART Universal Asynchronous Receiver-Transmitter. 45, 46, 53

USB Universal Serial Bus. 19

VCS Version Control Systems. 11, 13

WSL Window Subsystem for Linux. 54

XML Extensible Markup Language. 21, 28–30, 66, 71

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context

Big Science experiments, such as those conducted in particle physics, fusion, and other cutting-edge scientific fields, often demand specialized hardware capable of handling vast amounts of data at high speeds, usually with real-time constraints. These requirements necessitate robust and adaptable solutions, which are only sometimes available as Commercial Off-The-Shelf (COTS) components.

One solution is using Application Specific Integrated Circuits (ASICs), which are Integrated Circuitss (ICs) designed for a specific operation. However, ASICs are costly and time-consuming to develop, requiring extensive design and manufacturing processes that are not easily adaptable to the experimental environment, which may require rapid changes and iterations, as, once fabricated, they must be redesigned and fabricated again, which can cause several delays and stop fast and iterative innovation.

Field Programmable Gate Arrays (FPGAs) offer a solution to this problem while also meeting the demands of performance and timing. FPGAs can be reprogrammed as needed, offering the capability to prototype hardware configurations rapidly. This flexibility makes FPGAs particularly well-suited for the demanding nature of Big Science experiments. It is important to note that FPGAs implement the specific application in hardware, using logical elements and memory blocks, which makes it possible to implement any function that could be implemented using ASICs with the same performance.

However, developing applications for FPGAs offer several challenges. One of the main ones is the difficulty of creating the applications, requiring Hardware Description Language (HDL) to do so.

This project focuses on the National Instruments (NI) Reconfigurable Input/Ouput (RIO) family of devices. These devices are commonly used and combine an FPGA with a great selection of Input/Output (I/O) capabilities in various form factors, making them very flexible when integrating them in ever-changing experimental environments. They also ease development by providing a graphical environment when creating the FPGA application, LabVIEW FPGA. Despite this, interfacing with the FPGA remains a complex issue as there can be practically an infinite number of possible FPGA applications.

To address the complexity of interacting with RIO devices, the IRIO Design Rules were introduced in 2015 by the research group Investigación en Instrumentación y Acústica Aplicada (I2A2). These rules standardize resource naming and functionality division.

Alongside these rules, the IrioCore library was developed. Due to the standardization of resources, it was possible to offer a simpler Application Programming Interface (API) for interacting with the FPGA from a host. IrioCore is a C library that allows developers to control the different parameters for an FPGA application running in a RIO device.

This project goes into detail of the issues detected with IrioCore, both in code and during maintainability and development. It then tries to mitigate or solve them by reimplementing it into a more modern language, C++, while also assessing the addition of quality improvements to its workflow.

1.2 Problems

Several issues were found with the previous library, those are:

- **Memory Issues**
  - Parsing files to pointers using delimiters.
  - Potential memory leaks.
- **Control Mechanism Issues**
  - Main control structure had multiple fields, some modifiable by users leading to undefined behaviour.
  - Lack of clear documentation for parameters requiring direct modification within the structure.
  - Increased complexity.
- **Dependency Issues**
  - Required to generate extra file manually in a Windows machine per each FPGA application
- **Quality Issues**
  - High technical debt.
  - Quality assessment relied on manually run functional tests and sonar analysis.
  - Requirements of the actual RIO boards
  - Lack of a clear Continuous Integration and Continuous Delivery (CI/CD) process.

1.3 Objectives

The main objective of this project is reimplementing the previous IrioCore library in C++ without lose of functionality. While also improving the previously stated problems and adding extra functionality. The objectives can be separated in the following points:

- Preserve functionality
- Add support for R series RIO devices.
- Enhance memory management practices.
- Minimize the dependency on Windows as much as possible.
- Implement functions to effectively manage all required parameters.
- Reduce overall code complexity.
- Establish clear Development Operations (DevOps) and CI/CD processes.
- Improve and expand documentation.
- Enhance testing procedures and reduce, if possible, dependency on physical hardware.
- Offer backward compatibility with the previous library

1.4 Document Structure

This document covers the work realized as the Master's Final Project (MFP) for the Master Universitario en Software y Sistemas (MUSS), focusing on migrating a C library for managing FPGA RIO devices to C++ and improving its quality and maintainability. Although primarily software-focused, the project includes a hardware component.

The document is structured as follows:

- **Background**: Provides context on the software and hardware concepts and tools used, and an overview of the original C library, its structure, use cases, and interaction with RIO devices.
- **Migration to C++**: Details the process of migrating the C library to C++, including implementation of key aspects and maintaining compatibility with C.
- **Proposed Architecture - IrioCoreCpp**: Describes the new architecture, its implementation, the problems addressed, motivation, compatibility with C, and the development environment.
- **Quality and Maintainability Practices**: Explains the DevOps processes, CI/CD pipelines, testing strategies, and provides guidance for future developers on extending and maintaining IrioCoreCpp.
- **Results**: Presents the evolution of IrioCoreCpp, compares it with the original C library using various quality metrics.
- **Conclusions and Future Work**: Summarizes the conclusions from the results and discusses potential future work.

Chapter 2

Background & State of the Art

2.1 Software Quality Engineering

2.1.1 Software Design Processes

Software design is an intricate process comprising numerous components, each contributing to its complexity. Factors such as project size, the composition of the development team, and the level of expertise among team members influence the design process.

This section examines these processes, showing how many have become inseparable from software design. Understanding how these processes model the development cycle is essential to modern software design.

2.1.1.1 Development Operations

DevOps is a methodology that aims to accelerate development and deliver products faster, shortening the development cycle while maintaining the quality of the software.

As depicted in Figure 2.1, the DevOps lifecycle combines software development and IT operations. By creating processes and tools and encouraging collaboration and communication between them, it aims to ease the transition that previously existed and eliminate the friction that could lead to delays or misunderstandings.

Continuous development and product release results in faster releases that can solve existing problems or integrate new features faster than other methods. It can be considered an extension or complement to the Agile methodology based on the idea that development should be incremental and iterative [1].

This section focuses on project management tools to facilitate software project development, quality, and deployment and discusses the importance of automating these processes as much as possible.

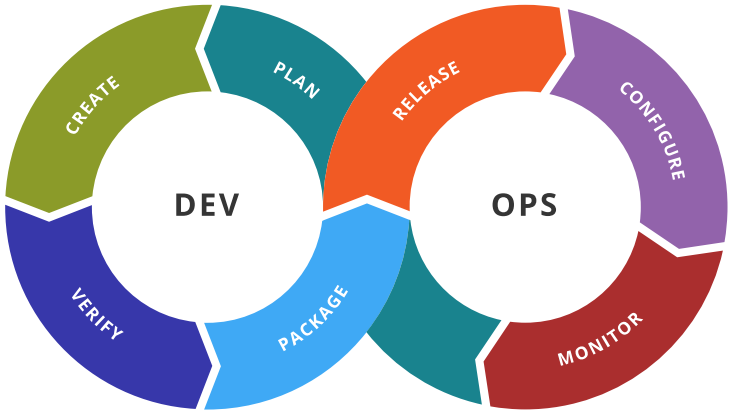

Figure 2.1: DevOps Lifecycle [2]

2.1.1.1.1 Continuous Software Automation

Developing software can be complex, with numerous steps or stages in converting source code into the final application. Furthermore, developing maintainable and high-quality software can make this process more confusing by adding additional stages that must be checked before releasing the final product. These stages could be varied; the main ones are compiling the source code, testing, and deploying the product. However, many other actions can be taken in any part of the cycle, like generating documentation, verifying that the code follows a structure, etc.

Continuous software automation forms a critical aspect of DevOps by helping take the necessary stages when developing software, automating these steps, as the name implies. Streamlining this process reduces the potential for human error and ensures that all code follows the same standards to ensure its quality and maintainability. In addition, it reduces the manual effort required and allows developers and IT teams to focus on the project.

Tools are available to facilitate the management of this process. They provide a framework for defining the stages and their dependencies, creating an order for the project lifecycle. Some examples of such tools are:

- **Maven**: A build and project management tool that uses pom.xml to manage dependencies and standardize the build process, emphasizing convention over configuration. It is mainly used for Java, but it is possible to use it with multiple languages.
- **CMake**: A build system generator that produces native build scripts for various platforms, simplifying the compilation process across different environments.
- **Make**: A build automation tool that uses Makefiles to define and manage the compilation and linkage of applications, widely used in Unix-based development environments.

2.1.1.1.2 CI/CD

CI/CD are a combination of practices fundamental in modern software development that aim to improve the efficiency and reliability of the software delivery process [3].

Figure 2.2 illustrates the CI/CD workflow as a process that goes from the source code up to the final product.

Continuous Integration (CI) consists of frequently building and testing the software. This can be done on every change when specific flags occur. It is crucial nowadays as multiple developers may work in the same code section and integrate other people's code. If testing is done appropriately, CI could assert the program's correct operation.

Continuous Delivery (CD) consists of producing new versions of the software quickly, and to make sure the latest software meets the requirements and quality set by the product owner, several checks must be passed before releasing it.

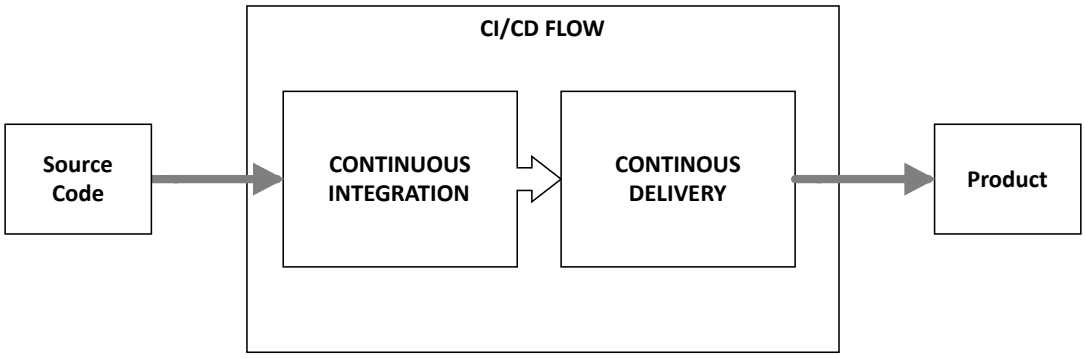

Figure 2.2: CI/CD flow

CI/CD practices are widespread in the industry nowadays and most, if not all, big software projects include them. One example of this is Netflix, which developed a custom CD platform for cloud applications. It allows the creation of specific pipelines for deploying the different components, introducing automated testing, canary deployments, and rapid deployment of hot-fixes if necessary. This ensures high availability and performance, crucial for the Netflix platform. Not only is CI/CD beneficial for managing the complexity of the platform, but it also allows teams to deliver features faster, thereby improving user experiences [4].

2.1.1.2 Testing

Testing is an integral part of CI in ensuring that the product meets the requirements and functions as intended. Software testing is an intense section of software development that is often overlooked but is crucial, providing value to the final product.

Numerous methodologies and techniques can be employed when creating tests, depending on the approach used when designing the test cases. Two of these methodologies are black-box testing and white-box testing, where the perspective of the software being tested is considered. In black-box testing, tests are written with inputs and expected outputs in mind, whereas in white-box testing, the internal structure of the code is known, and specific parts of it are addressed [5].

Testing can also be divided into different levels. Unit testing verifies the smallest units of code, while functional testing evaluates the functionality of the whole application.

There are also different approaches when developing software; one is specially focused on testing Test-Driven Development (TDD). TDD is an approach to software

development where tests are written before the actual code. The tests for new classes are written based on the expected behaviour (black-box), and then the code is written. If tests fail, it is due to the developed class not meeting the expected requirements [1] [5].

2.1.1.2.1 Unit-tests

Unit-tests are usually written by developers and aim to test the smallest section of code possible so as not to be affected by potential problems in other modules. They should be as independent as possible from different parts of the code. They should be possible to run without setting all the infrastructure necessary for a complete application test [1].

As they test the code itself, they follow a white-box testing approach, as it is necessary to know and understand the code; this is also why the developers write most unit-tests.

In this stage, it is common to use test doubles to emulate inputs from other modules that should feed the module under test. Test Doubles is covered in more detail in section 2.1.1.2.3.

Unit-tests are usually run with coverage tools to allow writing tests that cover as many lines of code as possible. A more detailed explanation of Code Coverage can be found in section 2.1.1.2.4.

2.1.1.2.2 Functional tests

Functional tests verify that the entire application functions as intended and often require the same or similar infrastructure as a production environment [1].

These tests emulate more typical user interactions with the software. However, error tests may also be created to ensure that if the application fails, it does so in a controlled manner.

Although these tests may be conducted using a white-box approach, treating them like a black-box or grey-box testing methodology is more appropriate, as this aligns more closely with the final user's product experience.

Functional tests are more commonly written by individuals external to the primary development project. This can be beneficial as it can also help identify issues that primary developers may have overlooked due to their intimate familiarity with the code or bias when assuming how a user may interact with the code.

2.1.1.2.3 Test Doubles

Ideally, an application should constantly be tested in the same environment if it will be deployed once released. However, this is not always possible due to the use of specific hardware or the inability to replicate the production environment due to time, complexity, or costs.

Figure 2.3 represents the complexity of real systems, where the module under test depends on several other components, which themselves rely on yet more components, increasing the complexity when testing. This is in part due to the fact

that developers may not always have control of these extra components. By contrast, test doubles allow us to circumvent these limitations. They allow developers to simulate the behaviour of other parts of the code, other modules or interactions with hardware, which can bring multiple benefits for testing.

There are three types of test doubles:

- **Mocks:** Objects that register calls they receive so you can verify if a certain method was called.
- **Stubs:** Objects that return predefined responses to specific calls.
- **Fakes:** Simpler implementations with working functionality but unsuitable for production.

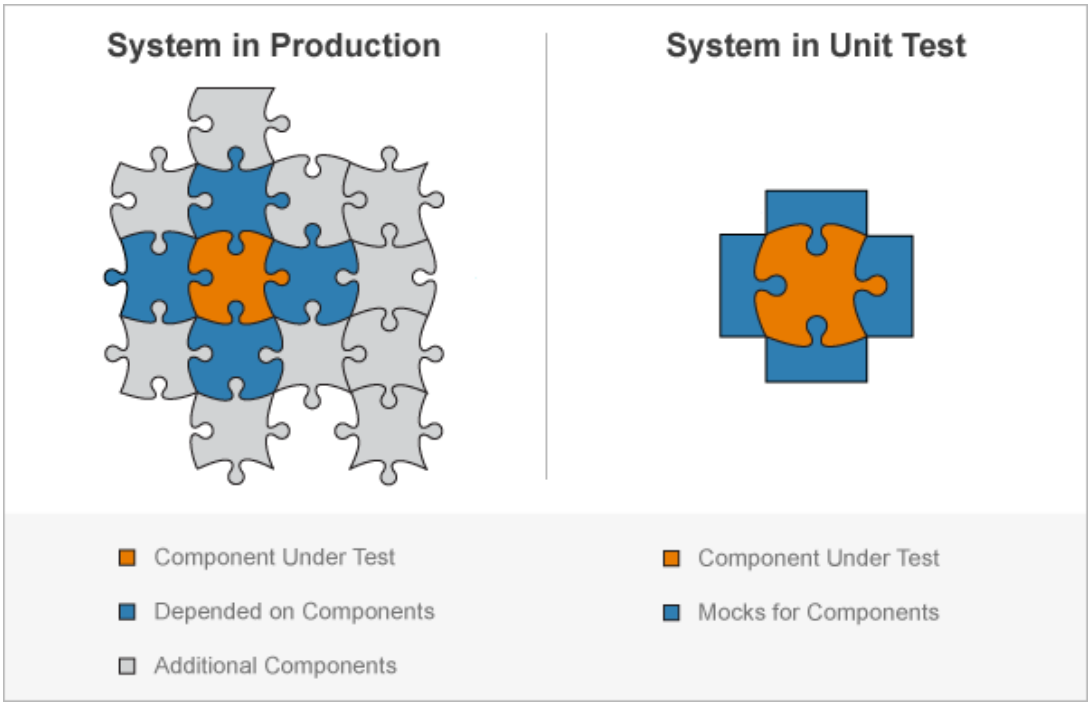

Figure 2.3: Test a component using test doubles [6]

Using test doubles usually results in faster tests, as they do not rely on external hardware or have large computational loads. This also facilitates running the tests continuously, as they do not require as much time.

Another critical benefit of test doubles is the control they provide, such as the ability to test the code that handles errors, as it can be challenging to recreate the conditions under which such errors may occur. For example, testing the code's behaviour in case of a connection interruption midway through the process. It can be difficult for traditional tests to time the interruption perfectly, as numerous external factors may affect it. With test doubles, it is possible to emulate a connection that stops after N packages, eliminating the timing constraints and making the test repeatable.

While test doubles are generally regarded as beneficial, they may also lead to potential problems that could have a more detrimental impact than a beneficial one. If not applied carefully, these objects may couple with the actual code, generating a dependency that will make the tests break easily when the code is refactored [7].

2.1.1.2.4 Code Coverage

Cove coverage measures the percentage of lines, functions, or branches executed during tests; it helps to understand the tests' effectiveness by indicating which parts of the code have been tested and which have not.

Depending on the quality requirements adopted, different types of code coverage can be applied or used in any combination. Line coverage measures the number of actual lines of code executed; function coverage is the number of functions that have been called from the code; and branch coverage is the number of possible combinations tested in conditional statements (if, while, for, etc.), between others. [8]

Test coverage serves as an indicator of how well existing tests check the different parts of the software application. It helps developers focus their testing efforts on specific parts of the code that may not have been thoroughly tested or edge cases that have not been covered.

Although test coverage may appear to be an effective indicator of the quality and robustness of a code, it is possible that it may provide a false sense of security, as bugs may still exist within the code. For example, line and function coverage only guarantees that a specific code section has been executed. Still, there may be paths within the code where bugs may exist, which could lead to unwanted behaviours. Branch coverage could help to reduce the possibility of issues; however, it also significantly increases the effort required for testing and the potential for creating an excessive number of tests, which may prove challenging to maintain in the long term. Still, code coverage provides a reasonable measure of the quality of the software when combined with other processes [9].

It is also essential to consider that test coverage should be regarded as an indicator of testing quality rather than a strict requirement. If taken as a requirement it may result in tests covering specific behaviours that may occur infrequently in reality. Significant resources must be allocated to simulate these conditions to create a test that could be more beneficial elsewhere.

The utility of code coverage remains a topic of debate. While both parties offer valid arguments, code coverage is still a valuable tool for improving tests and code quality [9]. Major organisations such as NASA rely on code coverage to increase confidence in mission-critical software, such as Orion and Draco missions [10][11]. Similarly, Google also uses code coverage as a measurement of its software, computing it for one billion lines daily across seven different programming languages [12].

2.1.1.3 Static code analysis

Static code analysis aims to detect possible points of interest where there may be a possible violation of programming practices. This means sections of the code may be sources of defects. This analysis is static because the code is not compiled or run; it is a pure source code analysis. Humans can perform static code analysis, but it is usually automated to some extent by tools that report possible zones of conflict [13] [14].

There are numerous types of static code analysis, one of the most prevalent being syntax or grammar checking. Developers typically run this during the development process by default, as it has been integrated into most, if not all, Integrated

Development Environments (IDEs). Syntax checking ensures that the code adheres to the appropriate language grammar rules, highlighting any discrepancies that would most likely prevent the program from compiling. It is frequently bundled with an autocomplete function, facilitating development.

Other forms of static code analysis include style enforcers, also known as linters. These are sets of rules, different from the ones applied by syntax checkers, which check the code and follow standards such as variable names, parameters, spaces/tabs, etc. They are beneficial in organisations with multiple developers, as they allow the code to follow a similar pattern and be easier to understand. There are more types of static analysis, such as code complexity or code duplication.

As stated before, static code analysis identifies *potential* defects; therefore, it is not uncommon to flag parts of the code with no problem. Such false positives would require a human to analyse and determine if it is a problem.

2.1.1.4 Version Control Systems

In any project, it is crucial to have a version history that displays how the project has evolved, to see who made those changes, and, if needed, to revert to previous versions. Version Control Systems (VCS) tools allow these features.

VCS in software development is especially important due to how common it is for multiple developers to work on the same project. These tools allow developers to work independently and combine their efforts more quickly than manually.

VCS can be deployed locally on the developer's computer. However, it is typical to work in tandem with repositories, which are remote hosts where the VCS is stored and where another developer can retrieve it, facilitating collaboration and providing backup of the source code.

VCS can be divided into two groups, Centralized Version Control Systems (CVCSs) and Distributed Version Control Systems (DVCSs). In CVCS, the code is stored in a central repository, from which developers get the code, make changes and upload those changes to the repository. DVCS changes this by creating a local copy of the repository on each developer system, allowing developers to merge, change branches and only commit the changes locally and upload all or parts of it to the central repository [15].

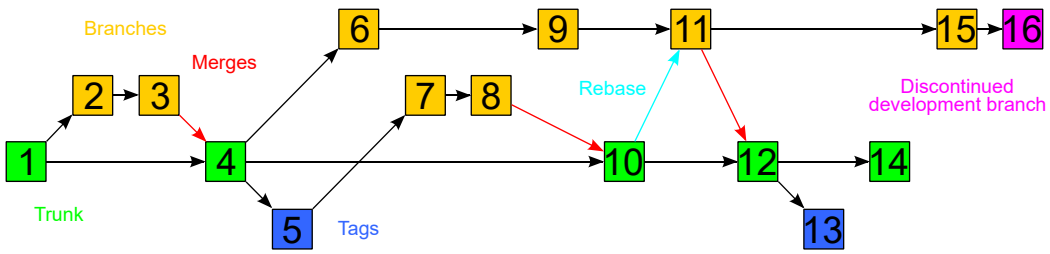

Figure 2.4: Example version control system [16]

2.1.1.5 Documentation

Documentation is an essential part of a software project's final quality. Documenting the software's function is crucial for users to understand or for future developers to maintain or expand it.

Documentation can come in several ways; the two most common are documentation in the form of user manuals or API documentation in which it is explained how the software is expected to be used; the other one is code comments, as programming language statements can not always capture all the necessary information needed to understand it without context. This is why, while it is important to write code that is as self-documenting as possible, sometimes it is not possible. Comments should be added describing the things that are not obvious from the code [1].

While writing comments is essential, adhering to coding standards is also important. These standards help understand the code and its maintainability and help it be followed quickly.

2.1.2 Software Design Techniques

Section 2.1.1 Software Design Processes discussed processes commonly employed in software development. This new section will cover more specific tools and techniques utilised in these processes.

2.1.2.1 Make

Initially, Make was created by Stuart Feldman in 1976 at Bell Labs to assert that his executables always included the source code changes and did not depend on humans to include them [17].

Specifically, Make is a software tool that allows building executables and libraries from source code while managing file dependencies with other files in the same project. It also figures automatically which files have changed or not, allowing only partial recompilation of the code when changes occur, drastically reducing compilation times. Make it is also not limited to specific languages, as the commands for compiling the source code are defined entirely by the developer [18]. Although Make is more commonly used for build automation, it also allows implementing it in multiple processes, which can include several DevOps operations.

Make is configured through Makefiles, which are the files that control how the source code is compiled. They define rules, build targets, and their dependencies. A rule is a series of commands that must be executed to create or execute a target, which can be a file or another rule. The rules have dependencies, files or rules that must be fulfilled before executing the rule.

Other software management tools may offer more flexibility and ease of use than Make. It is a widespread tool used in multiple projects, such as the Linux kernel, Mozilla Firefox, and LibreOffice. This is due to its relatively simple setup requirements and proven reliability over many years. Make has stood the test of time, demonstrating its effectiveness in managing build processes without needing a lot of extra configuration

2.1.2.2 GoogleTests

GoogleTest is a C++ testing framework developed and maintained by Google [19]. It allows the creation of any test (unit test, functional or integration). Also, it is supported in Windows, Mac and Linux, making it a good choice for testing cross-platform software.

The base of this framework is the assertions, which are statements that check whether a condition is true. The test uses these assertions to verify the code's behaviour. These assertions form part of a Test Case, a specific set of instructions to test some part of the software under test. These Test Cases can be grouped into groups of tests with similarities, which are called Test Suites. Test programs can also have one or more Test Suites. These assertions and Test Cases and Suites can be created by using macros that GoogleTest provides, giving test developers simpler syntax and cleaner tests.

GoogleTest also provides test fixtures, which allow the reuse of code for tests that use the same or similar configuration and objects. Test fixtures can be executed per Test Case or at the beginning of executing a Test Suite.

It also provides useful options for testing, such as filtering tests to run a subset of all available ones, repeating tests any number of times to detect tests that may fail only occasionally, or test shuffle to ensure that there are no dependencies between tests which have not been taken into consideration.

2.1.2.3 Fake Function Framework (fff)

Fake Function Framework (fff) framework designed to create fake C functions for unit testing. It is lightweight, a header-only library that can be easily included in projects with minimal modification to build configurations [20].

As its name suggests, fff allows creating fake objects to replace calls to C functions when unit testing. It is also possible to create stubs and mocks if necessary. It uses macros to generate the fake objects and assign return values or behaviours if necessary.

2.1.2.4 Cpplint

Cpplint is a static code checker for C/C++. It inspects the source code files, searching for style issues using Google's C++ style guide [21] [22].

Cpplint is a straightforward Command-Line Interface (CLI) application that is easy to use. Select the folder with the source files, and it will search them recursively and output the errors found. Yet, it also provides methods to customise the parsing, such as filtering some guidelines or files. This can be achieved using CLI arguments or, more commonly, configuration files, which are useful for CI actions as they allow keeping a history of the parameters used and their changes if using VCS.

2.1.2.5 Doxygen

Doxygen is a documentation generator tool that automates this step. It parses the code to extract information about classes, functions and variables from source code comments. It supports multiple outputs such as LaTeX, PDF and HTML [23].

It uses special codes to differentiate comments about specific parts of the code from documentation about variables, functions, or classes. The syntax is very similar to Markdown, supporting most of its features, which makes it easier to use, as Markdown is a very widespread syntax for software documentation.

Doxygen enables the automatic generation of diagrams from source code, including class and collaboration diagrams, which provide a useful overview of the software project. Additionally, it generates cross-referencing between related elements, such as going to the documentation about the type of a function argument, simplifying navigating through documentation.

2.1.2.6 Git

Git is a Distributed Version Control System that facilitates tracking file changes with speed and efficiency [24]. It enables multiple developers to work on a project simultaneously, allowing them to merge their changes when necessary. Its support for branching facilitates the integration of most types of workflow easily, accommodating almost any kind of developer, making it a widely used tool in software engineering.

2.1.2.7 GitHub

GitHub is a web-based platform, founded in 2008 and acquired by Microsoft in 2018, that enables developers to create, store and share their code. It is typically employed as a remote repository for Git repositories. However, it offers additional functionalities such as issue tracking, graphical visualisers for commits and diffs, CI/CD and wikis, etc.

2.1.2.7.1 GitHub actions

GitHub Actions is a CI/CD platform that allows developers to automate CI/CD pipelines. While its main focus is DevOps operations, the workflows can also be integrated with other parts of the GitHub environment, such as issues, GitHub Pages, etc. GitHub provides pre-built virtual machines on which to run these workflows [25]. It also allows to release these actions which can be reused in other workflows, if they are released in the GitHub marketplace, they are available for its integration in other workflows. A lot of bug companies such as Docker and Sonar, have their have created their own actions helping developers implement their DevOps operations in workflow using their tools easily.

GitHub actions are structured in the aforementioned YAML files, which are called workflows, which is each of the stages of the pipeline. These workflows can be configured to be triggered manually, scheduled, or by one or multiple event in the repository, such as pushed, pull request, tag creation, etc.

Each workflow then is has jobs, which are set of instructions, each running in a virtual machine. It is also possible to specify to create a container from an image on which the required command will be executed. It is also possible to upload artifacts, which are byproducts of the steps of each jobs which then another job in the same workflow could download and use, or for posterior analysis by a developer, for example a case where tests fails and the log is uploaded as an artifact which can be later downloaded by the developer to understand the problem. Dependencies between

jobs can be established so they might run in parallel or sequentially depending on the structure of each pipeline.

2.1.2.8 SonarQube

SonarQube is a static analysis tool by Sonar that inspects code quality for several programming languages. It offers information about potential sources of error, code repetition, code complexity, coverage, security risks, etc. This information allows developers and project managers to understand the project's status and quality better.

If the analysis is integrated through CI, it can serve as a warning if these minor issues accumulate. To inform developers of these changes, SonarQube provides quality gates, which are conditions that must be met to achieve scores. These scores are used as indicators of the project's quality.

Additionally, there is SonarCloud, a cloud-hosted and subscription-based version of SonarQube. It is designed for open-source projects or small to medium-sized businesses, whereas SonarQube is more suited to enterprise use.

2.1.2.9 Docker

Traditionally, applications had to run in a full Operating System (OS) on which all the dependencies and configurations needed to be configured. This could cause incompatibilities if multiple applications were running in the same system. Virtualisation could mitigate this problem; however, running an entire OS for each application would result in significant resource overhead [26].

Docker offers a solution to this problem by encapsulating the necessary resources in a small package called an image, which contains everything the application needs to run. When this image is executed, it creates a container, an isolated instance that runs the application. Containers share the host OS kernel but operate in their own isolated user space, minimising resource usage. A visual comparison between virtual machines and containers is shown in Figure 2.5.

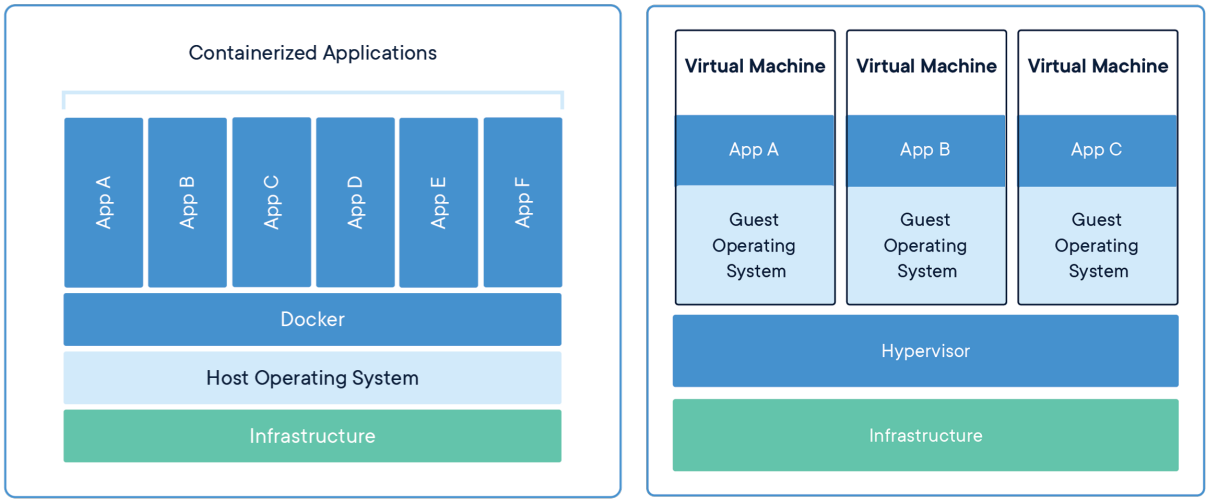

Figure 2.5: Docker Container vs. Virtual Machine [27]

In addition, the deployment mechanism of Docker is straightforward and portable across environments, ensuring that the same application can be executed consistently regardless of whether it is a development or production environment. Docker also integrates well with CI/CD pipelines, facilitating the creation of containers for building, testing, or deployment without the need to set up or maintain a specific machine.

2.2 Data Acquisition in FPGA Systems

2.2.1 FPGA Basics

FPGAs are devices capable of being reconfigured to suit many applications, thus offering tremendous flexibility. Unlike ASICs, which have a fixed functionality, FPGAs hardware logic can be reconfigured, which facilitates prototyping and is very helpful in iterative development, decreasing time and even costs compared with ASICs development cycles.

Figure 2.6 depicts the basic architecture of FPGAs devices, they are composed of several modules, Digital Signal Processing (DSP) blocks, Random Accesss Memory (RAM) blocks and Adaptive Logic Modules (ALMs). These modules are interconnected by a grid of configurable routing, which alters the electrical inputs and outputs these modules interact with, this grid is what allows reprogramming an FPGA for a virtually unlimited number of applications [28].

One of the major key benefits FPGAs offer is how tailored an implementation for a specific problem can be. This specialized implementation can improve performance and energy consumption compared to conventional software running on general-purpose processors while offering flexibility to fine-tune the algorithms and make changes compared to ASICs.

One of the main disadvantages of FPGAs is that some applications can easily require more logic resources than available, which may require compromises or adjustments to implement complex designs.

Another possible source of issues is the complexity of modelling the required hardware. Usually, HDLs are used, such as VHDL or Verilog. Using these languages can be complex due to working with electronic circuits and being different to more common computing languages, necessitating expertise that may take some time to acquire. High-level synthesis tries to help bridge these gaps by providing a more programming-like paradigm to develop applications. However, there is still a need to understand the hardware below at some level to achieve good results.

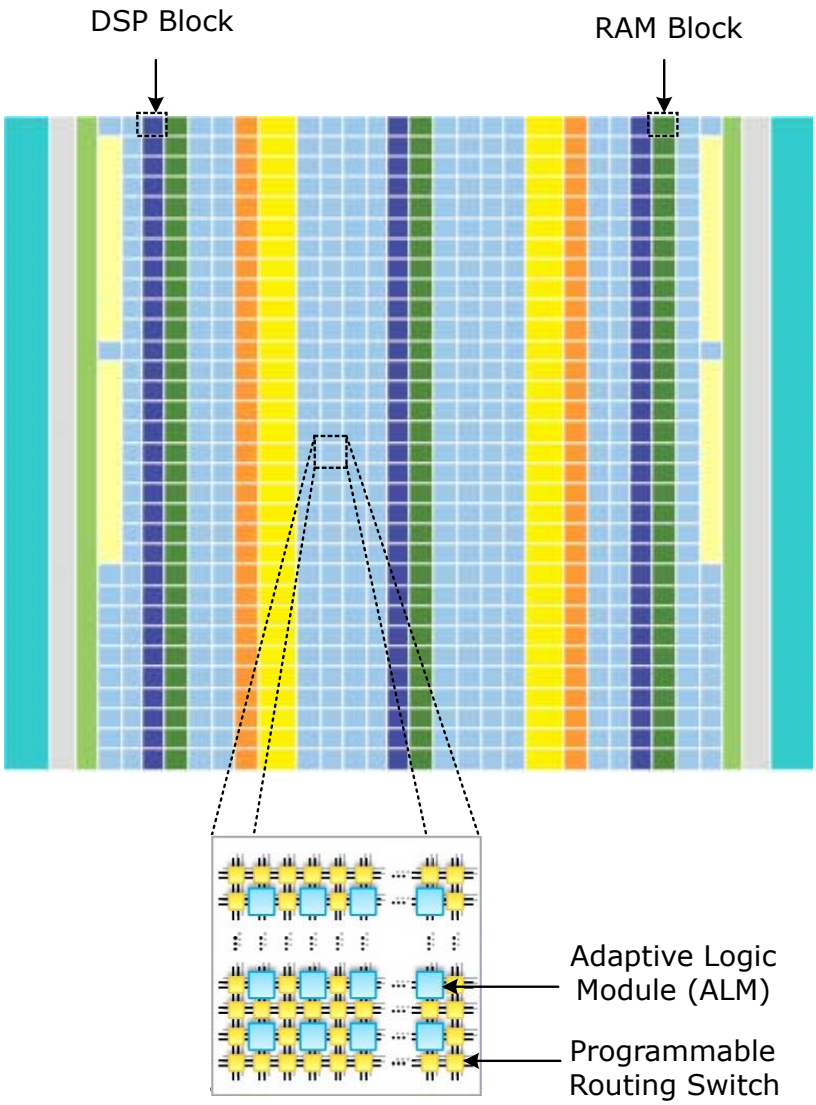

Figure 2.6: FPGA Architecture [29]

2.2.2 FPGA-based Systems

FPGAs are widely used in applications that require high performance, deterministic behaviour, or/and real-time response, such as digital signal and data processing [30][31][32]. One environment where FPGAs are a key component is in Big Science projects, such as fusion, as they have strict requirements for performance and real-time response to be able to react to the rapid changes of this environment, acquiring the necessary data while also protecting the equipment, such as interlock systems [33].

Systems using FPGAs are complex, as illustrated in 2.7. FPGAs comes in different formats [34], some require an external system that acts as host while others provide System-on-Chip (SoC) based on FPGAs, which integrates all the requirement components such as Central Processing Unit (CPU), RAM, peripherals and the FPGA in one package, which can then be run autonomously an independently of other systems, while also providing fast access to the FPGA resources.

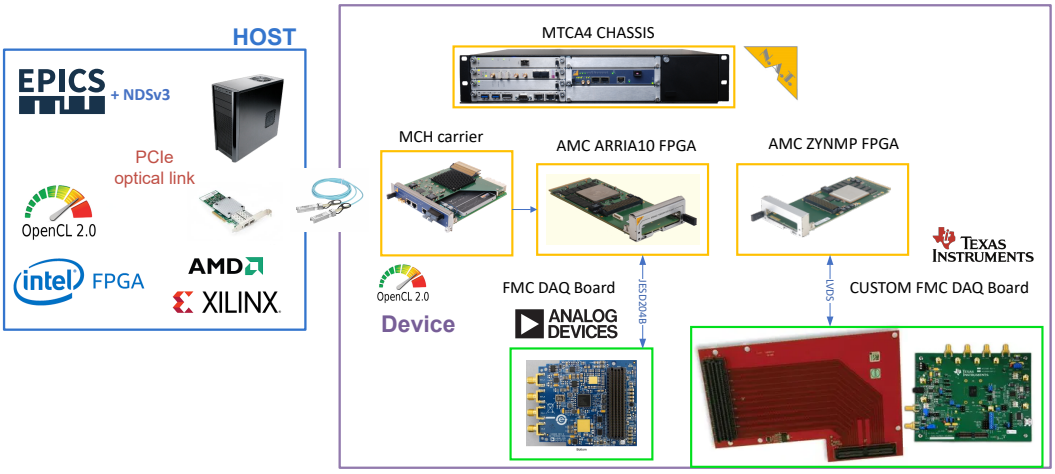

Figure 2.7: Application System using FPGAs [34]

2.2.3 RIO Devices

NI RIO devices are composed of a FPGA with reconfigurable I/O capabilities, which give developers the ability to define their own custom measurement and functionality. They are commonly used in data acquisition and signal processing applications with real-time limitations such as timing or synchronization. They offer multiple form factors, such as Peripheral Component Interconnect (PCI), Peripheral Component Interconnect Express (PCIe), PCI eXtensions for Instrumentation (PXI), PCI eXtensions for Instrumentation Express (PXIe) or Multi-System Extension Interface (MXI).

Figure 2.8 offers a simplified view of the RIO architecture, where sensor and actuators, such as motors, can be driven through their I/O modules. All driven by the logic implemented in the FPGA.

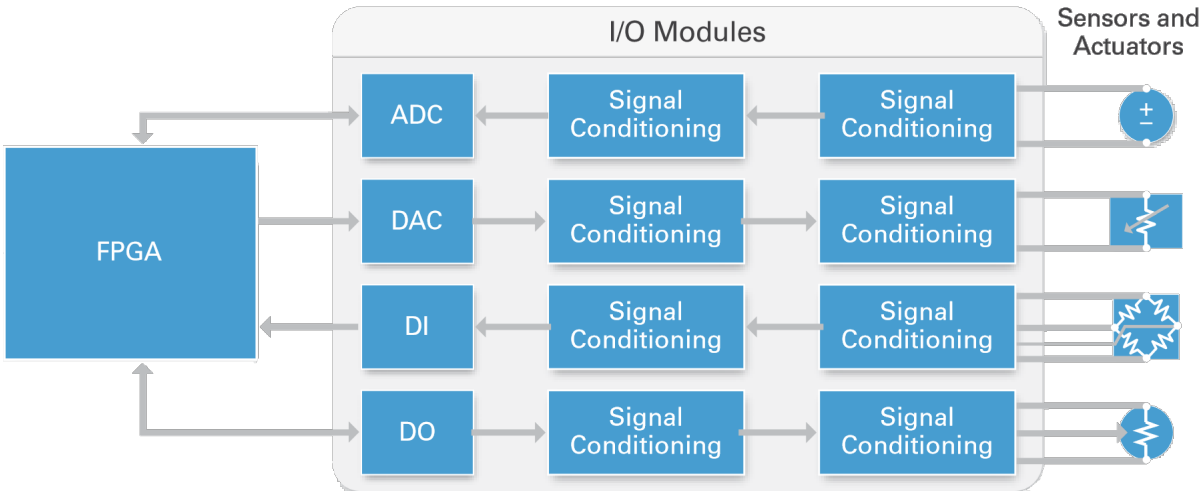

Figure 2.8: RIO Architecture [35]

One of the main advantages of RIO devices is that the FPGAs can be configured through Laboratory Virtual Instrument Engineering Workbench (LabVIEW). This greatly facilitates the creation of FPGA applications thanks to being a graphical programming environment and a widely utilized tool for these applications.

There are several families of RIO devices, each tailored to specific applications and requirements. The following sections will explain their differences, but they can be summarized in terms of form factor, acquisition rate, and I/O [36].

2.2.3.1 cRIO

Compact RIO (cRIO) devices are modular and rugged systems designed for industrial applications. They are composed of a chassis with the FPGA and multiple slots for modules, which can range from analog/digital adapters to motor drivers and interfaces for various industrial standards. The connection to this system is usually via Universal Serial Bus (USB), Ethernet or MXI.

Their ruggedness allows them to operate in harsh environments (electromagnetic fields, temperature, vibration, etc.), which other members of the RIO family may not be suitable for.

Figure 2.9: cRIO

Another main difference with other RIO devices is that cRIO support more I/O but at the trade-off of slower acquisition rates. Therefore, these systems are oriented to robust industrial controllers with high reliability as an alternative to Programmable Logic Controllers (PLCs) [37].

2.2.3.2 FlexRIO

Flexible RIO (FlexRIO) devices are designed for fast data acquisition and processing of acquired signals. Connection to these devices is commonly through PXI/PXIe or PXI/PXIe. A typical application is multiple FlexRIO boards into a PXI/PXIe chassis, each board with its module, which can range from analog/digital to cameras to custom modules specific to the application. The acquisition rate of these boards is significantly faster than cRIO.

Figure 2.10: FlexRIO with module

2.2.3.3 R Series

R Series devices are comparable to FlexRIO ones. Yet, the R Series lack the flexibility of modules and instead integrates analogue/digital I/O capabilities and the FPGA on the board itself. R Series are also available in standard form factors, such as PCI/PCIe or PXI/PXIe.

Figure 2.11: R Series

2.3 IRIO

Developing applications for RIO devices is a multi-step process. On the FPGA side, the device must be configured for the specific application. This is achieved through the use of LabVIEW, which, through graphical programming, allows developers to define the desired behaviour. Once this is complete, the configuration information is compiled into a bitfile, an Extensible Markup Language (XML) file containing all the configuration information describing the internal logic and digital circuit of the FPGA. This may be sufficient for some projects, as it may not require outside interaction and may run *standalone*. However, it is often accompanied by an application that interacts with the FPGA, reading, writing or controlling when it starts or stops. The way to interact with the device is through *resources* and *Direct Access Memory (DMA) channels*. The developer configures these in the LabVIEW application.

While a developer may implement any application, RIO devices are more suited to specific applications, which may result in similar results. To address this, IRIO was created, combining a Linux software library and a set of rules that dictate how and which resources must be configured. These rules are known as the IRIO Design Rules, and a copy can be found in [38]. The Linux library associated with IRIO, called IrioCore, simplifies interfacing with the low-level driver of RIO devices; this is illustrated in Figure 2.12.

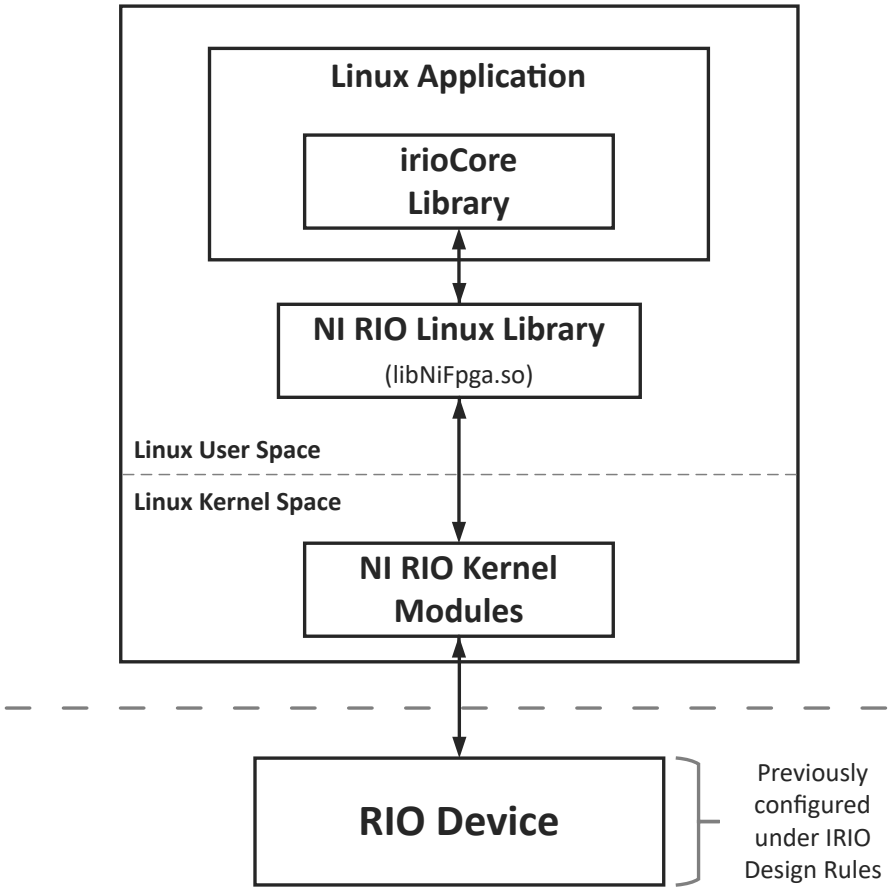

Figure 2.12: IrioCore Library Structure

The design rules also establish which resources are needed for each type of application and what naming convention should follow their resources to facilitate the development and maintainability of applications for RIO devices. Having naming conventions allows IrioCore to extract their addresses, enabling easier identification, which is crucial as the addresses can vary between applications and compilations. These addresses are obtained through the `C API Generator`, a Windows application provided by NI, which must be executed following the compilation of the bitfile. This generates a file later passed into the IrioCore library for parsing.

It has been mentioned that there are multiple types of applications supported by IrioCore for RIO devices; they receive the name of Profiles. They are:

- **Data Acquisition (DAQ)**: High throughput acquisition of signals from real world events such as sensors. The data is stored in DMA to be transferred quickly from the FPGA to the device. Oriented to real time applications working with these signals.
- **Image Acquisition (IMAQ)**: Similar to DAQ but with images. The data acquired are collection of values representing pixels. Processing of this data can also be part of the application, making use of the advantages of FPGAs for performance and its deterministic behaviour.
- **Point By Point (PBP)**: Designed for acquiring and generating analog and digital I/O operations on a per-sample basis. It is not intended for long-duration data collection or high-throughput applications.

Table 2.1 provides a detailed overview of the supported profiles and specifies the corresponding RIO boards that are compatible with each one.

Profile	Description	Boards supported
DAQ	Analog signal data acquisition	FlexRIO
		cRIO
		R Series
IMAQ	Acquisition of images	FlexRIO
PBP	Acquisition of specific analog samples	cRIO

Table 2.1: Profiles Supported by IrioCore

IrioCore and their design rules have been used in several Big Science fusion-related experiments, such as KStar [39] and ITER [40], while also being used for multiple published research papers and projects [36] [41] [42] [43].

2.4 Migration from C to C++

Migrating an existing library from C to C++ (understanding C++ as C++11 and above) must be studied beforehand as it may not always be necessary. C can be a good choice for libraries if the developers want closer control than in other languages. C++ also allows having the same power as in C but adds new features like Object-Oriented Programming (OOP), template meta-programming, Standard Template Library (STL),

ways to manage pointers safely, etc. However, this added complexity may lead to pitfalls, which is why it is not always recommended doing so.

2.4.1 Planning

As with all software projects, starting coding without sufficient preparation can lead to complications. When migrating code from C to C++, it is crucial to comprehend the existing structure, dependencies, and complexity. This can help understand how the various components interact, which can assist in the design of the C++ interfaces. When coding in OOP, it is recommended to follow principles, such as SOLID [44], and design patterns for cleaner code.

Establishing clear objectives for the migration is also essential, as they may not always be evident. Migration has many potential justifications, such as improving maintainability and memory safety.

2.4.2 Maintainability and usability

The use of OOP principles facilitates the implementation of more effective organisational structures. Introducing classes and objects enables a more modular approach, facilitating comprehension and maintenance. Furthermore, the utilisation of inheritance, polymorphism and templates enhances code reuse, which also contributes to maintainability and helps to avoid bugs, for example, those that can result from changes made in one section of the code, which is the same as in another part. It also generates a higher level of abstraction, simplifying the developer's use.

Additionally, access specifiers can create cleaner interfaces, hiding objects and methods that should not be used. In C, this may not always be possible or straightforward, resulting in users utilising inadequate variables, structures, or functions, potentially leading to undefined behaviour.

Moreover, using class constructors allows initialising other resources, which in C may have required the user to do it manually in a specific order. This reduces the probability of errors and simplifies its use. In contrast, destructors provide a mechanism to close, clean, and free required resources, even if the user does not explicitly clean them. This also applies in case of errors, as the destructor is called object destruction. Consequently, developers can rely upon this mechanism to initialise and destroy objects appropriately without needing a more profound comprehension of the module.

2.4.3 Memory management

Memory management is prone to errors. C and C++ let the developer handle memory allocation and deallocation, but C++ also provides mechanisms to help avoid potential problems. One of these features is smart pointers (such as `std::unique_ptr` and `std::shared_ptr`), which handle memory allocation and deallocation through their constructors and destructors, respectively.

This concept is called Resource Acquisition Is Initialization (RAII), which states that the life cycle of a resource (memory, socket, file, etc.) must be bound to the lifetime of the object using it [45]. This allows developers to create safer code while reducing the code complexity associated with manual memory management.

2.4.4 Error handling

In C, error handling is traditionally managed through return codes, where each function call returns a code indicating whether the operation was successful. Ideally, this would need to be checked after each function call, which can litter the code rapidly. C++ offers a more sophisticated error-handling mechanism than C. By using C++ exceptions, developers can more gracefully handle these situations, leading to cleaner and more maintainable code. Additionally, exceptions provide a structured way to handle errors, separating error-handling logic from the main program. This improves code readability and reduces the risk of resource leaks and undefined behaviour associated.

2.4.5 Compatibility

The newly created C++ library may still be required to support the other applications with the C library as a dependency. This is possible by creating "*wrappers*" that encapsulates the use of OOP and other C++ features into C code.

Name mangling in C++ enables features such as function overloading, classes, and polymorphism. To allow calling a C++ library from C, it is necessary to use `extern C`, which disables name mangling. This instructs the compiler to generate the references to the function in the same manner as it is used in C.

In the context of compatibility, two types can be considered. One of those is API compatibility, which is achieved using the same function and variable names, so the code using the new library would not need to change to compile successfully. The other type of compatibility is known Application Binary Interface (ABI), where the internal organisation of the memory stack and how the parameters are passed are identical to the previous C library. If the application is utilising the library via a shared library, it would be entirely transparent to the application whether it is using the previous version or the new version if it is ABI compatible.

Chapter 3

Proposed Architecture: IrioCoreCpp

3.1 Stated Problem

IrioCore was created in 2015 by the I2A2 research group with a relatively short development time, leading to several issues regarding quality and maintenance practices, resulting in shortcuts and the accumulation of technical debt. This made it challenging to maintain or refactor code to solve existing issues.

One of the main issues identified was management while operating using strings and memory accesses in general. The approach of parsing files using pointers to strings and delimiters created an environment where these types of vulnerabilities could arise. Additionally, there were concerns about potential memory leaks due to certain code paths where memory might not be freed in the event of non-fatal errors due to low coverage results.

The primary control mechanism in IrioCore involved a structure with multiple fields, some designed to be modifiable by the user while others should not. Modifying these values could lead to undefined behaviour. The library provided some getters and setters for specific parameters, but many parameters still had to be modified directly within the structure without clear documentation. This increased the developers' complexity and the risk of undefined behaviour and memory leakage.

Furthermore, IrioCore had a hard dependency on Windows and LabVIEW, requiring an additional manual step of running the bitfile through the `C API Generator` tool to create a header file for the library to parse. This extra step added complexity and dependency on the `C API Generator`, making the system fragile and prone to breaking if changes occurred in this tool.

The quality of IrioCore was assessed through functional tests and sonar analysis, which must be run manually. Initial testing involved executables without a testing framework, and later, GoogleTests were introduced to standardize and automate tests. However, these tests required RIO boards and the configuration of devices, making the testing process not easily repeatable and lacking a clear CI/CD process. Coverage was also not the primary focus, which could lead to potential problems.

3.2 Motivation for Developing IrioCoreCpp

Given the various issues with IrioCore, the decision was to develop a new library, IrioCoreCpp, instead of updating the existing one. The primary reason behind this was, to avoid being conditioned by existing code when addressing the identified problems, provide a new perspective, and introduce new features that benefit the end users.

IrioCoreCpp aims to mitigate potential vulnerabilities and memory leaks by leveraging modern C++ features such as RAII and OOP. Utilising C++ strings and smart pointers reduces the likelihood of memory-related issues and enhances overall security.

The new library also addresses the control mechanism issues by providing getters and setters for all parameters and encapsulating variables and functions within classes using access modifiers. This encapsulation ensures that the users can only interact with the intended parts of the library, reducing the risk of undefined behaviour.

IrioCoreCpp eliminates the manual step of running the bitfile through the `C API Generator` by parsing the bitfile directly. This change makes the system more robust and removes the dependency on the `C API Generator`, ensuring a more streamlined and reliable process.

Moreover, the development of IrioCoreCpp provided an opportunity to implement new features from the design stage, making their integration easier. While IrioCore supported only FlexRIO and cRIO boards, IrioCoreCpp extends support to R Series boards, encompassing all types of RIO devices.

3.3 Structure

Figure 3.1 represent the internal structure of IrioCoreCpp, showing how the user interacts with the application, how it is internally structured, and the interactions between these internal elements and the FPGA. This section introduces the elements shown in the figure and their interactions, while subsequent sections of this chapter provide more detailed explanations of how each element was defined and its specifics. It also covers the motivations behind upgrading the library. It also covers the rationale behind the decision to upgrade the library.

IRIO Design Rules define resources, which are the FPGA elements that serve as a way to interact with the program, some of them being optional depending on the profile and the application's needs. IrioCore grouped most of these resources by files, IrioCoreCpp defines an object, the terminal, specific to resources that serve a purpose. Depending on the profile, some terminals are created. The terminal constructor is also responsible for searching for the appropriated resources in the bitfile and checking their validity if necessary. IrioCoreCpp terminals are also divided into more groups than in IrioCore for better granularity and maintainability, making it more straightforward where each resource is parsed or used.

It was decided to create a separate library to parse the bitfile and extract the necessary resources. This library is called BitFile Parser (BFP). IrioCoreCpp and its terminals interacts with the BFP library to extract the necessary resources.

Subsequently, the terminals are grouped into profiles, creating the required terminals at construction and providing access to the user so they can interact with the FPGA.

The current supported profiles are Data Acquisition (DAQ), Image Acquisition (IMAQ), and Point By Point (PBP).

The profile is created by the main module of IrioCoreCpp, which reads from the FPGA the resources that determine the board type and the required profile. It then determines whether the profile requested is valid for the current board and, if so, the profile is created. The main module is also responsible for interacting with the FPGA for functionality that cannot be grouped into terminals. This includes starting the application in the FPGA or opening a session.

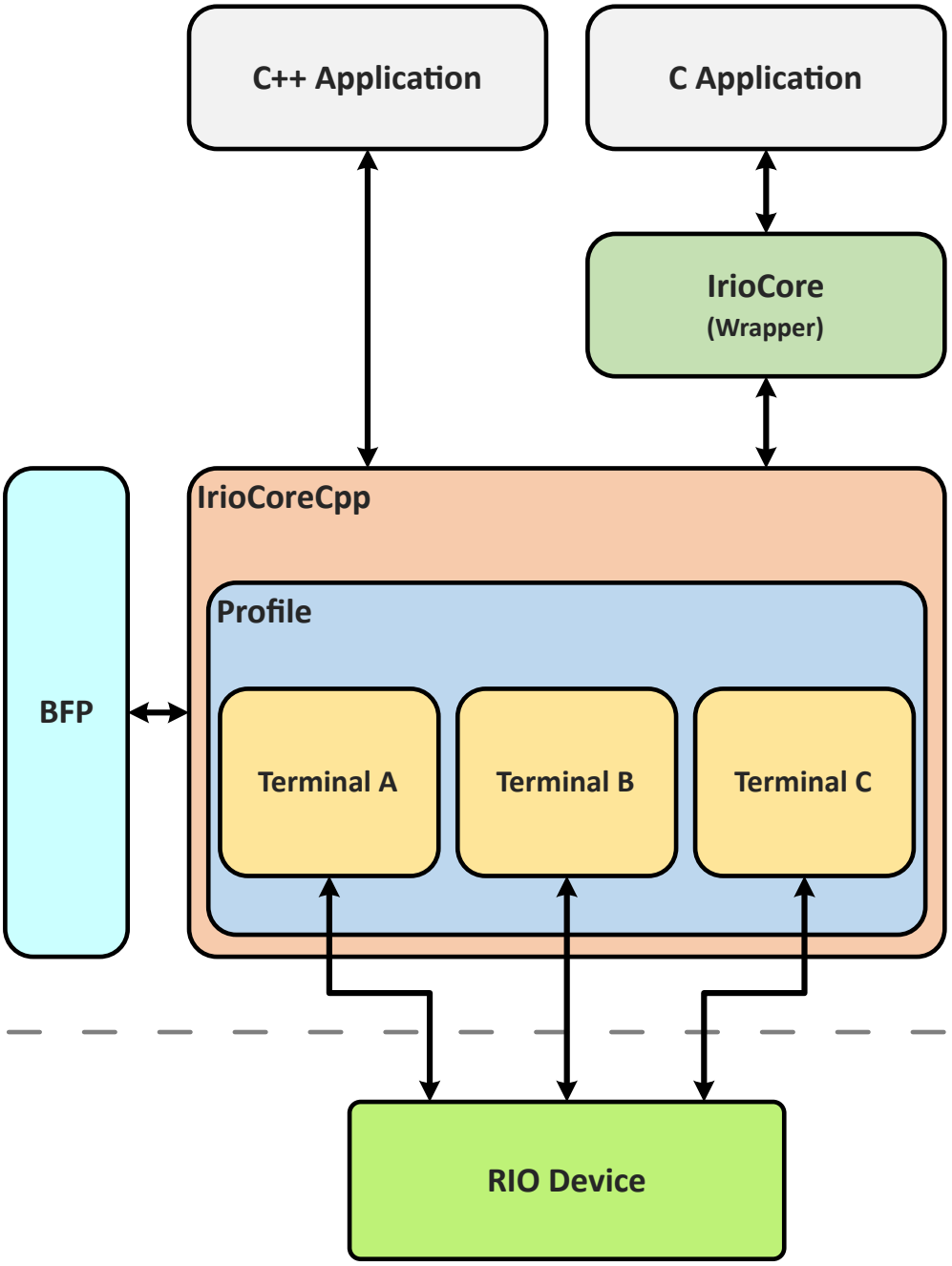

Figure 3.1: IrioCoreCpp Library Structure

3.4 Elements

3.4.1 Resources

The term "resources" is not to be confused with the logic gates, Lookup Tables (LUTs), DSPs, and other elements that are configured in an FPGA to create a specific application. In the context of IRIO, "resources" refers to the elements configured in LabVIEW, which serves as I/O for the user application. These elements can be considered as variables in a programming language. In the case of IRIO, the names of these resources follow the naming convention specified in the IRIO Design Rules. In LabVIEW nomenclature, these elements are called terminals; however, as it is the same name used for the classes in subsection 3.4.5, it was decided to refer to these elements as Resources.

Resources in IRIO can be categorised into three types depending on their number of elements and the required throughput needed for reading or writing operations:

- **Registers**: Single elements. Used for monitoring FPGA values and controlling the application behaviour.
- **Arrays**: Same as registers but a collection of the same data type of elements. They can be multidimensional, up to 3.
- **DMAs**: High-throughput, large data transfers between the FPGA and the host system.

These Resources can also be inputs or outputs. In the case of registers and arrays, they are referred to as control and indicators, respectively. While DMAs are differentiated according to whether the data is transferred Host to Target (HtT) or Target to Host (TtH).

The data type of the Resources can be integers of several lengths, from 8 to 64 bits, or decimal, both fixed and floating points. In the case of floating point data, it is recommended to minimise its use and, if possible, substitute it with fixed point data, as it consumes a significant amount of logic resources and may not fit in the FPGA.

3.4.2 Bitfile Parsing (BFP)

The information about the FPGA resources is contained in the bitfile, an XML file with the information about them and the data to configure the FPGA for the specific application. IrioCoreCpp needs to know the addresses of these resources to interact with them. However, as any bitfile may be loaded into the FPGA, a runtime parsing is needed to extract the addresses of the resources required by the IRIO Design Rules.

To parse the bitfile's XML, a third-party library is used, pugixml. Pugixml is a lightweight, easy-to-use and fast XML parser for C++, distributed under MIT license, which allows it to be used in IrioCoreCpp.

BFP uses pugixml to extract resource information and create objects representing the types of resources found in FPGAs. The bitfile separates the resources into two sections, Registers and DMA. As depicted in Figure 3.2, both share several similitudes, so a base class, Resource, was defined. Two specialisations inherit the Resource, DMA, and Register classes.

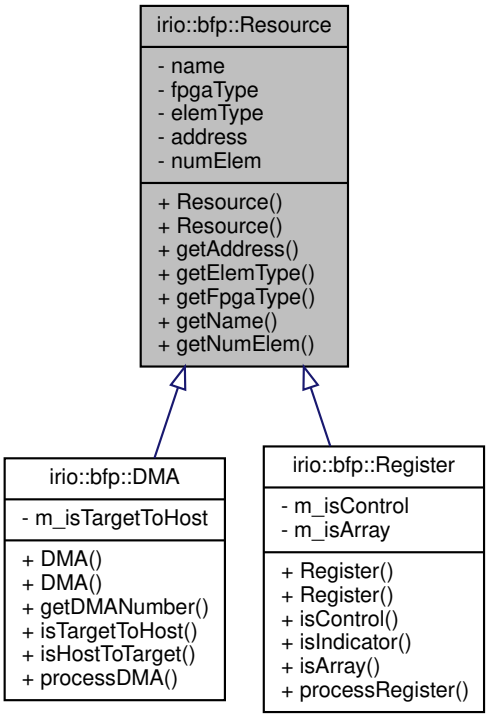

Figure 3.2: Inheritance Graph Resource

The resource class stores the name, the FPGA type (control or indicator), the element type, the address and the number of elements it has. The Resource and DMA classes' primary purpose is parsing these elements, as their structure in the XML differ. They also provide methods to get information specific to the Register and DMAs.

In the case of Registers, it should also be mentioned that they could be arrays; this is important as the low-level methods used to interact with arrays differ from those of non-array ones. The type of Register is determined by the FPGA type indicated in the bitfile; if the type corresponds to an array, a flag is set in the Resource class indicating it and the corresponding method, `isArray()`, returns it.

The DMA and Resource classes contain a static method responsible for parsing the XML and returning a newly created object. This was done as it was complicated to achieve by a constructor as the parent, Resource, required to be created before the XML could be parsed. The static method solves this, and by making it a static method of the class, it helps with organisation and maintainability as it is all contained in one class. The XML these methods receive is similar to the ones in Listing 3.1 and Listing 3.2.

```
<Register>
    <Name>SGFref1</Name>
    <Indicator>true</Indicator>
    <Datatype>
        <U32>
            <Name>SGFref1</Name>
        </U32>
    </Datatype>
    <Offset>4</Offset>
```

```
10      ...
11 </Register>
```

Listing 3.1: Section of Register XML

```
1 <Channel name="DMATtoHOST1">
2      <DataType>
3          <SubType>U64</SubType>
4      </DataType>
5      <Direction>TargetToHost</Direction>
6      <Number>0</Number>
7      <NumberOfElements>1023</NumberOfElements>
8      ...
9  </Channel>
```

Listing 3.2: Section of DMA XML

Up to this point, how the data regarding resources is extracted from the bitfile has been discussed. However, the mechanism by which this information is offered to the user has not been addressed. The BFP class (Figure 3.3) provides getters of relevant information in the bitfile, such as its signature, the unique identifier of that specific application, its version, and the registers and DMAs. While the signature and the version are straightforward, registers and DMAs are stored in a map at construction, where the key is the Resource's name and the contents. It is one of the previously mentioned classes, Resource or DMA.

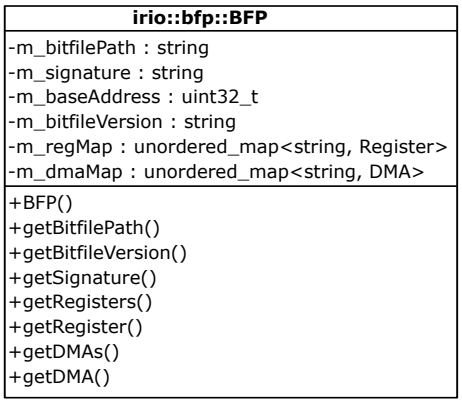

Figure 3.3: Class Diagram BFP Main Class

Error handling is managed through exceptions. In the BFP module, there are three sections on which an error may occur and subsequently an exception raised:

- **XML parsing**: Pugixml may report an error if it cannot parse the XML (file does not exist, XML is malformed, etc.), in this case a `BFPParseBitfileError` is thrown.
- **Getting a Register**: If the name of the specified Register does not exist, a `ResourceNotFoundError` is thrown.

- **Getting a DMA**: If the name of the specified DMA does not exist, a `ResourceNotFoundError` is thrown.

The BFP library does have certain limitations. Not all the types supported by FPGA LabVIEW are implemented; only the ones supported by the original IrioCore. However, it was designed to be extended if it is necessary to add new types.

3.4.3 Modules

Each RIO board has different parameters for analog acquisition and generation, changing their conversion values depending on their capabilities. In the case of cRIO and FlexRIO, these values could change depending on the module, or modules in the case of cRIO, connected.

Figure 3.4 shows the proposed solution for supporting these modules, a base class from which specialisations are created with the relevant values required, i.e.: Analog to Digital Converter (ADC) and Digital to Analog Converter (DAC) conversion values, and the maximum and minimum analog output values can be generated. It also allows these values to be obtained depending on the coupling mode configured supported.

One limitation of this approach is that adding each new module would necessitate the creation of a new class, potentially resulting in a slow process. The modules supported for FlexRIO are NI5761, NI6581, NI5734, and NI5781, while for cRIO, the supported modules are NI9205 and NI9264.

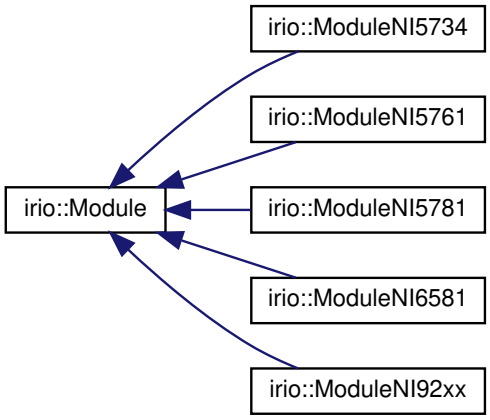

Figure 3.4: Modules Inherit Graph

3.4.4 Platforms

The current version of IrioCoreCpp supports three types of RIO boards: cRIO, FlexRIO and R Series. Depending on it, the number of available resources can differ and mainly, in the case of cRIO and FlexRIO, the maximum number of modules they can have varies, from 16 to 1, respectively. The information stored in a Platform is the maximum number of resources, separated by type, analog input, analog output, aux analog, digital, aux digital, DMAs and signal generators, along with the above-mentioned maximum number of modules supported and a Platform identifier.

Figure 3.5 represents the Platform classes for each of the supported RIO boards; each of the specialisations sets the maximum number of resources and modules as

constants of the parent class, making them read-only, which are then used by the IrioCoreCpp program when parsing the resources.

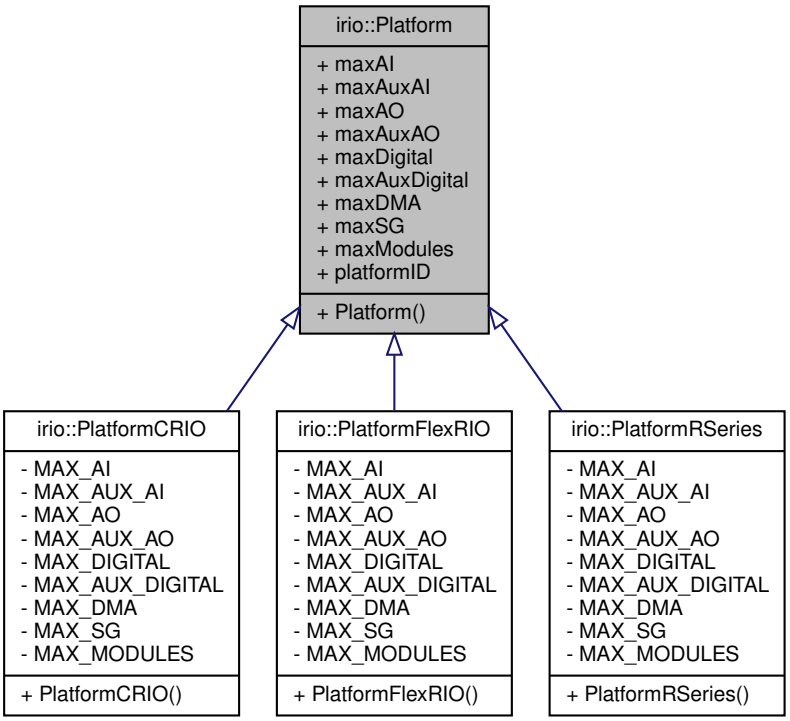

Figure 3.5: Platforms Inherit Graph

3.4.5 Terminals Groups

The IRIO Design Rules define several resources, which are divided according to different functionalities that the application can utilise. The IrioCoreCpp library employs a similar separation to group the resources into different classes called Terminals. The Terminals are responsible for extracting their required resources from the bitfile and checking their validity, a process illustrated in Figure 3.6. For example, they can check that all their corresponding resources are also available if two signal generators are defined. Additionally, the terminals offer a variety of methods for executing operations pertinent to their functionality.

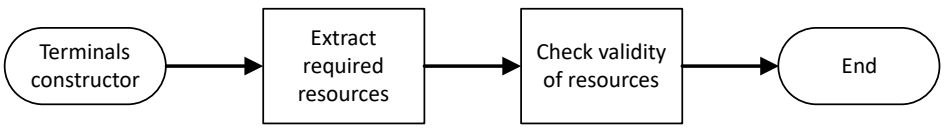

Figure 3.6: Terminals constructor process

How users interact with the FPGA is through the Terminals classes and their associated methods. However, returning the classes may necessitate copying them, which could present certain challenges as internal objects may not be able to be copied. One potential solution is to return a pointer to the corresponding Terminals class instead. Even if it is a smart pointer, this introduces additional complexity, as the user must now manage the IrioCoreCpp main class, which is a "normal"

object, and pointers, which can be confusing. To address this issue, it was decided to create Terminals Implementation classes, referred to as `TerminalsImpl`, which are the actual code interacting with the FPGA. These classes store the pointer to the implementation and call the implementation's methods. This approach ensures that the pointer is isolated from the user, allowing the class to be utilised as any other class. Furthermore, it permits the creation of multiple instances of the same Terminal with identical values. Figure 3.7 illustrates this process.

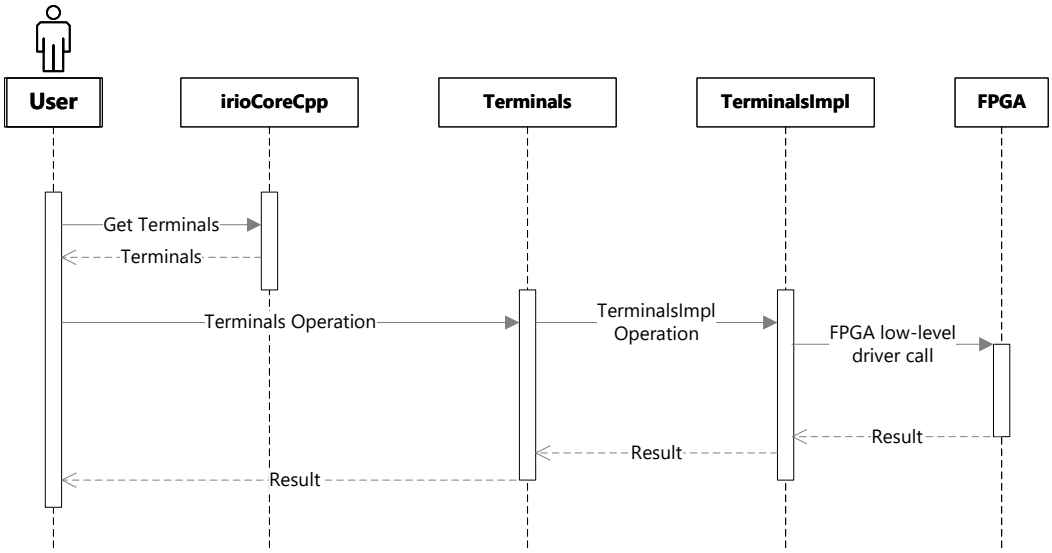

Figure 3.7: Terminals and TerminalsImpl usage

Figure 3.8 depicts the inheritance graph of the Terminals classes. All inherit from a base class, `TerminalsBase`, which stores the pointer to the equivalent `TerminalsImpl` class. This allows all the classes to be stored in the same type of container, which is used for retrieving Terminals when the user needs them (See subsection 3.4.7).

When comparing Figure 3.8 and Figure 3.9, while for most Terminals there is an equivalent `TerminalsImpl`, that is not the case for `TerminalsDMADAQCPU` and `TerminalsDMAIMAQCPU`, the reason is that these classes only call their parent's constructor, `TerminalsDMADAQ` and `TerminalsDMAIMAQ`, with specific resource names for CPU implementation of DAQ and IMAQ, and as such, do not require any methods to be implemented.

The following sections cover in more detail each Terminals class, covering which resources each is responsible for and checks performed to ensure that the defined resources are consistent with the different methods they provide to manage the FPGA application. While there is a distinction between Terminals and TerminalsImpl, as previously discussed, the following sections will focus on the details of the implementation. The Terminals classes methods are just a wrapper to a pointer to the equivalent implementation. For more information about Terminals and their methods, refer to the IrioCoreCpp Doxygen documentation, available online at `https://i2a2.github.io/irioCoreCpp/group__Terminals.html`.

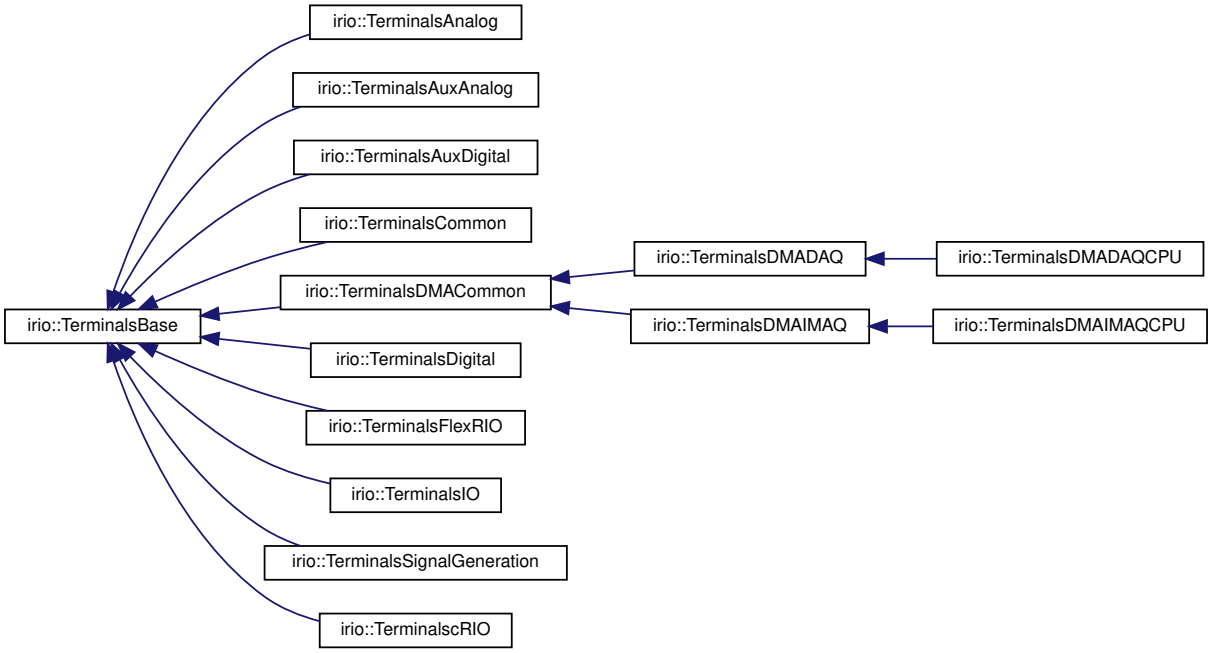

Figure 3.8: Terminals Inheritance Graph

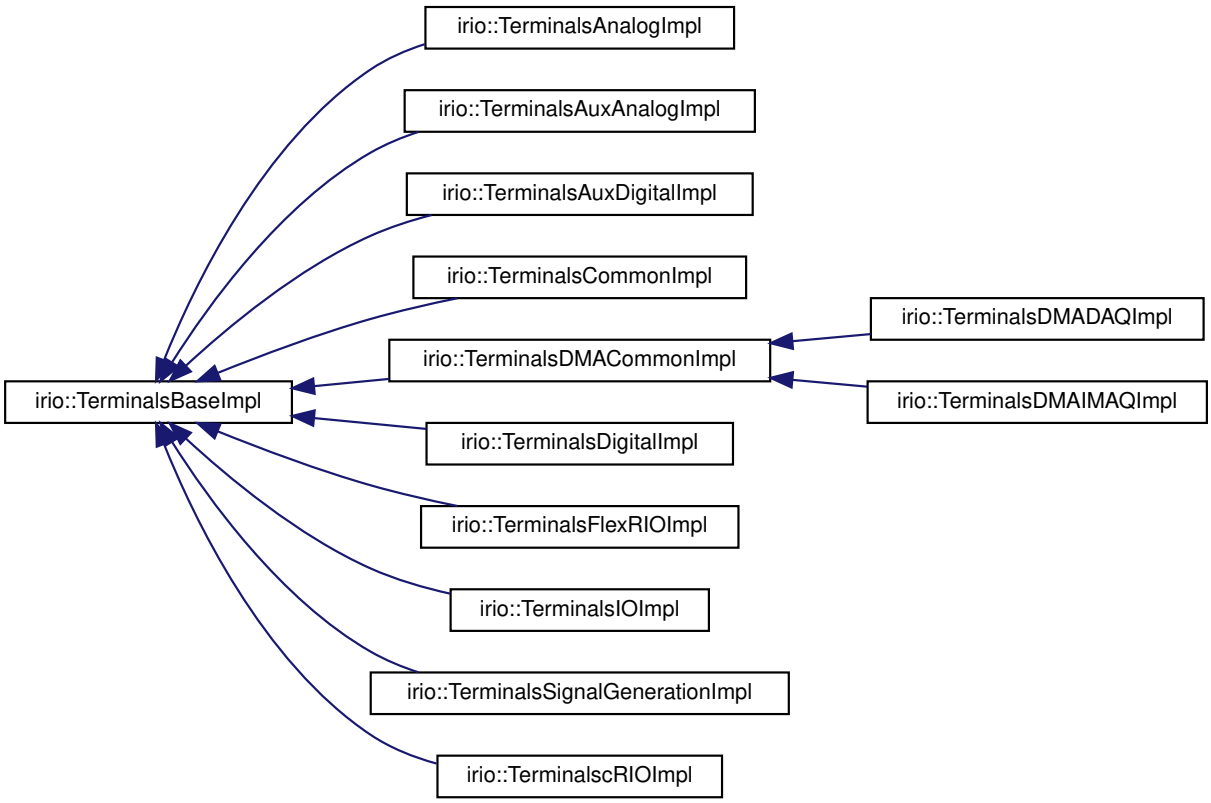

Figure 3.9: TerminalsImpl Inheritance Graph

3.4.5.1 TerminalsBaseImpl

`TerminalsBaseImpl` (Figure 3.10) contains all of the common variables that are common in all TerminalsImpl classes. It only includes the session ID for the FPGA, which is used in almost all interactions with the low-level driver.

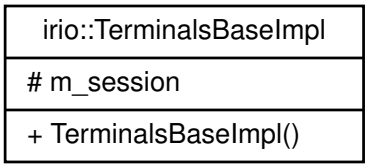

Figure 3.10: TerminalsBaseImpl Class Diagram

3.4.5.2 TerminalsCommonImpl

The `TerminalsCommonImpl` class (Figure 3.11) parses all the resources common to all functionalities and provides methods for managing them. It controls health monitoring in the FPGA such as the device temperature, the acquisition status and if its initialisation was successful. It also provides information on how the FPGA has been configured, like the reference frequency used for the FPGA application. It enables and disables acquisition, enables the debug mode if necessary during development, and provides information about the maximum and minimum sampling rate that can be configured.

Internally, it also checks that the reported FPGA version of the application matches the one specified during the IrioCoreCpp constructor.

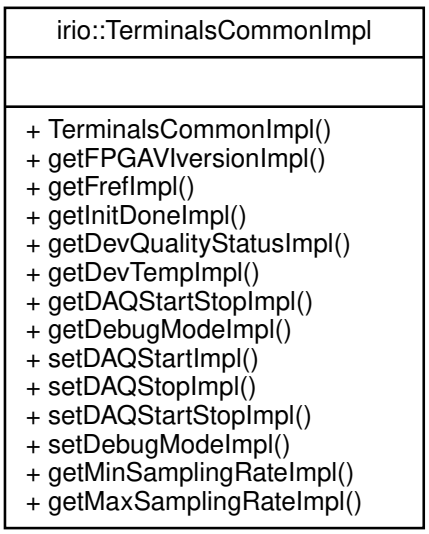

Figure 3.11: TerminalsCommonImpl Class Diagram

Table 3.1 presents a list of resources parsed by the `TerminalsCommonImpl` module, along with descriptions for each resource. All these resources are mandatory; the library will throw an exception if they are missing.

Resource	Description
FPGAVIVersion	Version of the LabVIEW application
Fref	Reference clock of the FPGA for the sampling rate
InitDone	Whether the FPGA board has finalize initizializing successfully
DevQualityStatus	Acquisition status
DevTemp	FPGA temperature
DAQStartStop	Control acquisition status. True to start.
DebugMode	Simulate the acquired data if true.

Table 3.1: Resources used by TerminalsCommonImpl

3.4.5.3 TerminalsAnalogImpl

The `TerminalsAnalogImpl` class (Figure 3.12) is responsible for acquiring and generating analog samples. This must not be confusing with other terminals that acquire or generate signals; the `TerminalsAnalogImpl` class only aims to acquire or generate analog samples for monitoring, triggering or controlling the FPGA application.

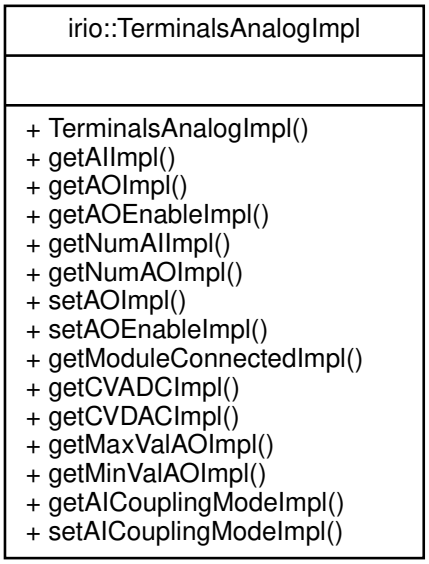

Figure 3.12: TerminalsAnalogImpl Class Diagram

Table 3.2 shows the resources parsed by the `TerminalsAnalogImpl` class. The `<n>` means that there can be more than one by replacing it with a number, e.g. there can be AI0 and AI1. The maximum number of analogue terminals is given by the platform on which the FPGA application is implemented; this information is stored in the Platform class (3.4.4 Platforms). As noted in the table, all these resources are optional, meaning that in any application, they can be between 0 and the maximum supported, and if they do not exist, no error will be thrown. However, in the case of analog output, it must be specified that there are two resources: `AO<n>`, which is the actual resource on which the sample is written, and `AOEnable<n>`, which is the

method by which the AO resource with the same <n> is enabled. This allows writing a value to AO<n> and not generating it until AOEnable<n> is set. Because of this, the TerminalsAnalogImpl class will check that if, given n as 5, A05, or A0Enable5 is present in the bitfile, the other must also be present; otherwise, it will generate an error.

Resource	Description	Required
AI<n>	Analog Input sample	Optional
AO<n>	Analog Output sample	Optional
AOEnable<n>	Enable/Disable AO<n>	Optional

Table 3.2: Resources used by TerminalsAnalogImpl

3.4.5.4 TerminalsAuxAnalogImpl

The TerminalsAuxAnalogImpl class (Figure 3.13) handles parsing and the read and write operation of the auxiliary analog resources. Auxiliary refers to the resources oriented to read or write internal variables in the FPGA.

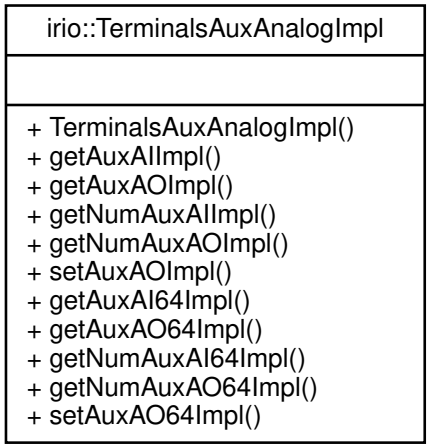

Figure 3.13: TerminalsAuxAnalogImpl Class Diagram

Table 3.3 shows the resources parsed by the TerminalsAuxAnalogImpl class. The <n> means that there can be more than one by replacing it with a number, for example, there can be auxAI0 and auxAI1. Implementing much more auxiliary analog resources than analog is possible due to not requiring real I/O hardware. There is also support for 8 bytes (64 bits) auxiliary analogue resources for larger FPGA internal variables to read or write. All the auxiliary resources are optional; they do not need to be defined in the FPGA application to be IRIO Design Rules compliant.

Resource	Description	Required
auxAIn	Auxiliary Analog Input internal FPGA variables	Optional
aux64AIn	Auxiliary Analog Input internal FPGA variables (64 bits)	Optional
auxAOn	Auxiliary Analog Output internal FPGA variables	Optional
aux64AOn	Auxiliary Analog Output internal FPGA variables (64 bits)	Optional

Table 3.3: Resources used by TerminalsAuxAnalogImpl

3.4.5.5 TerminalsDigitalImpl

The `TerminalsDigitalImpl` class (Figure 3.14) is responsible for acquiring and generating digital samples. It must be noted that this class is not oriented to generate digital signals; it is aimed to obtain or generate digital samples for monitoring, triggering or control of the FPGA application.

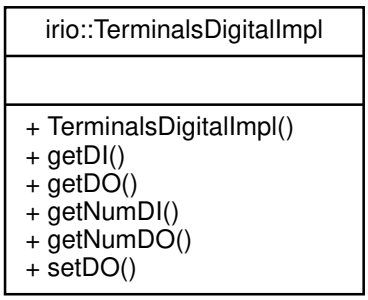

Figure 3.14: TerminalsDigitalImpl Class Diagram

Table 3.4 shows the resources parsed by the `TerminalsDigitalImpl` class. The `<n>` means that there can be more than one by replacing it with a number, for example, there can be `DI0` and `DI1`. The maximum number of analogue terminals is given by the platform on which the FPGA application is implemented; this information is stored in the Platform class (3.4.4 Platforms). All the digital resources are optional; they do not need to be defined in the FPGA application to be IRIO Design Rules compliant.

Resource	Description	Required
DI<n>	Digital Input sample	Optional
DO<n>	Digital Output sample	Optional

Table 3.4: Resources used by TerminalsDigitalImpl

3.4.5.6 TerminalsAuxDigitalImpl

The `TerminalsAuxDigitalImpl` class (Figure 3.15) handles parsing and the read and write operation of the auxiliary digital resources. Auxiliary refers to the resources oriented to read or write internal variables in the FPGA.

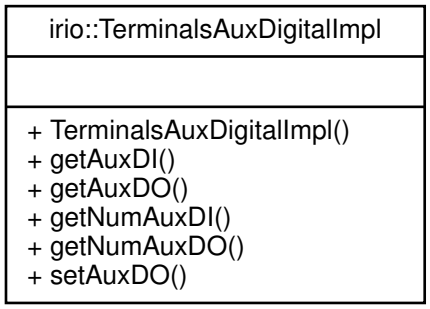

Figure 3.15: TerminalsAuxDigitalImpl Class Diagram

Table 3.5 shows the resources parsed by the `TerminalsAuxDigitalImpl` class. The `<n>` means that there can be more than one by replacing it with a number, for example, there can be `auxDO0` and `auxDO1`. Implementing more auxiliary digital resources is possible than digital due to not requiring real I/O hardware. All the auxiliary resources are optional; they do not need to be defined in the FPGA application to be compliant with the IRIO Design Rules.

Resource	Description	Required
auxDI<n>	Auxiliary Digital Input internal FPGA variables	Optional
auxDO<n>	Auxiliary Digital Input internal FPGA variables	Optional

Table 3.5: Resources used by TerminalsAuxDigitalImpl

3.4.5.7 TerminalsSignalGenerationImpl

The `TerminalsSignalGenerationImpl` class (Figure 3.16) is responsible for parsing the resources for signal generation and providing methods to configure this functionality.

Table 3.6 shows the resources parsed by the `TerminalsSignalGenerationImpl` class. The `<n>` means that there can be more than one by replacing it with a number, for example, there can be `SGSignalType0` and `SGSignalType1`. The number of signal generators implemented in the FPGA application is given by the value set in `SGNo`. All other signal generator resources must be the same number of times. No other signal generation Resource will be searched if `SGNo` is absent in the FPGA application.

In the case of the signal frequency to configure, the resource `SGUpdateRate<n>` is given in the decimation factor to apply. This decimation is applied to the reference frequency of the signal generator, `SGFref<n>`. The frequency to configure can be calculated as $freq_n = SGFref_n / SGUpdateRate_n$.

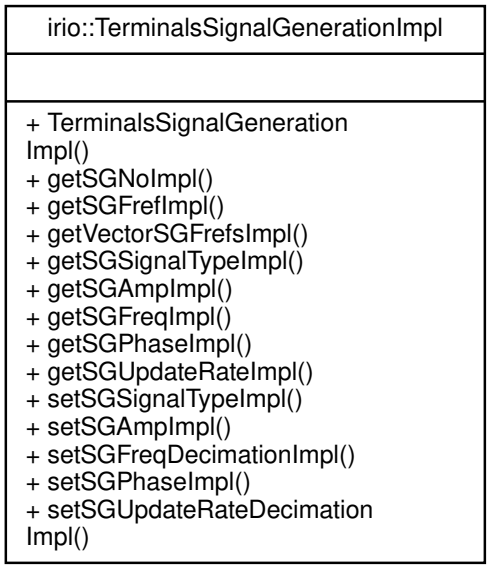

Figure 3.16: TerminalsSignalGenerationImpl Class Diagram

Resource	Description	Required
SGNo	Number of signal generators in application	Optional
SGSignalTypen	Signal shape to be generated	Optional
SGAmpn	Amplitude of signal to be generated	Optional
SGFreqn	Decimation factor of the frequency of the signal to be generated	Optional
SGPhasen	Phase of signal to be generated	Optional
SGUpdateRaten	Rate on which the output is updated	Optional
SGFrefn	Reference frequency of the signal generator	Optional

Table 3.6: Resources used by TerminalsSignalGenerationImpl

3.4.5.8 TerminalsIOImpl

The `TerminalsIOImpl` class (Figure 3.17) is responsible for parsing the resources for PBP and providing methods to configure this functionality.

Table 3.7 shows the resources parsed by the `TerminalsIOImpl` class. In this case, the only extra resource needed for the PBP functionality is configuring the sampling rate; this resource configures the decimation to be applied to `Fref`. The final sampling rate frequency can be calculated as $freq_{PBP} = Fref/SamplingRate_{PBP}$.

Resource	Description	Required
SamplingRate	Decimation to apply to the FPGA fref. Determines the sampling rate of the PBP operations	Optional

Table 3.7: Resources used by TerminalsIOImpl

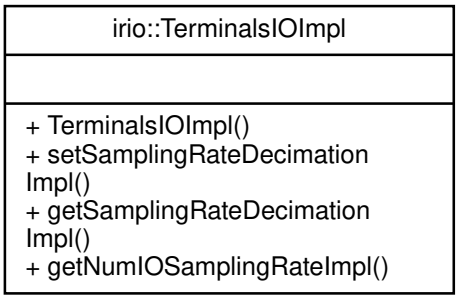

Figure 3.17: TerminalsIOImpl Class Diagram

3.4.5.9 TerminalsFlexRIOImpl

The `TerminalsFlexRIOImpl` class (Figure 3.18) is responsible for parsing the resources specific to FlexRIO boards and reading their values from the FPGA.

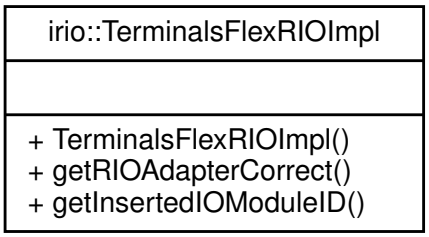

Figure 3.18: TerminalsFlexRIOImpl Class Diagram

Table 3.8 shows the resources specific to FlexRIO. They are related to the module attached and provide its ID. The other resource checks the inserted module is valid for the current FPGA application. Both resources are mandatory, and if any are not present, an exception will be thrown.

Resource	Description	Required
RIOAdapterCorrect	Indicates if the FlexRIO adapter the module is compatible with the current application	Mandatory
InsertedIOModuleID	ID of the module attached to the FlexRIO board	Mandatory

Table 3.8: Resources used by TerminalsFlexRIOImpl

3.4.5.10 TerminalscRIOImpl

The `TerminalscRIOImpl` class (Figure 3.19) is responsible for parsing the resources specific to cRIO boards and reading their values from the FPGA.

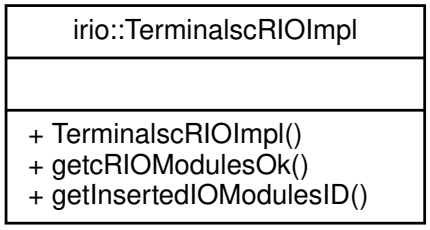

Figure 3.19: TerminalscRIOImpl Class Diagram

Table 3.9 shows the resources specific to cRIO. As cRIO can support multiple modules, the information about which ones are attached is given in an array, where each element represents a slot of the chassis and contains the ID of such module. A resource indicates whether all the modules needed for the FPGA application are present and ready. Both resources are mandatory, and if not available, an exception is thrown.

Resource	Description	Required
cRIOModulesOK	Indicates if all the cRIO The modules attached are the correct ones for the current application	Mandatory
InsertedIOModulesID	Array with the different IDs of the modules attached to the cRIO board	Mandatory

Table 3.9: Resources used by TerminalscRIOImpl

3.4.5.11 DMA Terminals

3.4.5.11.1 TerminalsDMACommonImpl

All the functionalities supported using DMA need specific resources. The TerminalsDMACommonImpl class (Figure 3.20) parses them and checks if any do not comply with the IRIO Design Rules. `TerminalsDMACommonImpl` also provides methods to read and write configuration parameters related to the acquisition. It also handles operations of managing the acquisition: cleaning the DMAs, starting/stopping the acquisition and reading the data acquired. For this last method, it provides a blocking and a non-blocking method. The blocking method also provides a timeout that if data is not ready before it, an exception is thrown.

Table 3.10 has the resources parsed by the `TerminalsDMACommonImpl` constructor. `DMATtoHOSTNCh` determines the total number of DMAs in the FPGA application. This number must match the number of `DMATtoHOST<n>` and `DMATtoHOSTEnable<n>` found. The `<n>` in these resources must also match to comply with the IRIO Design Rules. `DMATtoHOSTFrameType` is an array where each element corresponds to one DMA and indicates the frame type used. The frame type is how the data is organized in a data block; right now there are two types of frame types (Table 3.11): `Format A`

is just the data, no extra information, while `Format B` adds hardware timestamping, indicating the time of acquisition in the FPGA.

Resource	Description	Required
DMATtoHOSTNCh	Array, where each element is the number of channels in that DMA	Mandatory
DMATtoHOSTFrameType	Array where each element is the frame type used by the DMA	Mandatory
DMATtoHOSTSampleSize	Array where each element is the size in bytes of the DMA elements	Mandatory
DMATtoHOST<n>	FIFO memory with the data acquired	Mandatory
DMATtoHOSTEnable<n>	Enables or disables writing to the DMA FIFO	Mandatory
DMATtoHOSTOverflows	U16 elements where each bit represents if the equivalent DMA has overflowed	Mandatory

Table 3.10: Resources used by TerminalsDMACommonImpl for CPU Acquisition

Frame Type	Description
Format A	Each block of data contains only the data itself
Format B	At the beginning of the block of data, there is a timestamp indicating the time at which the data was obtained.

Table 3.11: Types of data format

Table 3.10 specified is for CPU acquisition; this is due to the possibility of implementing different types of acquisition, for example, GPU, where the data would be transferred to this device and processed. While the resources required would be the same, their names change; that is why it is specified that, in this case, the resource names are for the CPU. However, the current version of IrioCoreCpp only supports CPU acquisition.

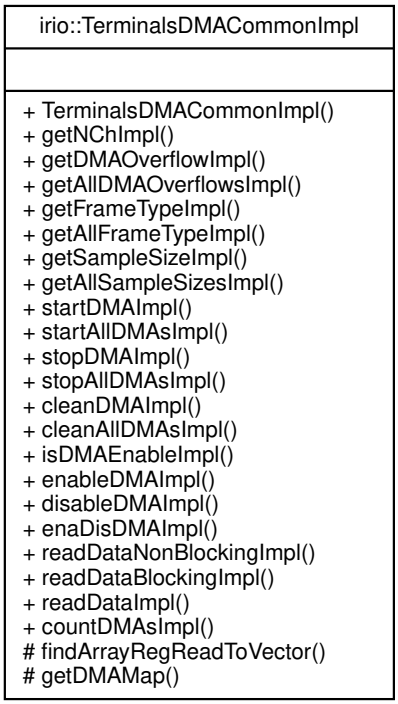

Figure 3.20: TerminalsDMACommonImpl Class Diagram

3.4.5.11.2 TerminalsDMADAQImpl

The `TerminalsDMADAQImpl` class (Figure 3.21) is responsible for parsing the resources related to DMA signal acquisition. It also provides methods to read and write configuration parameters.

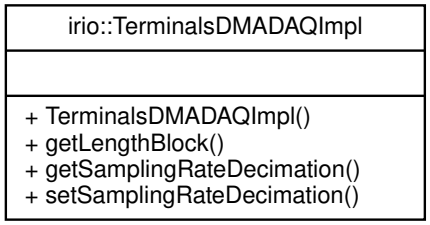

Figure 3.21: TerminalsDMADAQImpl Class Diagram

Table 3.12 shows the resources parsed by the `TerminalsDMADAQImpl` class. With `DMATtoHOSTSamplingRate<n>`, the decimation for each DMA sampling rate can be configured. The final sampling rate frequency can be calculated as $freq = Fref/DMATtoHOSTSamplingRate_n$.

It also checks that the number of `DMATtoHOSTSamplingRate` found matches with the DMAs found by the `TerminalsDMACommonImpl` class (3.4.5.11.1).

Resource	Description	Required
DMATtoHOSTBlockNWords	Array, where each element is the length of the data block for each DMA	Mandatory
DMATtoHOSTSamplingRate<n>	Decimation to configure for DMA sampling rate	Mandatory

Table 3.12: Resources used by TerminalsDMADAQImpl

3.4.5.11.3 TerminalsDMAIMAQImpl

The `TerminalsDMAIMAQImpl` class (Figure 3.22) parses the resources related to the IMAQ functionality. This Terminal also provides methods to configure the CameraLink (CL) device, send Universal Asynchronous Receiver-Transmitter (UART) messages to configure extra parameters and obtain the images acquired. It provides two methods to get the image data: blocking and non-blocking; the blocking option can be used with a timeout, and if used, an exception will be thrown if the image data has not been received in less than the specified time.

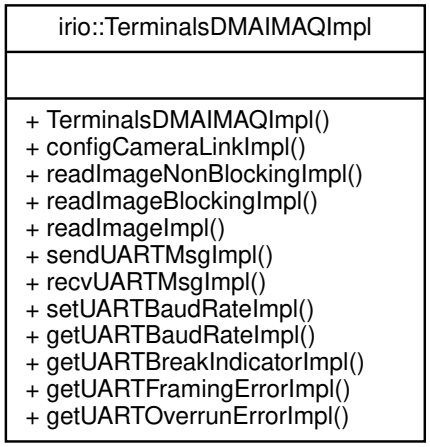

Figure 3.22: TerminalsDMAIMAQImpl Class Diagram

Table 3.13 shows the resources `TerminalsDMAIMAQImpl` needs to parse. They consist of one part on the configuration required for the CL device and the other regarding the UART communication. For most of these resources, there are no methods to interact with them directly; they are modified by more generic methods such as `configCameraLink` and for the case of the UART, `sendUARTMsg` and `recvUARTMsg`.

Resource	Description	Required
SignalMapping	Select the signal mapping for the CL interface	Mandatory
Configuration	Select CL interface type	Mandatory
LineScan	Set CL Line Scan	Mandatory
FVALHigh	Set CL FVAL Active High	Mandatory
LVALHigh	Set CL LVAL Active High	Mandatory
DVALHigh	Set CL DVAL Active High	Mandatory
SpareHigh	Set CL Spare Active High	Mandatory
ControlEnable	CL Control Enable	Mandatory
uartTransmit	Activate to transmit data	Mandatory
uartReceive	Activate to receive data	Mandatory
uartSetBaudRate	Activate set Baud Rate	Mandatory
uartBaudRate	Enumerated value with the BaudRate	Mandatory
uartByteMode	This Terminal allows configuring the mode for transmitting a collection of bytes. If true, the user can send an array of bytes.	Mandatory
uartTxByte	Byte to be transmitted	Mandatory
uartRxBytea	Data received	Mandatory
uartTxReady	Transmitter ready	Mandatory
uartRxReady	Receiver Ready	Mandatory
uartBreakIndicator	UART break indicator	Mandatory
uartFrammingError	Frame Error	Mandatory
uartOverrunError	Overrun Error	Mandatory

Table 3.13: Resources used by TerminalsDMAIMAQImpl

3.4.6 Profiles

Different combinations of Terminals will be required depending on the application functionality and the board used. Different Profile classes were created to manage this. They aim to instantiate the necessary Terminals and provide methods to interact with them.

The types of Profiles supported are the same as the original library, IrioCore, Data Acquisition (DAQ), Image Acquisition (IMAQ), and Point By Point (PBP). See section 2.3 for more details.

Figure 3.23 depicts the inheritance of the different available Profiles. The base class, `ProfileBase`, instantiates the Terminals common to all Profiles. Then, there are the Profiles implementing `ProfileBase` depending on the functionality, and finally, there is a last level of implementations depending on the board and functionality. These last Profiles are the ones that are instantiated in IrioCoreCpp.

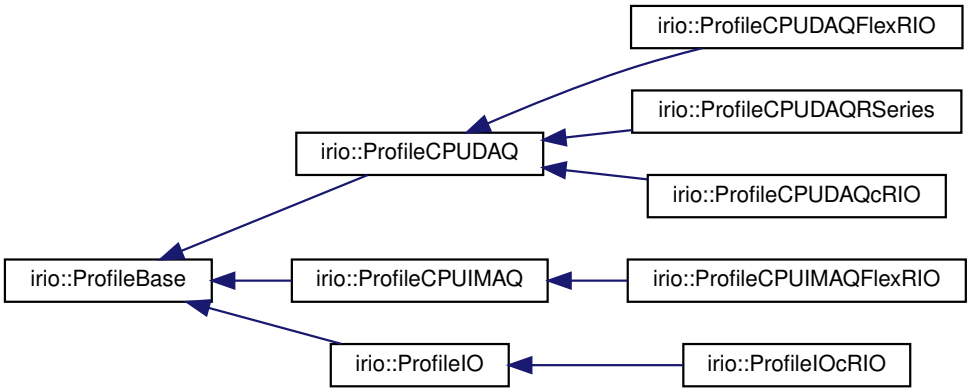

Figure 3.23: Profiles Inheritance Graph

Table 3.14 illustrates the Terminals associated with each Profile. Ultimately, the Profiles will instantiate these Terminals and those of the Profile they inherit from. There is a peculiar case with `ProfileCPUDAQRSeries`, as it does not add any new Terminals. This has been done to maintain order in the structure, where each board supported for functionality has its class. This also allows for future improvements to add extra Terminals specific to R Series, if necessary.

Profile Class	Inherits From	Terminals Added
ProfileBase		TerminalsCommon
ProfileCPUDAQ	ProfileBase	TerminalsAnalog
		TerminalsDigital
		TerminalsAuxAnalog
		TerminalsAuxDigital
		TerminalsSignalGeneration
		TerminalsDMADAQCPU
ProfileCPUIMAQ	ProfileBase	TerminalsDigital
		TerminalsAuxAnalog
		TerminalsAuxDigital
		TerminalsDMAIMAQCPU
ProfileIO	ProfileBase	TerminalsAnalog
		TerminalsDigital
		TerminalsAuxAnalog
		TerminalsAuxDigital
		TerminalsSignalGeneration
		TerminalsIO
ProfileCPUDAQFlexRIO	ProfileCPUDAQ	TerminalsFlexRIO

Continued on next page

Table 3.14: Terminals used by each Profile in IrioCoreCpp

Profile Class	Inherits From	Terminals Added
ProfileCPUDAQcRIO	ProfileCPUDAQ	TerminalscRIO
ProfileCPUDAQRSeries	ProfileCPUDAQ	
ProfileCPUIMAQFlexRIO	ProfileCPUIMAQ	TerminalsFlexRIO
ProfileIOcRIO	ProfileIO	TerminalscRIO

Table 3.14: Terminals used by each Profile in IrioCoreCpp

Figure 3.24 represents the base class of the Profiles. Two crucial methods need to be explained: `addTerminal` and `getTerminal`. The `addTerminal` method adds to a map any Terminal, as a pointer, used by any of the Profiles; this is possible due to all Terminals inheriting `TerminalsBase`, as the map's key the actual type of the Terminal is used. To retrieve them, templates are used; in this case, `getTerminal` is a template method where the template type is one of the Terminal types, and its pointer is then extracted from the map and converted back to its actual type. If the Terminal is not present for the given Profile and is not on the map, an exception for the `ResourceNotFound` type is thrown.

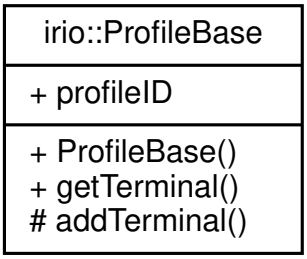

Figure 3.24: ProfileBase Class Diagram

3.4.7 Irio (IrioCoreCpp Main Class)

The Irio class (Figure 3.25) is IrioCoreCpp main class. It is the primary method users interact with the library; it manages one bitfile for a specific RIO device. By providing the bitfile, it parses it and detects the configured Platform and Profile, thus creating the necessary Terminals. It also provides a method to start the FPGA, starting the application loaded on it. For a user to load a new bitfile into the RIO device, it is necessary to create a new Irio object following the destruction of the previous one.

Most of the methods provided by Irio are Terminals getters. As covered in subsection 3.4.6, the associated Terminals will be created once a Profile is selected. The user must access such Terminals to manage the FPGA application. To do so, Irio provide getters for each possible type of Terminal; if the Terminal does not exist for the Profile selected, a `ResourceNotFoundError` exception is thrown.

Figure 3.26 depicts the steps taken when the Irio class is created. It first will try to search for the RIO device with the specified serial; this is done through the use of NI System Configuration library [46]. If the device is found, then it parses the bitfile using the BFP library. Afterwards, the `Platform` resource is read, indicating the type of the RIO board being used. After the Platform is selected, the Profile is determined; this takes into consideration the value of the `DevProfile` resource and

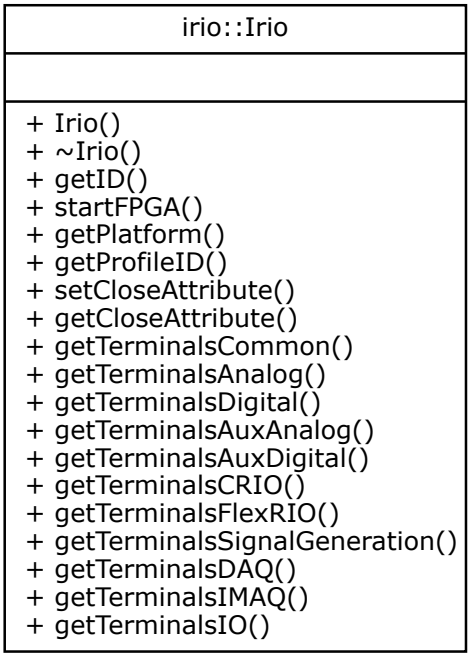

Figure 3.25: Irio class diagram

the RIO board, only accepting valid combinations (Table 2.1). Once the Profile has been created, a check is performed where the FPGA version reported by the FPGA application is matched with the one specified in the Irio constructor to avoid loading an incorrect version. Finally, if a resource is not found in the Terminals, instead of throwing an exception immediately in the Terminal, this error is logged and, at this point, is checked if any of these errors have occurred; if so, the found and not found resources are printed into a log file and an exception is thrown. If no error occurs, the Irio class is ready to be used.

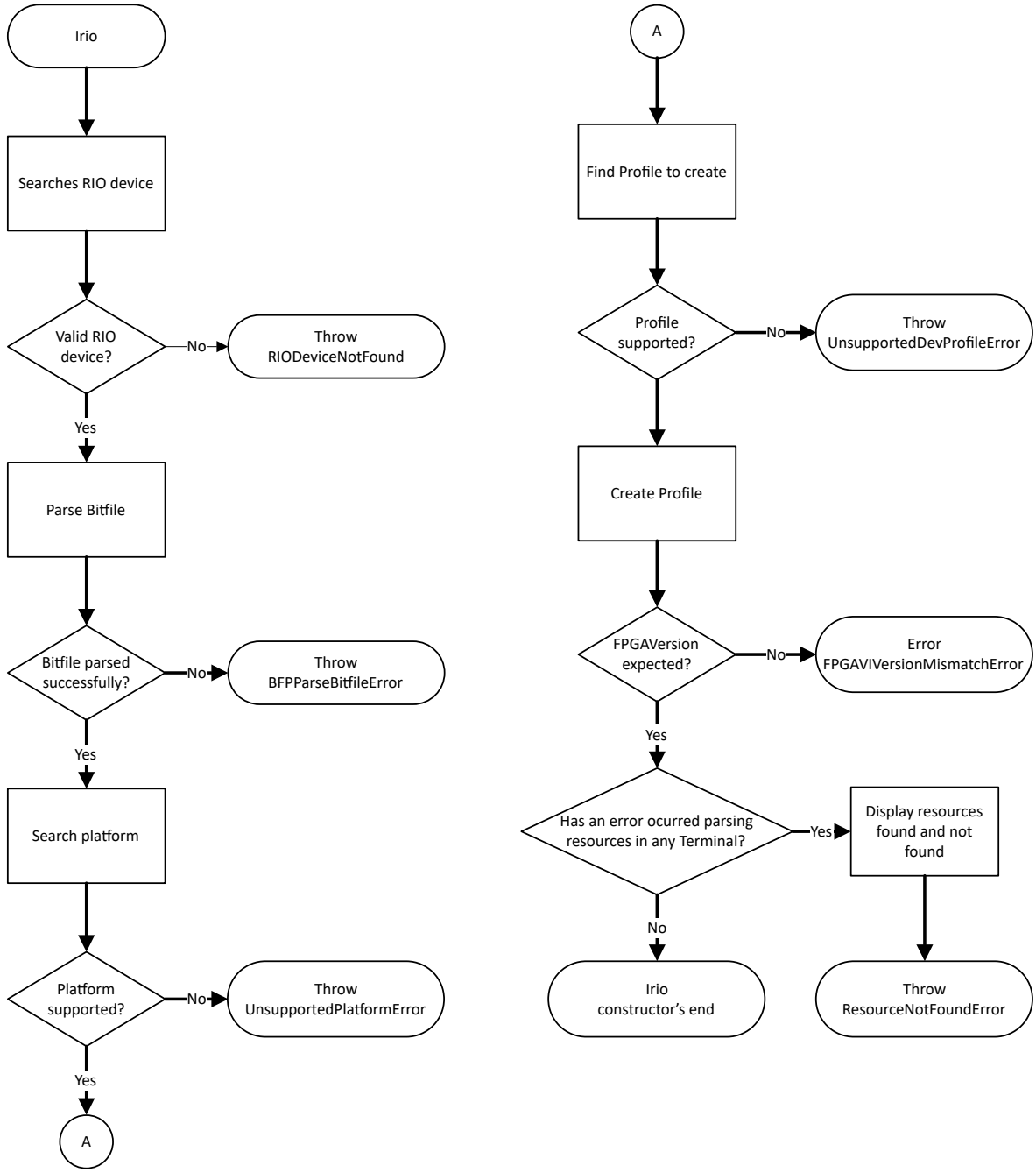

Figure 3.26: Irio constructor flowchart

3.5 Maintaining Compatibility With Previous IrioCore

Maintaining compatibility with existing applications was one of the main requisites for upgrading the previous IrioCore library. Given that the original IrioCore library was a C library, the solution involved creating a C wrapper for IrioCoreCpp, where a C interface is exposed to the user, but the underlying functionality is C++. It has been decided to keep the original name, IrioCore, for this wrapper as it allows applications already using it not to modify the libraries it links with. To differentiate between them, the previous version would be referred to as "the previous IrioCore", while the new one would be "the current IrioCore".

To maintain compatibility, the same API was used, preserving the function names and how their parameters are expected to be used and returning the result of the function call, indicating an error, if any, occurs. However, the ABI compatibility was not maintained. This was due to removing elements from the main structure used by the original IrioCore. They were removed for several reasons. Firstly, the elements were no longer required as their purpose was for internal operations of the previous IrioCore. Secondly, they allowed users to modify internal variables that could cause undefined behaviour if modified directly and not through a function. Nevertheless, not all variables could be removed, as some were used by applications whose maintenance is not scheduled yet and should remain compatible with the current IrioCore.

As a consequence of not being ABI compatible, all previously linked programs must now be recompiled. However, as the name of the library and the API have been retained, no additional changes are required to the code or the build procedures.

The method chosen for managing the creation and destruction of the Irio class (3.4.7 Irio) is through the functions `irio_initDriver` and `irio_closeDriver`. These functions are the first and last functions that should be called when using IrioCore, both in the previous and the current versions. `irio_initDriver` initialises the library and opens the session with the FPGA, while `irio_closeDriver` closes the sessions and cleans up. If any other functions of IrioCore are used without a valid session created with `irio_initDriver`, they will fail. This behaviour is used in the current IrioCore to, in `irio_initDriver`, create an Irio object with the specified bitfile and RIO device and return a session ID that is used in the rest of IrioCore functions. This ID is the key to a map where a pointer to the Irio object associated with that ID can be retrieved and operated on. In conclusion, it can be generalised that the Irio object lifecycle is bound to the `irio_initDriver` and `irio_closeDriver` functions.

3.6 Error handling

The IrioCoreCpp library leverages C++ exceptions for error handling. As illustrated in Figure 3.27 several custom exceptions were created, they all inherit from *IrioError* several custom exceptions were created, all of which inherit from IrioError, which in turn inherits from `std::runtime_error`. While these exceptions are primarily used for reporting failures, they are also employed to inform about anticipated behaviours the user may want to be notified of, such as timeouts. Table 3.15 provides an overview of each of the IrioCoreCpp exceptions.

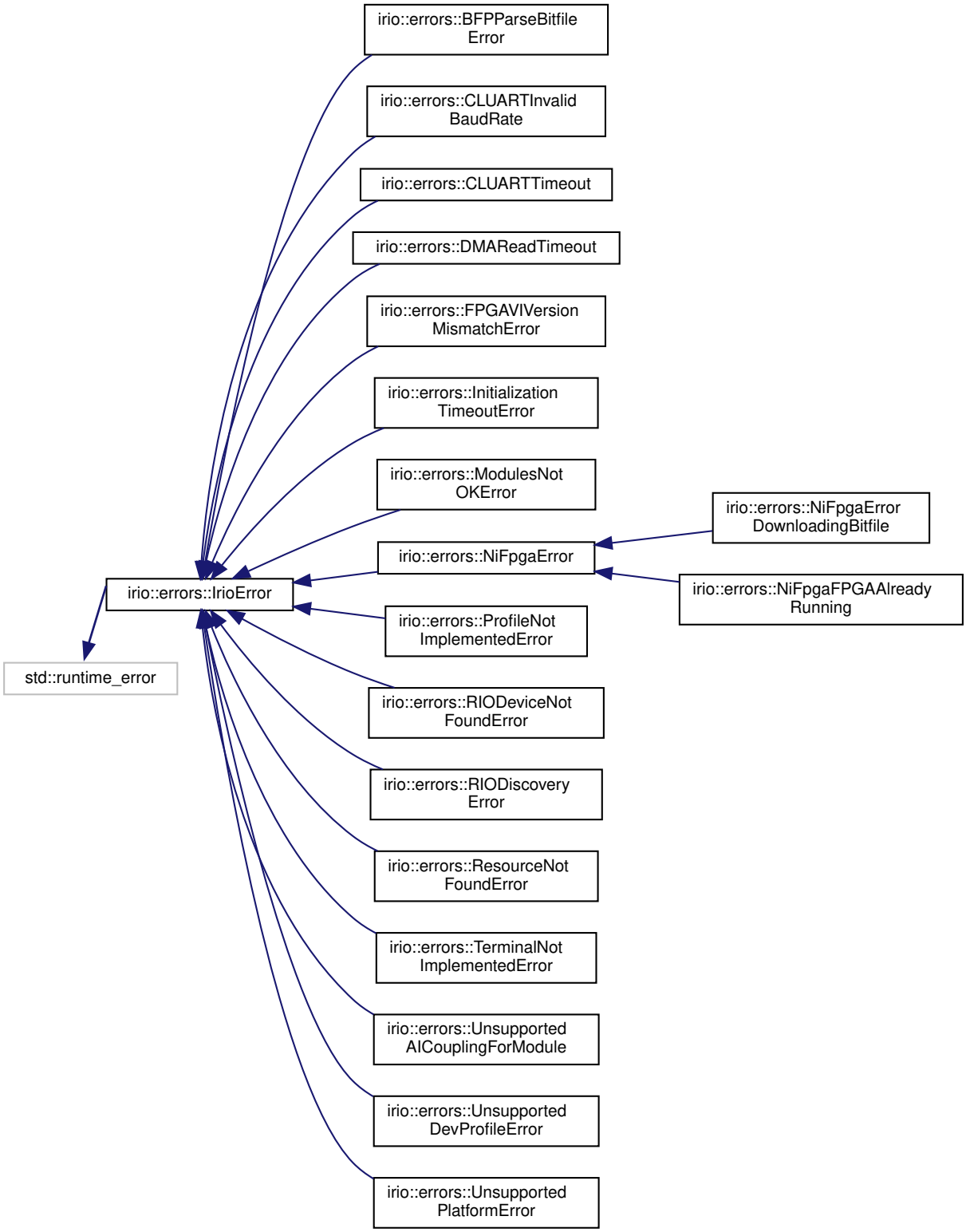

Figure 3.27: Inheritance diagram of IrioCoreCpp errors

Exception	Description
IrioError	Base Irio exception
TerminalNotImplementedError	Tried to get a Terminal not available in the current Profile.
ProfileNotImplementedError	The Profile requires has not been implemented yet
ResourceNotFoundError	A resource (Register or DMA) has not been found in the bitfile
FPGAVIVersionMismatchError	The FPGAVIVersion read from the FPGA and the one specified at construction does not match
UnsupportedPlatformError	The specified Platform read form the FPGA is not supported
UnsupportedDevProfileError	The specified Profile read form the FPGA is not supported
InitializationTimeoutError	The FPGA application took more time than allowed to initialize (The InitDone resource was not set)
CLUARTTimeout	Timeout in a UART operation in the CL device (Only applies to IMAQ Profiles)
CLUARTInvalidBaudRate	The BaudRate specified for the UART in the CL device is not a valid one (Only applies to IMAQ Profiles)
ModulesNotOKError	The modules attached to the RIO device are not the required for the FPGA application or an error related to them has occurred
RIODiscoveryError	An error has occurred while searching for the RIO device
RIODeviceNotFoundError	The specified serial does not match with any of the connected RIO devices
NiFpgaError	Error in the low-level driver (NiFpga library)
NiFpgaErrorDownloadingBitfile	Unable to download the bitfile to the FPGA
NiFpgaFPGAAlreadyRunning	Tried to start the FPGA application when it was already running
DMAReadTimeout	Timeout expires waiting for DMA data
BFPParseBitfileError	Error parsing the bitfile
UnsupportedAICouplingForModule	The specified coupling mode is not valid for the given module

Table 3.15: Description of IrioCoreCpp exceptions

For error handling in the IrioCoreCpp C wrapper, the current IrioCore, the return of status codes of the original remains, along with the Status structure. It has

been adapted through try-catch blocks where the exceptions from the Table 3.15 are caught, the appropriate error code and the message are set in the Status structure, and the code is returned.

3.7 Development Environment

3.7.1 Software

Table 3.16 covers the software used during development. Some of the resources covered mention CODAC, the development and interface environment provided at ITER [47]. This was required for IrioCoreCpp to demonstrate its backward compatibility with already existing projects using IrioCore, which were developed in the ITER environment.

Resource	Description
RedHat 8 with CODAC Core System 7.2	Current development and interface environment deployed at ITER
Rocky Linux 9	System based on RHEL, to simulate a similar environment to ITER without the CODAC component
EDT CameraLink simulator	CameraLink simulator software tools
RedHat 6 with CODAC Core System 4.2	Used to execute the CameraLink simulator
Windows 11	Main development machine
WSL	Window Subsystem for Linux (WSL). Ubuntu and Rocky Linux. Used while development to test if the libraries compile for these distributions and run unit tests
Visual Studio Code	IDE used for development

Table 3.16: Software Environment Used

3.7.2 Hardware

Figure 3.28 shows the main hardware used during development. It consists of 3 FlexRIO boards, the respective modules to test DAQ and IMAQ, the chassis and the interface with the host system. Table 3.17 covers in more detail the hardware used during the development and testing of the project.

Figure 3.28: Chassis with the FlexRIO boards used during development

Resource	Description
FlexRIO PXIe-7966R (x2)	FlexRIO boards used for DAQ and IMAQ
FlexRIO PXIe-7965R	FlexRIO board to test digital acquisition
FlexRIO Module NI5781	Used to test DAQ functionality. Connected to one of the FlexRIO 7966R boards
FlexRIO Module NI6581	Used to digital acquisition functionality. Connected to the FlexRIO 7965R board
FlexRIO Module NI1483	CL module. Used to test IMAQ functionality. Connected to one of the FlexRIO 7966R boards
NI PXIe-1062Q	Chassis that houses the FlexRIO boards
NI PXIe-8370	Allows to control the FlexRIO boards using an MXI-Express link. Connected to chassis
NI PCIe-8371	Allows to connect the chassis to the host system through an MXI-Express link

Continued on next page

Table 3.17: Hardware Environment Used

NI SMB-2163	Terminal block used for testing digital acquisition. Connected to the FlexRIO Module NI 6581
EDT PCIe DVa C-Link	PCIe card connected to system with CL simulator
Generic Desktop PC	Generic PC used for CI simulator
Generic Desktop PC	Generic PC used for development
Industrial PC	Industrial PC used as host for the RIO boards

Table 3.17: Hardware Environment Used

Chapter 4

Proposed Software Engineering Process

4.1 Software Architecture

Figure 4.1 offer a higher level view of how the different elements that build IrioCoreCpp interact. Each interaction is labelled with a number; the ones with only a number are operations started by the user application, while the ones with the same number and a letter are the sequence of events that occur. The different operations could throw exceptions; more details can be found in section 3.6. These operations are explained below:

1. Application creates an Irio object. The application specifies which bitfile and the RIO Serial to use. An exception will be thrown if the bitfile is not found or invalid for the specific folder. Not finding a RIO device with the given RIO Serial will also throw an exception.

   1A The bitfile is parsed using BFP. It searches for the resources (Registers and DMAs). If the bitfile is invalid, it will throw an exception.

   1B The resources found are parsed and used to create the appropriate Profiles and Terminals. Finalizing the creation of the Irio object.

2. The application, using the Irio object, realizes FPGA management operations; in this case, it is mainly the start FPGA operation.

   2A Calls the low-level driver to start the FPGA.

   2B Check that the low-level call was successful, that the FPGA application reports it has been initialized successfully, and that the modules if required, are connected and ready. If anything fails, an exception is thrown.

3. To obtain a Terminal, the application would call the Irio method `getTerminals<Name of the Terminals>()`, for example, `getTerminalsAnalog()`.

   3A The Irio object will call the Profile method to get the Terminals requested

   3B If the Terminals exist for the current Profile, they will be returned. If not, an exception is thrown.

4. Using the Terminals object obtained in the previous part. The application can use any of the available methods to read/write in the FPGA resources, which will control the behaviour of the FPGA application.

   4A The Terminal will check the validity of the parameter, if any. And will call the low-level driver to interact with the FPGA. If the parameter is invalid, an exception will be thrown.

   4B The low-level driver call will return if the call was successful. And the value read if a read operation is performed.

   4C If an error has occurred in the low-level call, an exception will be thrown. If not, if necessary, the value read will be returned.

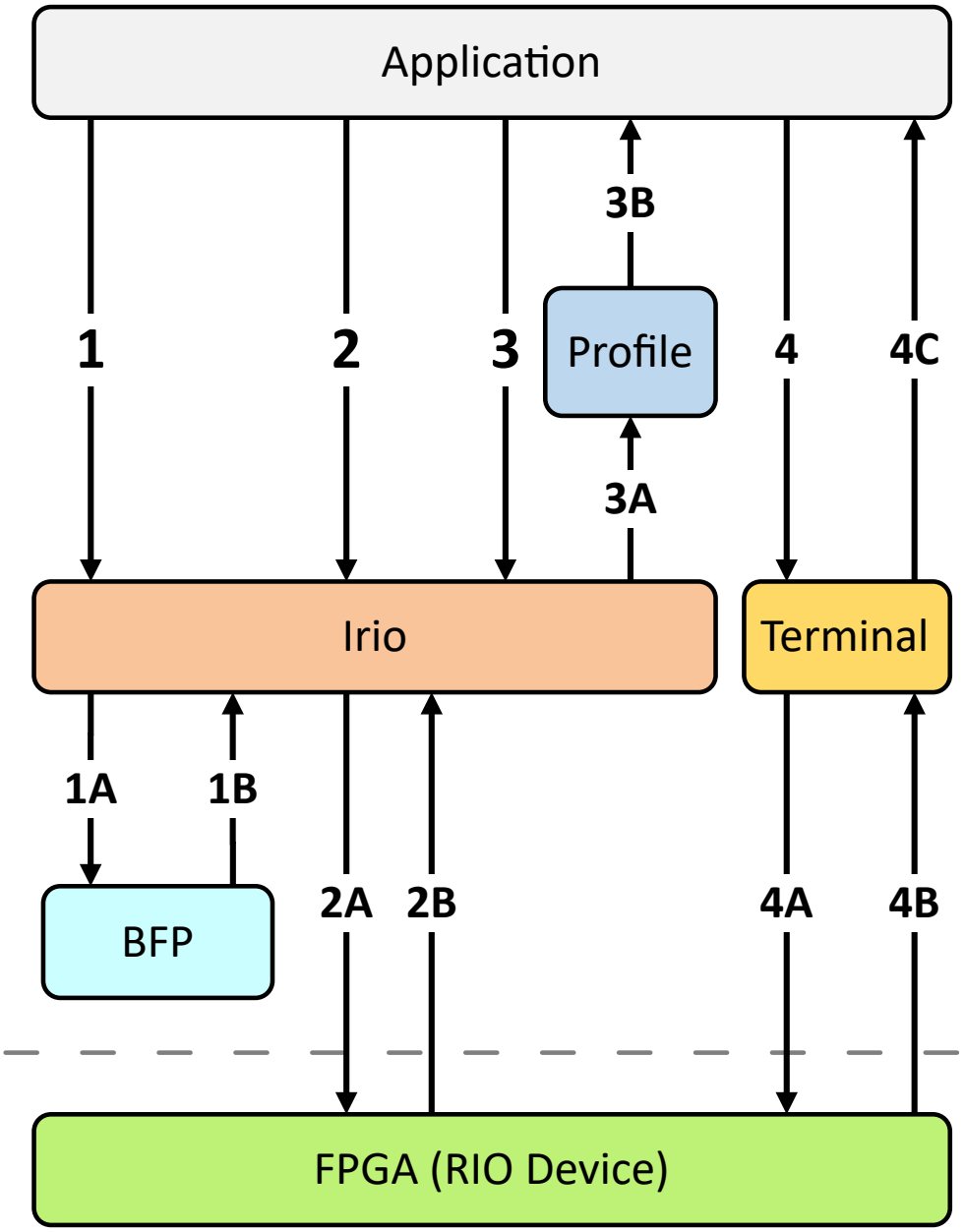

Figure 4.1: Software Architecture IrioCoreCpp

4.2 DevOps Processes

4.2.1 Environments

In software development, an environment is defined as a set of rules and configurations where the application is developed, tested, and deployed. In the case of IrioCoreCpp, using real hardware boards introduces some complications to the development process. From a cost and infrastructure perspective, it is not feasible or practical for each developer to have access to these devices at all times. The cost of the devices and their associated chassis and connections is considerable, and it is not feasible to constantly move them to another developer system.

For this software project, IrioCoreCpp, two environments have been defined: a development environment and a staging environment.

4.2.1.1 Development environment

The development environment is where the initial coding and testing take place. In the case of IrioCoreCpp, a series of procedures have been created to automate and simplify them. This includes building the application, debugging and testing, and generating coverage, documentation, and packages for installation within the system. This allows the developer to facilitate being part of the CI process, where new features can be added and bugs can be fixed while maintaining the quality of the code.

4.2.1.2 Staging environment

The staging environment serves as a means of validating changes to the code, both in terms of quality and to ensure that no unintended consequences have been introduced during the process. This environment is also responsible for generating the packages used when releasing a new version of IrioCoreCpp. It ensures that all necessary checks are completed successfully before generating the release. In this case, the staging environment for IrioCoreCpp occurs in GitHub. Here, actions are triggered in response to specific events.

4.2.2 Tools used

4.2.2.1 Make

The Make utility was chosen to automate most stages in the DevOps process. This was done to maintain as much structural similarity as possible to the C/C++ project template employed by ITER. The objective was to make a system as much as possible compatible with ITER development process. This process involves using Maven, a software management tool, along with specific plugins that execute different goals, ultimately using the Make utility to compile the C/C++ code. Then, a question arises: why not use Maven? This was due to the plugins mentioned above, as most are not freely available outside of ITER and would make distributing IrioCoreCpp and allowing others to generate a working version harder.

Although maintaining compatibility with ITER was one of the main factors of using Make for managing the different DevOps stages, it was also chosen due to several other factors, such as familiarity with the tool, the availability of support

and documentation, the relatively simple setup, its reliability and is a commonly available tool in most platforms.

The way the different DevOps operations have been integrated using Make is through a main Makefile that controls the possible workflows and their dependencies. This main Makefile executes other Makefiles, which are the specific implementations of the workflows, which helps with easier maintainability as it is more organised. Some of these recipes also support specifying parameters which change their behaviours, giving more flexibility to the developer. The recipes implemented using Makefiles are covered in subsubsection 4.2.3.1.

4.2.2.2 Docker

While setting all the required dependencies for a program is feasible for the developer, it may not be for CI/CD operations, as the machine running the workflows may be on the cloud or shared between multiple projects. To have all the dependencies together, be maintainable, and avoid polluting systems with various installations and incompatibilities between module versions, one solution is to use Docker.

In the case of IrioCoreCpp, most of the CI/CD operations are run in hosted servers. Installing the dependencies to compile and generate the packages can be time-consuming, which can also mean economic costs in the case of hosted servers. For this purpose, a Docker image with IrioCoreCpp's 'dependencies was created, and the CI/CD system can create a container with it to run the required workflows.

The original IrioCore was developed in an RHEL OS. Therefore, it was decided to use an OS based on it as an image base. Rocky Linux was chosen as it is currently actively supported, and Docker images are provided, specifically Rocky Linux 9. It was decided to use a minimal image as most software is not required and to keep the image size down.

Table 4.1 depicts the various packages installed in the created Docker image. The aim has been to minimise the size of the image, only installing the minimum necessary. This was done because the image is downloaded when the various workflows are executed. Therefore, it is not efficient to download an image that includes large packages that may only be used by a single workflow.

Group	Packages installed
Package manager	dnf
Build packages	gcc
	gcc-c++
	make
	python
	python-pip
IrioCoreCpp dependencies	rsync
	pugixml-devel
	gtst-devel
	ni-syscfg-devel
	ni-flexrio-modulario-libs-devel
	cpplint
Coverage	lcov
	lcov-cobertura
Packaging	alien
Misc	unzip

Table 4.1: Packages installed in Docker image

4.2.2.3 GitHub

To facilitate version control versioning and encourage collaboration, it was decided to use a remote DVCS, specifically GitHub. This was chosen due to its widespread popularity and adoption within the developer community, as well as some of the advantages it offers for public projects, such as usage free of remote hosts for CI/CD pipelines, called GitHub Actions.

4.2.2.3.1 GitHub Actions

GitHub Actions is a CI/CD platform from GitHub that allows automating various tasks, such as testing, compiling, and deploying code, all triggered by events within the repository, such as push and pull requests. Thanks to the popularity of GitHub, one great advantage is that it provides a marketplace of pre-built actions that can be used to create more complex ones. Also, GitHub Actions are written using a relatively simple syntax using YAML.

GitHub actions created multiple workflows in the CI/CD pipeline for the staging environment that allows compiling, testing and releasing IrioCoreCpp. These workflows have been covered in more detail in subsubsection 4.2.3.2.

4.2.3 Pipelines

Two pipelines were created, one for each of the environments defined (subsection 4.2.1). Each pipeline defines multiple stages, although some of them are very similar. The

following sections provide a more detailed account of each pipeline and its stages.

4.2.3.1 Development environment pipeline

The different stages of the development environment pipeline are mainly designed to be run at the developer's discretion. They are implemented using Makefiles, one for each stage and a main Makefile that controls which stages are run and manages their dependencies on other stages, Figure 4.2 illustrates these dependencies. Some of the stages also allow specifying parameters that modify their behaviour. This made them more flexible, which may be desirable in a development environment.

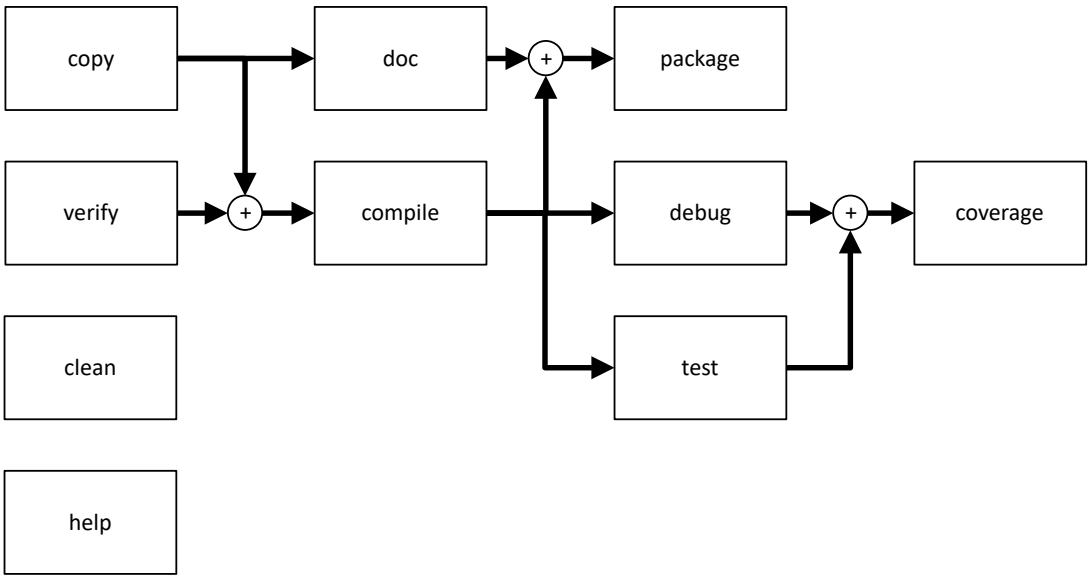

Figure 4.2: Development environment workflows dependencies

While the subsequent sections will cover in more detail the available stages of the pipeline, Table 4.2 and Table 4.3 summarise them along with the available parameters for the stages, respectively.

Stage	Description
help	Display a message with the available recipes and their parameters
copy	Copy the `src/` folder into `target/`, creating it if necessary. This allows the separation of build artifacts from the source code.
clean	Clean the project. Removes the `target/` folder and its content
verify	Enforce Google's C++ style guidelines and file name and directory structure
compile	Compile the project's source code
debug	Compile the project's code with debugging symbols. Equivalent to `make compile Debug=1`

Continued on next page

Table 4.2: Development Pipeline Stages

test	Run tests. Unit tests are always run independently of if extra tests have been selected.
coverage	Generate the project's code coverage. Line and function coverage. It also checks that the result is above 90% in both
doc	Generate Doxygen documentation
package	Generate the project's RPMs

Table 4.2: Development Pipeline Stages

Stage	Parameters	Behaviour
verify	SkipVerify	If set, the lining and folder and naming checks are skipped.
compile	Debug	If set, the code is compiled with debugging symbols and without optimisations
test	SkipTests	If set, all tests are skipped
	AddTestsFunctionalIrioCore	Adds functional IrioCore tests
	AddTestsFunctionalIrioCoreCpp	Adds functional IrioCoreCpp tests
package	INSTALL_DIR	This optional variable sets the install location. Libraries will be placed in `INSTALL_DIR/lib` and headers in `INSTALL_DIR/include`. If `INSTALL_DIR` is not set, it defaults to `/usr/local`, unless `CODAC_ROOT` is set, in which case `INSTALL_DIR` will use the value of `CODAC_ROOT`.

Table 4.3: Development Pipeline Stages Parameters

4.2.3.1.1 help

A simple stage that displays information about the different stages that can be used and their parameters. Its purpose is to allow developers to know the available recipes without having to consult the documentation or analyse the Makefile, making it easier for them to execute the different stages and, as a result, increasing the probability that the different stages will be executed and the quality and maintainability of the code is evaluated.

It does not have any other stage as a dependency or a dependency of another stage.

4.2.3.1.2 copy

It was decided to separate the source code from the build artefacts and other files generated when testing. For this purpose, before compiling the code or the documentation, the complete folder with all the projects' code, including the test, called `src`, is copied to a new folder, `target`.

It does not have any other stage as a dependency but is a dependency of most of the different stages as it is a base for it. The only ones not needing the `target` folder are the help, verify, and clean stages.

4.2.3.1.3 clean

Cleans the project. It removes the `target/` folder and its content, cleaning all the build artefacts and other files generated. If the `target` folder is not present, nothing is done.

It does not have any other stage as a dependency or a dependency of another stage.

4.2.3.1.4 verify

Figure 4.3 illustrates the verify stage; it consists of two processes, one where linting is applied to the code and another that checks the folder structure of the project and if some of the files follow a naming convention.

The lining enforces Google's C++ style guidelines, file name, and directory structure. It statically checks the source code and applies Google's C++ style guidelines to it, not succeeding if any is present. To do so, the tool cpplint was developed and maintained by Google initially; however, nowadays, it is not, so the tool used for this project is, in reality, a fork of it [21].

The folder structure and naming check mainly that only headers are in a specific folder and that the headers and source files naming convention start and end in a particular way. The naming convention is straightforward in the case of the current IrioCore as it was decided to maintain the structure and naming convention of the previous one.

This stage has one possible parameter that can alter its behaviour:

- SkipVerify: The linting, structure and naming convention checks are skipped if set. This is useful for other stages that execute verify, saving time.

It does not depend on any other stage but on most of them, except for help, clean, copy, and doc.

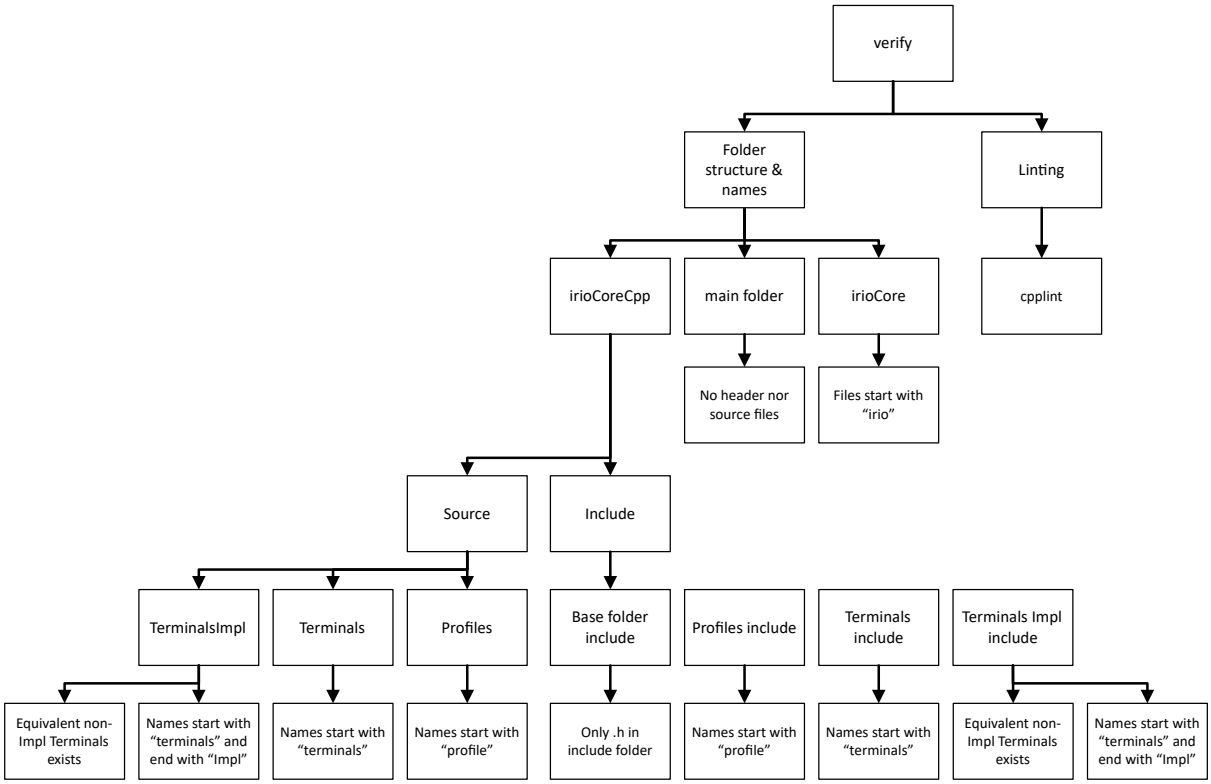

Figure 4.3: Development pipeline verify stage

4.2.3.1.5 compile

It executes the Makefiles in the following folders:

- `target/main/c++`: Folder containing the IrioCoreCpp, IrioCore and BFP libraries.
- `target/test/c++`: Folder containing the unit and functional tests.

This stage has one possible parameter that can alter its behaviour:

- Debug: If set, the code is compiled with debugging symbols and without optimisations. This allows code debugging and creating coverage files.

It depends on the verify and copy stages. It is dependent on the tests and package stages.

4.2.3.1.6 debug

Compiles the project's code with debugging symbols and without optimisations. It also enables coverage flags. Executing the compile stage with `Debug=1` is the same as executing the compile stage. As a developer, it is simpler to have a dedicated debug stage that remembers the parameters needed for compiling with symbols.

The debug depends on the compile stage, which also depends on the coverage stage, as it enables the flags to generate coverage files.

4.2.3.1.7 test

Run the tests. It uses a Python script and predefined XML files to execute the tests and log the results; this is covered in more detail in section 4.3.

By default, if no other parameter is defined, it only executes the unit tests for BFP, IrioCore and IrioCoreCpp. Extra parameters can be added to execute the functional test of IrioCore, IrioCoreCpp or both. However, even when functional tests are added, unit tests for all the libraries in the project will still be run.

This stage has three possible parameters that can alter its behaviour:

- SkipTests: If set, all tests are skipped, including unit tests.
- AddTestsFunctionalIrioCore: Adds functional IrioCore tests.
- AddTestsFunctionalIrioCoreCpp: Adds functional IrioCoreCpp tests.

The test stage depends on the compile stage. It depends on the coverage stage, as the test stage needs to be run to know the lines and functions covered.

4.2.3.1.8 coverage

Generates code coverage for BFP, IrioCore and IrioCoreCpp libraries. The tool used to generate the coverage is lcov, the results of which are then converted to HTML for subsequent analysis if required. This stage also asserts the total coverage in both lines and function is above 90%.

The coverage stage depends on the debug, and the test stage, and no other stage depends on it.

4.2.3.1.9 doc

Generates the Doxygen documentation. It uses the Doxyfile in `src/main/doc` as the input file. It generates both HTML documentation and LaTeX files that could later be compiled into a PDF.

The doc stage depends on the copy and package stages, as one of the packages generated is the HTML documentation.

4.2.3.1.10 package

Generates the projects's packages. As the main target system is based on RHEL, these are RPM files. The packages generated are:

- bfp: BFP's shared library
- bfp_devel: BFP's static library and headers. They are only needed if developing a project that uses them.
- irioCore: IrioCore shared library.
- irioCore_devel: IrioCore static library and headers. They are only needed if developing a project that uses them.
- irioCoreCpp: IrioCoreCpp shared library.

- irioCoreCpp_devel: IrioCoreCpp static library and headers. They are only needed if developing a project that uses them.
- irioCore_doc: HTML documentation for IrioCoreCpp, IrioCore and BFP libraries

The installation directory can be changed using the `INSTALL_DIR` parameter, libraries will be placed in `INSTALL_DIR/lib` and headers in `INSTALL_DIR/include`. If `INSTALL_DIR` is not set, it defaults to `/usr/local`, unless `CODAC_ROOT` is set, in which case `INSTALL_DIR` will use the value of `CODAC_ROOT`.

4.2.3.2 Staging environment pipeline

While the environment pipeline was thought to be started by each developer, the staging pipeline is thought to be executed every time there are specific changes to the project. For this reason, this pipeline is executed where the changes the developers made would finally converge in the remote repository, which is GitHub. Using GitHub actions, several stages that create the staging pipeline have been created; a summary of them can be seen in Table 4.4. Table 4.5 summarises what events in the repository trigger the different stages. Subsequent sections cover them in more detail.

Stage	Description
verify	Enforces Google's C++ style guidelines and file name and directory structure
compile	Compile the project's source code
test	Run unittests for BFP, IrioCore, IrioCoreCpp
sonar	Uses the cloud-based static code analysis solution SonarCloud to analyse the code and measure its quality and security. Part of these is generating coverage from unittest and assuring it is greater or equal to a threshold (90%).
doc	Generates Doxygen documentation
package	Creates the project packages in the form of RPM and DEB packages.
release	Generates a release in the repository and attaches the generates packages for that release. It also publishes the documentation to the repository GitHub Pages

Table 4.4: Available Staging Stages

	manual	on call	push	pull request	tag created
verify	X	X	(all branches)	(all branches)	
compile	X	X	(main)	(main)	
test	X	X	(main)	(main)	
sonar	X	X	(main)	(main)	
doc	X	X	(main)	(main)	
package	X	X			
release					X

Table 4.5: Staging pipeline stages trigger events

4.2.3.2.1 verify

This stage is conceptually the same as the one defined in the development pipeline (4.2.3.1.4 verify), using a linter in the source code, checking the file structure and ensuring the files follow various naming conventions.

The only difference is that the verify stage of the staging pipeline is run every time there is a push or pull request to any branch. It can also be executed if another action calls for it.

4.2.3.2.2 compile

This stage is conceptually the same as the one defined in the development pipeline (4.2.3.1.5 compile).

The only difference is that the compile stage of the staging pipeline is run every time there is a push or pull request to the main branch. This was decided because changes can be pushed to other branches during development that may not be complete, and seeing these commits fail at these stages does not contribute to the quality of the project. It can also be executed if another action calls for it.

4.2.3.2.3 test

This stage is conceptually the same as the one defined in the development pipeline (4.2.3.1.7 test). However, for this stage, only the unit tests that use test doubles are run as they are run in a cloud-hosted virtual machine with no access to RIO boards for the functional tests. Tests are more detailed in section 4.3.

As already mentioned, the verify stage in the development pipeline only runs unit tests. The tests are also run every time there is a push or pull request to the main branch. This test stage also uploads the test results as an artifact of the workflow. It can also be executed if another action calls it.

4.2.3.2.4 sonar

It uses SonarCloud, a cloud-based static analysis tool that measures the code's quality. It detects possible issues and calculates an approximation of several quality metrics, such as code complexity, code coverage, and duplicated code. Then, quality gates can be applied to see if the project meets the criteria. Quality gates are a set of quality-related parameters that the projects must achieve.

In the case of this stage, it generates the code coverage; however, lcov, the tool used for it, is not compatible with SonarCloud, so another step is added into the stage to convert it to one of the supported ones; in this case, it was decided to be cobertura, this conversion is achieved through a Python script, lcov_cobertura. Once the analysis is completed, it is sent to SonarCloud servers, which respond to whether the code passes the different quality metrics configured. The configured Quality Gate parameters are displayed in Table 4.6.

Metric	Applies to	Threshold
Coverage	New code	Greater or equal to 80%
	Overall code	Greater or equal to 90%
Duplicated lines	New code	Less or equal to 3%
	Overall code	
Issues	Overall code	Less than 100 issues
Technical Debt	Overall code	Less than one day of work (8 hours)
Maintainability Rating	New code	Grade A
	Overall code	
Reliability Rating	New code	Grade A
	Overall code	
Security Rating	New code	Grade A
	Overall code	

Table 4.6: Quality Gate SonarCloud for IrioCoreCpp

The sonar stage runs every time a push or pull request is made to the main branch. It can also be executed if another action calls for it.

4.2.3.2.5 doc

Generates the Doxygen documentation. It is the same as the one for the development pipeline (4.2.3.1.9 doc).

The doc stage is executed every time a push or pull request is made to the main branch, and it uploads the resulting HTML as an artifact. Although at first glance it may seem like there is no need to generate the documentation for every change, running it allows one to check that the comments included, or the lack of, still generate the documentation without warning, as it is configured to treat warnings as errors.

It also can be executed if another action calls it.

4.2.3.2.6 package

Creates the project packages. It works almost the same as the package stage in the development pipeline (4.2.3.1.10 package). In this case, extra dependencies, `doxygen` and `graphviz`, must be installed. Although it uses a custom Docker image as the base for this stage (4.2.2.2 Docker), it was decided not to install those to keep the image size down due to only being used in this stage. Another difference is that the development pipeline package stage only generated RPMs. This one converts them to DEB packages to use in Debian-based systems. This is done through the use of the `alien` tool. Finally, this stage also uploads the generated packages, both RPMs and DEBs, as artifacts.

The package stage is executed only manually or if another stage calls it.

4.2.3.2.7 release

The objective of the release stage is to avoid generating a release that does not meet the quality criteria. Due to this, the stage comprises multiple jobs, each responsible for specific tasks to ensure this quality. Most of the stages are reused from other parts of the pipeline; this stage calls them instead of being triggered by the tag push because it is easier to control if another of the stages fails and avoids generating a release.

The different jobs of the release stage are illustrated in Figure 4.4, along with the dependencies of other jobs. Table 4.7 covers each job and their descriptions.

Job	Description
verify	Runs linter and checks project structure and file naming conventions. Uses the verify stage (4.2.3.2.1 verify)
compile	Compiles the project source code. Uses the compile stage (4.2.3.2.2 compile)
checkVersion	Checks the constant defined in the main Makefile `VERSION`, which is then used to fill the constant `IRIOVERSION`, present in IrioCore's `irioDriver.h`, matches the tag created
doc	Generates Doxygen documentation. Uses the doc stage (4.2.3.2.5 doc)
package	Creates the project packages. Both RPMs and DEBs. Uses the package stage (4.2.3.2.6 package)
unittest	Runs unittests for BFP, IrioCore and IrioCoreCpp. Uses the test stage (4.2.3.2.3 test)

Continued on next page

Table 4.7: Description of each job in the release pipeline

Job	Description
sonar	Static analysis of code quality and security, including line and function coverage. Uses the sonar stage (4.2.3.2.4 sonar)
publishDoc	Uses the HTML Doxygen documentation artifact generated by the doc job and publish it to the project's GitHub Pages
genRelease	Once all previous jobs have been completed successfully. It creates the GitHub release, attaching the RPMs and DEBs artifacts created by packaging.

Table 4.7: Description of each job in the release pipeline

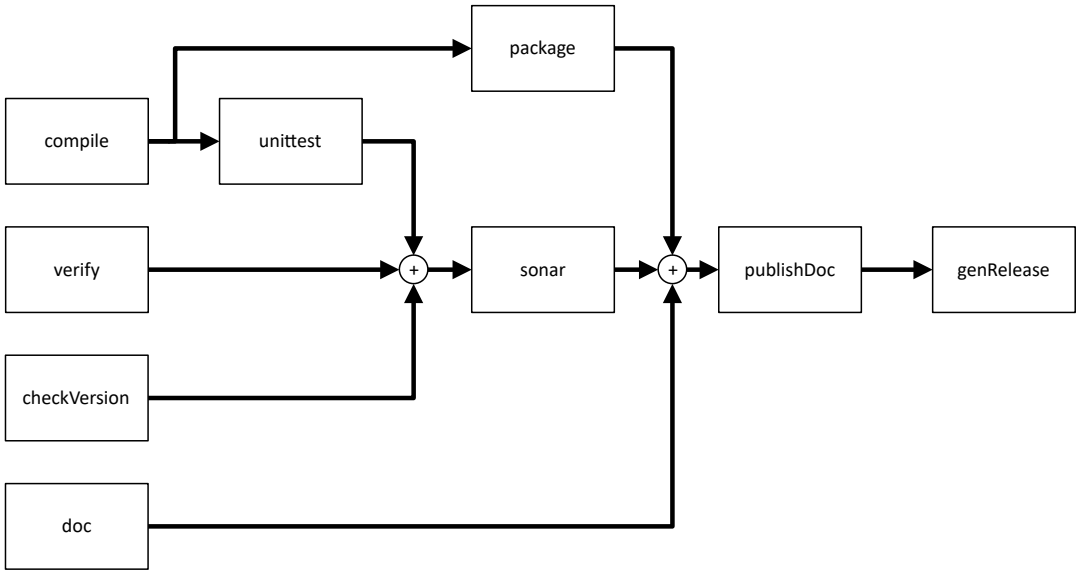

Figure 4.4: Jobs relationship in release stage

4.3 Testing

For testing all libraries, BFP, IrioCore and IrioCoreCpp unit and functional tests were created. It was decided to use the Google Test framework for this purpose as it is a robust testing framework, providing assertions and fixtures that facilitate the creation of tests. Moreover, as a C/C++ framework, it can be used for both IrioCoreCpp and IrioCore.

A Python script was developed to facilitate the management of test execution and the required configuration. The script utilises XML files as inputs, containing all the test groups to be executed and their configuration if necessary. This approach enables the execution and reporting of multiple types of tests with a single execution. It also allows specific filters to execute only particular tests, making creating more detailed test reports possible. These files have been named *testplans*. Each contains one or more test groups, executed separately, and their results are also reported in different

groups.

Several possible *testplans* are included with the project; however, custom ones can also be created. A schema has been provided; the custom test plans must comply to guarantee the Python test script works correctly.

4.3.1 Unit-testing

4.3.1.1 BFP

The BFP library only has unit-tests as it is relatively simple, and all its functionality could be tested with them. The test includes testing different types of resources, accessing the data parsed and testing that invalid inputs result in controlled errors. These tests could be run without external hardware, so they parse a file; test doubles were not used.

4.3.1.2 IrioCore

Tests were divided into different test suites, each testing different code sections. They are divided according to their distribution in the source files. The tests are grouped into Common, cRIO, FlexRIO, Analog, Digital, DMA, IMAQ, IO and Signal Generation. As this library calls the functions in IrioCoreCpp, test doubles were used for the functions that called the low-level driver. Each test suite sets the minimum common test values in their fixture.

4.3.1.3 IrioCoreCpp

Several tests were created to test each part of the library. Depending on which Terminal the code was being tested, they were divided into groups and test suites, adding common functionality and modules. As the low-level driver functions are called, test doubles were used to avoid this dependency. In its fixture, each test suite sets the minimum common values to test the functionality. The test suites are divided into Common, Analog, Aux Analog, Digital, Aux Digital, DMACPUCommon, DMACPUDAQ, DMACPUIMAQ, IO, GPU, SignalGenerator, FlexRio, cRIO and Modules. Although there is a test suite for GPU, this functionality is not available, and the test only checks that the corresponding error of not being implemented is thrown.

4.3.1.4 Test doubles

The use of test doubles for unit tests is motivated by several factors, the primary of which is it avoids the need to utilise real hardware, which is expensive and not always available. It also has extra benefits, such as reducing the time required to run the tests, being able to integrate them into a CI pipeline, and the capacity for multiple developers to test their changes without taking turns to test on the RIO boards.

The framework selected for test doubles was fff [20] for its simplicity. It is handy for creating fake implementations of functions in C, which are mainly the functions that interact with the RIO boards. Figure 4.5 offers a high-level view of this process.

A problem that had to be solved was with functions that, in some cases when reading a resource from the FPGA, a series of specific values needed to be read to test the different parts of the code. Due to using the same low-level function, creating a fake

that returned one value was not possible. The solution implemented is that during the test fixtures or the tests, one could use test functions that add to a map the resource address and the value it should return. These values are stored in a map, and when one of these low-level functions is called, the map is accessed, and the value specified is returned.

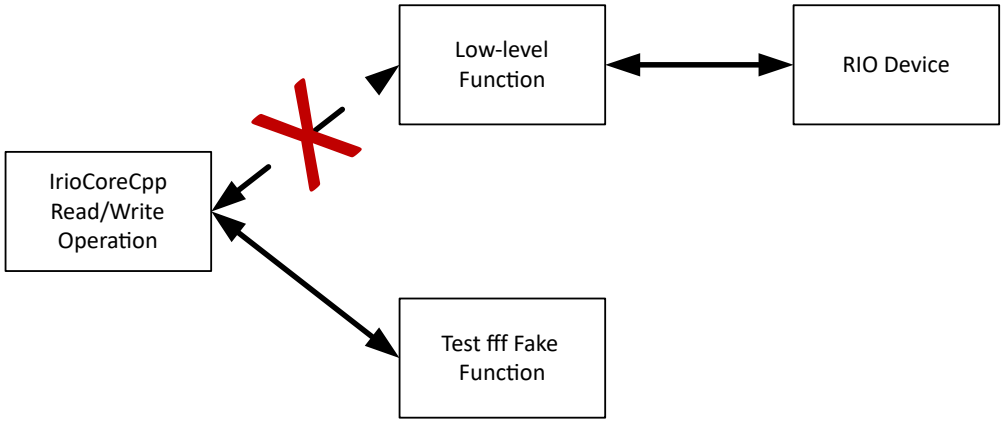

Figure 4.5: Diagram for the use of fff for low-level calls to RIO devices

4.3.2 Functional testing

Functional tests were developed for both IrioCore and IrioCoreCpp. In the case of IrioCore, these tests were reused from existing ones. Both for testing the library linkage was correct, and it replaced the previous IrioCore to ensure the behaviour was the same. For IrioCoreCpp functional tests, different test suites were created for each of the supported Profiles and Modules to test all the available functionality.

Functional tests require the use of the RIO devices. The device used for the tests needs to be specified, along with extra parameters, so the test runs accordingly. Table 4.8 summarises the required environment variables and those used only for specific ones.

Environment variable	Description
RIOSerial	RIO device serial number. Needed for all tests.
RIODevice	RIO device model. Needed for all tests.
Coupling	Coupling mode for test using the Module 5761
maxCounter	Max image counter for IMAQ tests

Table 4.8: Environment variables for functional testing

Functional tests covering common

Functional tests that test common functionality have been developed in a way to be possible to use interchangeably any of the RIO devices. The test will choose the appropriate bitfile accordingly to the type of device specified in `RIODevice`.

4.3.3 Integration testing

Integration testing has not been the focus of this project. However, it has been tested manually that the new implementation, IrioCoreCpp, still works with other applications. These applications are part of science experiments on ITER, and it is important to maintain compatibility with them as changing them is not always possible.

These tests have been conducted following the instructions in their respective test plans, and they make sure they pass successfully. The only change to them is that they have been recompiled, linking to the IrioCoreCpp library.

4.4 Extending IrioCoreCpp: Methodology and Maintenance

This section will guide the reader through steps that must be followed when introducing functionality to IrioCoreCpp. The guide is divided into different parts: The first part is about IrioCoreCpp, explaining the steps taken for defining new resources, creating or expanding Terminals, Modules, and Platforms and creating or expanding Profiles. Then, it delves into how to introduce new changes into the C wrapper, IrioCore. And finally, it describes how to create the tests.

4.4.1 IrioCoreCpp

IrioCoreCpp is divided into several elements, Resources, the FPGA elements on which read/write operations are performed and how the application is controlled; Terminals, groups of Resources according to the diverse functionality that RIO devices support; and Profiles, group of Terminals depending on the type of applications that can be implemented. There are also Platforms, which are the type of RIO device supported and information regarding them, and Modules, which are adapters that add specific functionality to the RIO devices. These elements are more detailed in section 3.4.

This section outlines the complete process of creating Profiles, including the Resources and Terminals associated, explaining the necessary steps for their creation and maintenance. While the overall process is described, different subsections allow for the completion of intermediate steps, such as adding Terminals to a Profile or adding Resources to Terminals.

4.4.1.1 Resources

Defining the Resources consists of defining their names as constant in the appropriate header file. The Resources names are divided accordingly on which Terminal they are parsed. The files are located in the folder `irioCoreCpp/include/terminals/names`. Their naming convention is as follows: `namesTerminals<Terminal name>.h`.

In this header file, the Resource name must be all upper case, start with `TERMINAL_` and follow as closely as possible the name in the FPGA, for example, `AOEnable` constant name would be `TERMINAL_AOENABLE`.

These names must be defined as `constexpr` and `char[]`. While using `#define` would also result in the same functionality, `constexpr` must be used as it adds type

checking, making the code safer. The reason `std::string` or similar is not used for the type is that up to C++17, it is not possible to define an immutable string-like variable.

If the header file is created, the first line of it must be `#pragma once` to avoid including the header file multiple times and causing problems compiling. While `#pragma once` is a non-standard preprocessor, it is widely supported. It creates cleaner and simpler code than the alternative, "include guards".

```
1 #pragma once
2
3 constexpr char TERMINAL_SGNO[] = "SGNo";
4 constexpr char TERMINAL_SGSIGNALTYPE[] = "SGSignalType";
5 constexpr char TERMINAL_SGAMP[] = "SGAmp";
6 constexpr char TERMINAL_SGFREQ[] = "SGFreq";
7 constexpr char TERMINAL_SGPHASE[] = "SGPhase";
8 constexpr char TERMINAL_SGUPDATERATE[] = "SGUpdateRate";
9 constexpr char TERMINAL_SGFREF[] = "SGFref";
```

Listing 4.1: Example Resource names for TerminalsSignalGeneration

4.4.1.2 Terminals

subsection 3.4.5 covered Terminals groups and how internally they are divided into Terminals and TerminalsImpl classes.

4.4.1.2.1 TerminalsImpl class

When implementing a new Terminal or modifying an existing one, the process begins by creating or modifying the TerminalsImpl class. First, the header file must be created; it must be added to `irioCoreCpp/include/terminals/impls` and must follow the naming convention of: `terminals<Terminal Name>Impl.h`

When defining a new TerminalsImpl class, it must always inherit the class `TerminalsBaseImpl` or a class that ultimately inherits it. The class name must also follow the naming convention of: `Terminals<Terminal Name>Impl`. The TerminalsImpl class must always be part of the namespace `irio`.

TerminalsImpl classes must at least have the same methods as the ones the user can access using the non-Impl ones. However, the ones in TerminalsImpl must end with Impl in their name. TerminalsImpl classes could also require additional methods or functions to implement the functionality.

As constructor parameters, it should have, at the very least, one for the `ParserManager`, which is responsible for parsing and logging error searching for Resources, and the session required to interact with the FPGA.

4.4.1.2.2 Terminals class

After the TerminalsImpl class has been created, the non-Impl one presented to the user must be created. The header file must be at `irioCoreCpp/include/terminals`, and the file name must follow the naming convention: `terminals<Terminal Name>.h`. This header file must not include the corresponding Impl file, as it will only

be included in the source file. This is done to avoid giving the user access to the Impl files; once the library is compiled, the included Impl files must not be distributed to the final user. The Terminals classes must inherit the `TerminalsBase` class or one that ultimately inherits it. It must also be part of the `irio` namespace.

The Terminals source files must include the equivalent TerminalsImpl header file. The Terminals class must provide methods for the user as required. As constructor parameters, it should have, at the very least, the `ParserManager`, which is responsible for parsing and logging error searching for Resources, and the session required to interact with the FPGA.

As covered in subsection 3.4.5, the non-Impl Terminals must only call the equivalent Impl methods of the TerminalsImpl. First, the equivalent TerminalsImpl class should be created and passed to the parent class to achieve this. This class expects a shared pointer; to do so, use `std::make_shared`. Listing 4.2 provides an example of how the constructor should be.

```
1  TerminalscRIO::TerminalscRIO(ParserManager *parserManager,
2                               const NiFpga_Session &session) :
3  TerminalsBase(
4      std::make_shared<TerminalscRIOImpl>(parserManager, session)) {
5  }
```

Listing 4.2: Example constructor Terminals class

The Terminals class must provide methods for the user as required. These methods must call the Impl method, accessed through the `m_impl` variable provided by `TerminalsBase`. However, `m_impl` is an object of type `TerminalsBaseImpl`, so it must be cast up to the appropriate `TerminalsClassImpl` so the relevant methods can be accessed. This can be achieved using `std::static_pointer_cast`. Listing 4.3 illustrates an example of this process.

```
1  bool TerminalscRIO::getcRIOModulesOk() const {
2    return std::static_pointer_cast<TerminalscRIOImpl>(m_impl)->
         getcRIOModulesOk();
3  }
```

Listing 4.3: Example calling a TerminalsImpl method from the Terminals class

4.4.1.3 Profiles

Profiles are groups of Terminals (subsection 3.4.6). So, the end purpose is the process of how to add these Terminals.

One can consider that there are two types of Profiles. There are Profiles for the different functionalities and Profiles for a specific functionality for a particular RIO Platform. For example, there is one Profile for CPUDAQ functionality, `ProfileCPUDAQ`, and then there are more Profiles for each of the Platforms with CPUDAQ capabilities, `ProfileCPUDAQcRIO`, `ProfileCPUDAQFlexRIO`, and `ProfileCPUDAQRSeries`.

4.4.1.3.1 Profiles specific to functionality

First, the header file must be created, located in the following directory `irioCoreCpp/include/profiles`. The file name must follow this naming convention: `profile<Functionality>.h`. The Profile must be part of the `irio` namespace.

Of the two Profiles, this is the main one, as it is the one that provides almost all the Terminals required. It must inherit `ProfileBase`, which provides methods to add Terminals and retrieve them. So, in the constructor, it must add all the required Terminals using the method `addTerminal`, which requires as argument an rvalue of the specific Terminal, as illustrated in Listing 4.4. The source file must be in `irioCoreCpp/profiles` and must adhere to this naming convention: `profile<Functionality>.cpp`.

```
1  addTerminal(TerminalsAnalog(parserManager, session, platform));
```

Listing 4.4: Example addTerminal in Profile class

As these classes' only purpose is to create the Terminals, no extra public methods should be defined. As the ones needed, the ones to retrieve the Terminals are already provided by `ProfileBase`.

As constructor arguments, it should have a pointer to the `ParserManager`, which logs errors parsing Resources and would be passed onto the created Terminals, the session needed to interact with the FPGA using the low-level driver, the Platform object with information of the specific RIO Platform being used and an identifier of the Profile that is being created.

4.4.1.3.2 Profiles specific to functionality and to RIO Platforms

First, the header file must be created; this must be in the following directory `irioCoreCpp/include/profiles` and its file name follow the next rule: `profile<Functionality><RIO Platform>.h`. The Profile must be part of the `irio` namespace.

These Profile classes must inherit the relevant functionality Profile from the previous section (4.4.1.3.1). It must only add the specific Terminals to the Platform if it exists.

As these classes' only purpose is to create the Terminals, no extra public methods should be defined. As the ones needed, the ones to retrieve the Terminals are already provided by `ProfileBase`.

Its source file must be located in `irioCoreCpp/profiles` and must adhere to the naming convention: `profile<Functionality><RIO Platform>.cpp`.

As constructor arguments, it should have a pointer to the `ParserManager`, which logs errors parsing Resources and would be passed onto the created Terminals, the session needed to interact with the FPGA using the low-level driver, and the Platform object with information of the specific RIO Platform being used. When calling the parent class constructor, it should also be defined as a constant of the particular profile ID. Listing 4.5 shows an example of this process.

```
ProfileCPUDAQFlexRIO::ProfileCPUDAQFlexRIO(
        ParserManager *parserManager,
        const NiFpga_Session &session,
        const Platform &platform)
    : ProfileCPUDAQ(parserManager, session, platform,
          PROFILE_ID::FLEXRIO_CPUDAQ) {
    addTerminal(TerminalsFlexRIO(parserManager, session));
}
```

Listing 4.5: Example Profile specific to a functionality and to a RIO Platform

These specific Profiles must be included in the following header file:
`irioCoreCpp/include/profiles/profiles.h`

4.4.1.4 Platforms

Platforms are classes that hold information about types of RIO boards (subsection 3.4.4). In general, it should not be necessary to create new Platforms. However, if it were required to create new ones or modify the existing ones, this section covers how.

Platforms are defined in the file `irioCoreCpp/include/platforms.h`. All Platforms must inherit from the class Platform and declare private constants with the maximum numbers supported for each. The Platform must be part of the `irio` namespace.

If a new Platform is created, it must be added to the enum `PLATFORM_ID`, which is also defined in `platforms.h` and define the expected value in the Resource `Platform` in the FPGA that would indicate that value. In `irioCoreCpp.cpp`, in the method `Irio::selectDevProfile`, it should be added to the map the valid Profiles for the new Platform and add the cases in the switch that created the selected Profile. This means that new Profiles would need to be created for this Platform following the indications in the previous section (4.4.1.3).

4.4.1.5 Modules

The different Modules classes hold information about analog acquisition and generation (subsection 3.4.3). To add new modules, a new class must be created for each new one. To do so, first, it must be defined in `irioCoreCpp/include/modules.h`. This new class must inherit the `Module` class, which provides methods to get the different parameters and to change the AC coupling mode. It should define the various configuration parameters as private variables depending on the AC coupling mode; this must be done using the structure `ConfigParams`. Adding the new Module to the enum `ModulesType` is also required. The Module must be part of the `irio` namespace.

The implementation of the new Module must be in `irioCoreCpp/modules.cpp` and the constructor must be called the method `addConfig`, provided by the parent class `Module`, as many coupling modes support, passing each time the configuration parameters for that Module. It must also set a default AC coupling mode on it. Listing 4.6 provides an example of it.

```
ModuleNI5761::ModuleNI5761() :
    Module(ModulesType::FlexRIO_NI5761) {
  addConfig(CouplingMode::AC, m_configAC);
  addConfig(CouplingMode::DC, m_configDC);
  setCouplingMode(CouplingMode::AC);
}
```

Listing 4.6: Example Module constructor

4.4.1.6 Errors/Exceptions

Defining new exceptions for IrioCoreCpp involves defining new classes that inherit from the `IrioError` class or any of its derived classes if more specific behaviour is needed. This exception must be defined in `irioCoreCpp/include/errorsIrio.h` and must be part of the `irio` namespace and inside this namespace must also be part of the `errors` namespace (full namespace: `irio::errors`).

`IrioError` inherits from `std::runtime_error`, this exception's constructor expects an `std::string` with the exception message. If any new exception needs the same type of constructor, use the `using` keyword, which tells the compiler to inherit the contructor. Listing 4.7 provides an example of an exception which reuses the IrioError constructor, which in turn uses the `std::runtime_error` constructor and also provides a default constructor with a predefined message.

```
class ProfileNotImplementedError: public IrioError {
  using IrioError::IrioError;
 public:
  ProfileNotImplementedError() :
      IrioError("Profile not implemented yet") {
  }
};
```

Listing 4.7: Example Module constructor

4.4.2 IrioCore

Once the required Terminals, Profiles, Modules, Platforms, or exceptions have been added or modified in IrioCoreCpp, they should be added to IrioCore.

If new functions need to be created, they must be separated from the existing files, depending on their core functionality. In general, in the header files, there must only be functions that the user can use, no internal function, and their names should always start by `irio_`. However, if a generic function is created that can be used in different parts of IrioCore, then it can be defined in `irioUtils.h` and implemented in `irioUtils.cpp`, as this header file would not be included in the final package.

Generic functions are provided for get and set operations. They use as parameter an `std::function<void()>` where multiples behaviours can be implemented, but they wrap this function in a try-catch block with the typical exceptions get/set operations will be thrown, the catch here would convert the exception in a valid Status that can be returned in the IrioCore functions. If extra exceptions are created for get/set operations, another catch should be added to the `operationGeneric` function.

In general, if a new Terminal has been added, a function must be created in `irioUtils` to get them to facilitate the process when implementing the functionality. These functions will get an instance of the `Irio` class, which should have been created before by the user calling `init_driver`, and then use the `getTerminals<name Terminal>()` methods of `Irio`, Listing 4.8 shows an example.

```
1  irio::TerminalsDigital getTerminalsDigital
2                          (const std::string &rioSerial,
3                            const std::uint32_t session) {
4    return IrioInstanceManager::getInstance(rioSerial, session)
5                  ->getTerminalsDigital();
6  }
```

Listing 4.8: Example getTerminals function in irioUtils in IrioCore

4.4.3 Packaging

In RHEL systems, packages are created using SPEC files. These files define the name, version, package requirements, and files to include in the package and where to install them, among others. For each of the packages defined above one SPEC file was created, these files have basic information such as the package name, the requirements and a description. To make the system more flexible, the version and the files are not directly included but are defined, copied and substituted in the SPEC files by other Makefiles, one per package.

Each of these Makefiles are executed when calling the package stage, they generate, if necessary, the structure needed for the SPEC files, sets the package version, and copy the files to the required folders to create the package. This is done through the use of a generic package, Makefile. This Makefiles expects several parameters to be defined when calling it, these parameters can be found in Table 4.9.

Parameter	Description
FILES	List of the files to include and where they should be installed. These must be done using : to separate the source file from the destination such as: `<srcFile1>:<dstFile1> <srcFile2>:<dstFile2>···`
SPEC_FILE	The SPEC file to use as a base should match the equivalent to the one defined in `PACKAGE_NAME`
PACKAGE_NAME	Name of the package
PACKAGE_VERSION	Version of the package
BASE_DIR	Base dir from which to search the source files. Generally, it should be: `$(TOP_DIR)/$(COPY_DIR)`

Continued on next page

Table 4.9: Parameters required for the generic package Makefile

OUTPUT_DIR	The directory on which to output the generated RPMs. Generally, it should be: `$(TOP_DIR)/$(COPY_DIR)/packages`

Table 4.9: Parameters required for the generic package Makefile

If a new package needs to be generated, a new Makefile for that package needs to be created in `workflowStages/packaging` and must be named as `package_<package name>.mk`. This Makefile must call the generic Makefile with the appropriate parameters (Table 4.9), and the generic Makefile name is inherited from the parent Makefile in the variable `PACKAGE_MK`. Other variables that can be useful inherited from the parent Makefile are summarized in Table 4.10.

Variable	Description
PACKAGE_MK	File name of the generic Makefile in charge of creating the packages
TOP_DIR	Top directory of the project
COPY_DIR	Directory on which the compiled project files are
BASE_LIB_INSTALL_DIR	Directory where libraries should be installed. Extra folders can be created if necessary, but always from this directory.
BASE_INC_INSTALL_DIR	Directory where header files should be installed. Extra folders can be created if necessary, but always from this directory.
BASE_DOC_INSTALL_DIR	Directory where generated documentation should be installed. Extra folders can be created if necessary, but always from this directory.

Table 4.10: Useful variables when creating packaging Makefile

Finally, to execute the Makefile when executing the package stage (paragraph 4.2.3.1.10), the newly created Makefile needs to be added to `PACKAGES_MAKEFILES` variable in the file `workflowStages/packaging/Makefile`.

Chapter 5

Results

5.1 IrioCoreCpp

5.1.1 Evolution

Figure 5.1 depicts the evolution of the development of the IrioCoreCpp library. It illustrates the evolution of coverage and Lines Of Code (LOC) over the total number of commits pushed. Coverage is displayed as line coverage and calculated by executing all available tests in that commit. However, as this is not feasible to do manually, a script was used to automatise the process, so some concessions have been made. Although most functional tests can be executed automatically, some require user input to configure the hardware or enter specific commands; these tests have not been run in these automatic tests. However, these tests are uncommon, so coverage is almost unaffected; depending on the case, coverage would only increase less than 4%. It also needs to be noted that due to not having all the modules required to test cRIO boards, the functional tests related to these boards have been skipped; the impact on this is more significant, as it could increase coverage up to 10%. The results from these tests can be found in the annexe.

In the initial phase of the project, spanning approximately 300 commits, it can be observed that the project's LOC vary rapidly. This is because the majority of the functionality is being developed. It can also be observed where new functionalities are introduced, as the LOC increase drastically. The coverage usually follows the LOC, rising along it. However, there are also periods where it can be observed that the coverage decreases significantly. In nearly all instances, this phenomenon follows an increase in LOC, indicating that tests were not developed for the newly introduced functionality. This is particularly evident in the initial 40 commits.

The gaps portrayed in Figure 5.1 indicate that the library did not compile in those commits, which are more common at the beginning of the project. However, from commit 275 to 307, a significant gap can be observed where the project did not compile. These commits resulted from a restructuring of the code, where multiple files were renamed and moved for better organisation, and numerous commits were pushed after a few changes, which resulted in this code instability. However, if the dates from both commits (`e1bdf51` and `e1bdf51` respectively) are checked, these commits were all pushed in two days. This was a byproduct of merging two branches and not squashing the changes.

Approximately at commit 330, the CI/CD pipelines were mainly fully developed and integrated into the process. From this point on, the project exhibits more stability, compilation-wise, and the sections where it does not compile are narrower than previously.

From commit 228 onwards, coverage remains high, only decreasing slightly between commits 310 and 340, but newer below 85%.

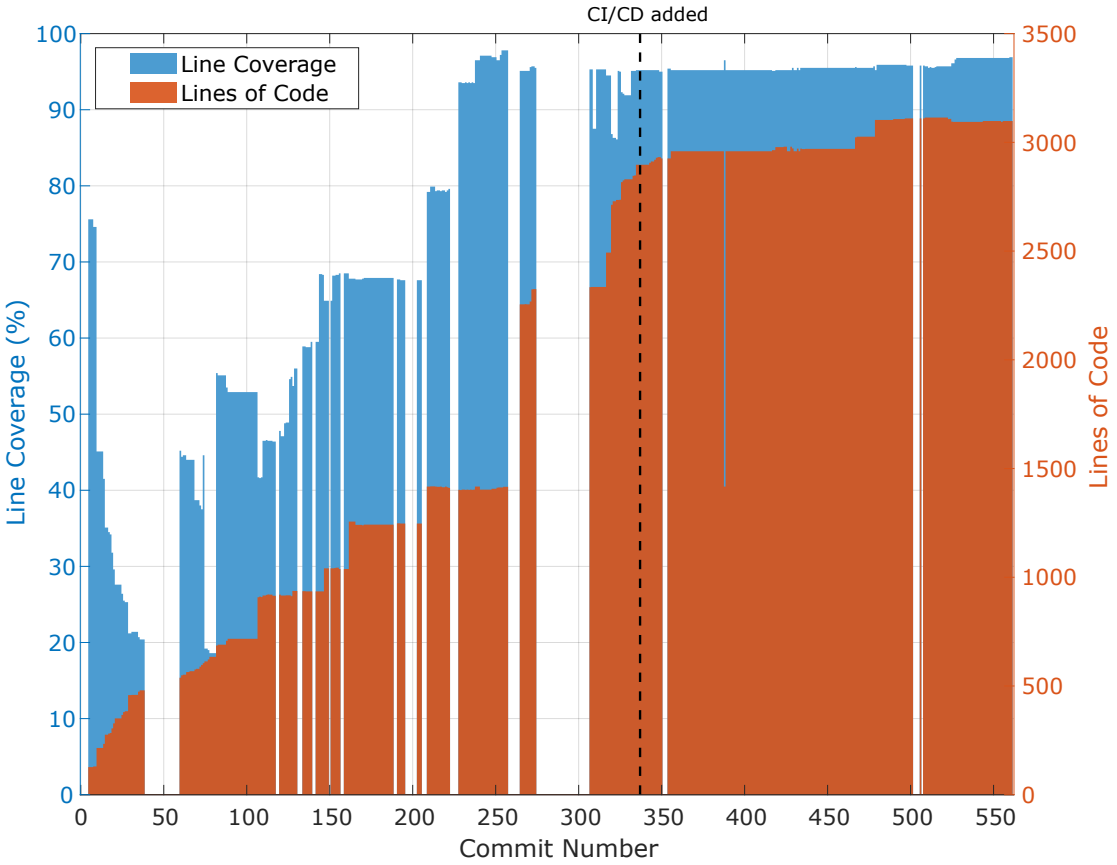

Figure 5.1: Evolution of IrioCoreCpp in Line Coverage and LOC

Figure 5.2 displays the evolution of function coverage through the project commits. The evolution is practically the same as in Figure 5.1. Until half of the project, function coverage is slightly lower than line coverage, around 10%. From the second half, both coverages remain close and above 95%.

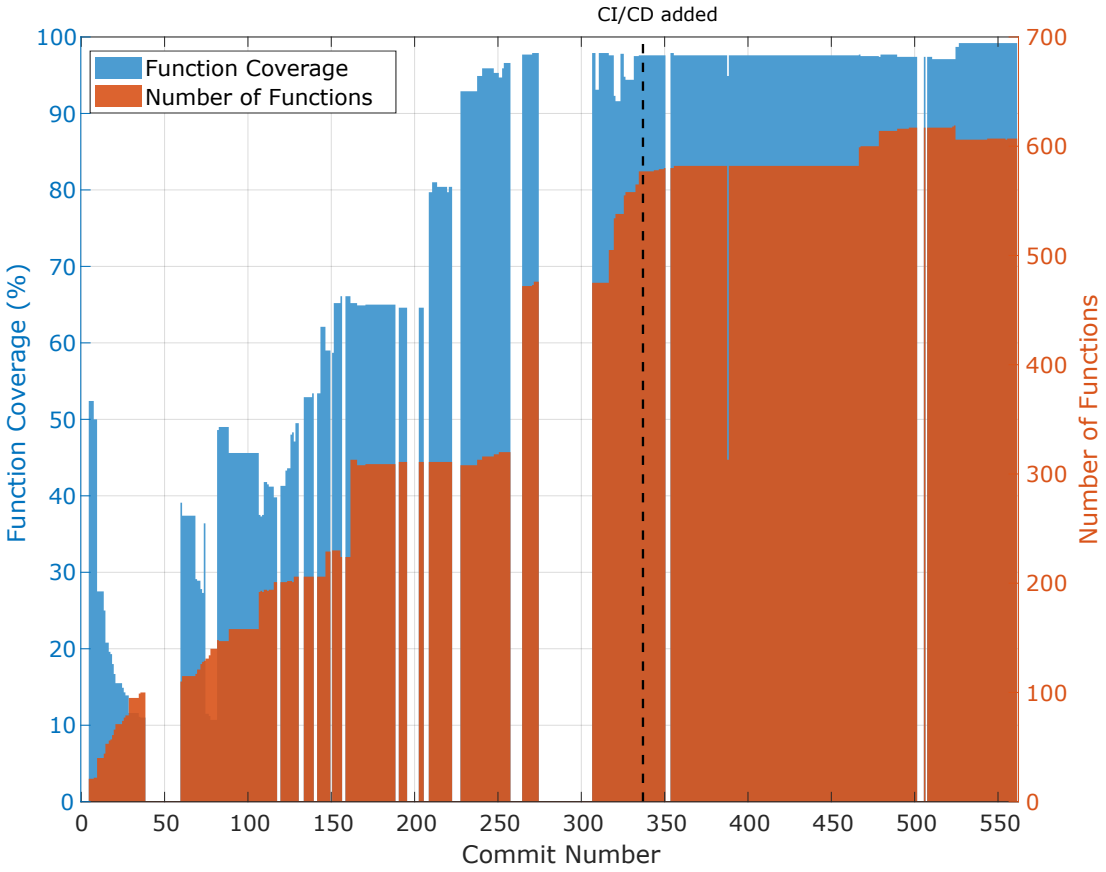

Figure 5.2: Evolution of IrioCoreCpp in Function Coverage and Number of Functions

5.1.2 Test Execution Time

Functional tests require real hardware and user interaction to configure them, so measuring the time it takes to run them is not possible; the estimation is around 20 minutes. In the case of unit tests, they do not require user input and, through the use of mocking, run much faster; the time they take to run depends on the system on which they are being run; however, it is significantly faster than functional, at around 10 seconds.

5.1.3 Static Analysis

Static analysis has been performed as part of the CI/CD pipeline. This was done using SonarCloud and the results of the analysis can be found in:

`https://sonarcloud.io/summary/overall?id=i2a2_irioCore&branch=v2.0.0`

Figure 5.3 shows the overall results of the analysis. The main points to observe are that the project does not have duplicated code, there are no issues related to security or reliability, and the maintainability issues remain low, at 14, which is calculated to suppose 57 minutes of effort according to the tool. It is necessary to explain that the problems accepted are 98 due to the majority being related to the tool detecting the new IrioCore library as C++ and applying rules related to this. The number of issues divided into the libraries is shown in Table 5.1 where it can be seen how 90% of the total issues are related to the new IrioCore.

Figure 5.3: Sonar static analysis results for IrioCoreCpp and the new IrioCore

Library	Number of accepted issues
irioCore	90
irioCoreCpp	8

Table 5.1: Number of accepted issues of IrioCoreCpp project

Other interesting metrics are the ones related to measuring the project's complexity. The analysis provides two types of complexity measurements: cyclomatic and cognitive. Cyclomatic complexity is a relatively widespread measurement in software development; it quantifies the number of paths in the code, indicating its logical complexity and the minimum number of test cases required for full path coverage. On the other hand, cognitive complexity is a measurement developed by SonarSource which offers an idea of how complex the code is to understand, taking into consideration human factors such as nested control structures and non-linear flow [48][49].

Table 5.2 and Table 5.3 shows the measurements obtained for the project's two parts. It shows how, in both cases, IrioCoreCpp is significantly higher than the new IrioCore; this makes sense, as this last one is a C wrapper, and most of the logic is in the IrioCoreCpp library. These results are complicated to analyse as there is no absolute scale, and it depends on the project. In this case, the complexity seems relatively high and would mean that some sections of code could be refactored to lower these values. Section 5.2.4 Static Analysis compared these values with the previous IrioCore, which may help to add a perspective of the changes between them.

	Cyclomatic Complexity
irioCore	236
irioCoreCpp	563

Table 5.2: Cyclomatic complexity of IrioCoreCpp project

	Cognitive Complexity
irioCore	81
irioCoreCpp	217

Table 5.3: Cognitive complexity of IrioCoreCpp project

5.2 Comparison Previous Library

5.2.1 Coverage

Figure 5.4 shows how line coverage has evolved compared to the previous IrioCore. The previous version of IrioCore only included functional tests, whereas IrioCoreCpp incorporates both functional and unit tests. However, it must be noted that the line coverage results are lower than expected in both functional test cases. This was because the modules for the cRIO boards were unavailable.

IrioCoreCpp improves coverage in functional tests by 13%. Including unit tests using test doubles also allows for the covering of code without the need for hardware. Also, it will enable testing error situations, which makes it possible to obtain its 95% line coverage. The graph also shows how combining unit and functional tests in IrioCoreCpp a 97.3% is obtained. This difference of 2.3% compared to unit tests is due to the code that searches if the specified RIO board is available, which is bypassed when using test doubles in unit tests.

The remaining coverage is mostly error situations where simulating the required condition is challenging, even with test doubles.

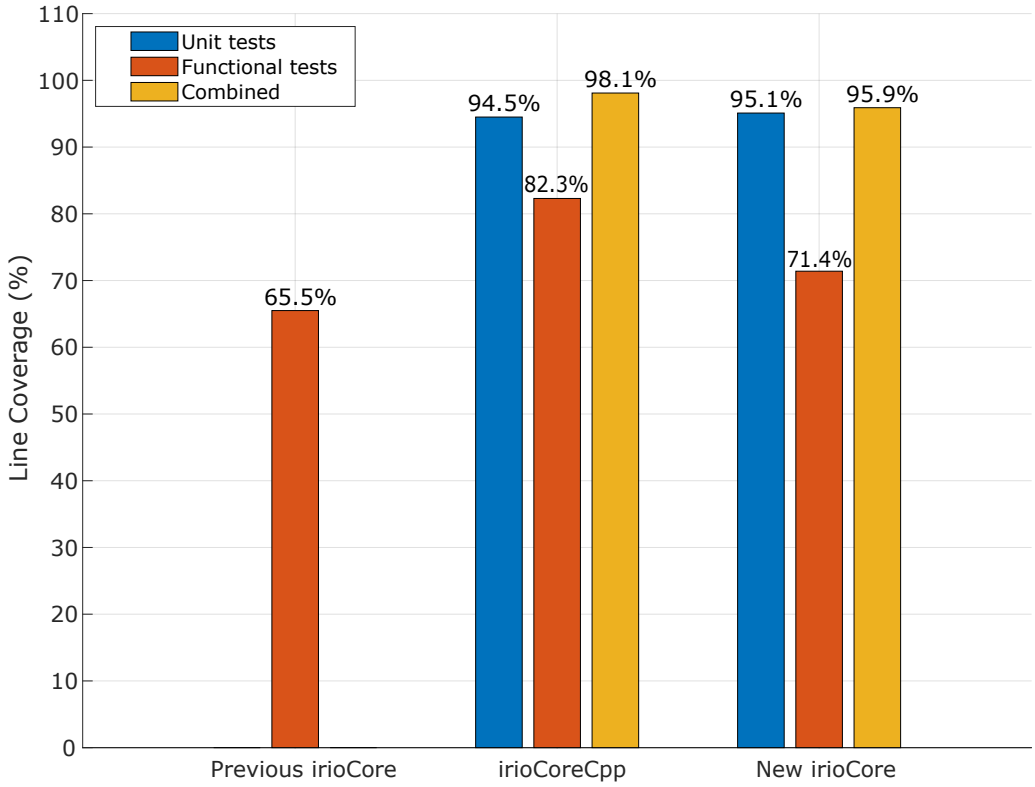

Figure 5.4: Line coverage comparison between previous IrioCore and IrioCoreCpp

Figure 5.5 shows the function coverage comparison between the previous IrioCore and IrioCoreCpp. However, it must be noted that the line coverage results are lower than expected in both functional test cases. This was because the modules for the cRIO boards were unavailable.

Functional tests in IrioCore cover most of the functions, with a 94% coverage.

In contrast, functional tests in IrioCoreCpp and the new IrioCore get 86.5% and 82.6%, respectively. These changes with unit tests obtained higher percentages of function coverage, with 96.8% for IrioCoreCpp and 98.1% for IrioCore. If the unit and functional tests are combined, almost 100% function coverage is obtained for IrioCoreCpp and the new IrioCore, with 99.8% and 99% each.

Comparing these results with Figure 5.4, it can be observed how the previous IrioCore only had 65.5% line coverage, a big contrast with its function coverage. This indicates that while functions were well-covered with the functional tests, many lines in those were not being executed.

In contrast, both IrioCoreCpp and the new IrioCore show similar behaviour in both line and function coverage, with similar results. Although the reason for lower function coverage than previous IrioCore is in part due to the use of template meta-programming, which generates functions for types which are not being tested in functional tests, these differences are covered in subsection 5.2.2.

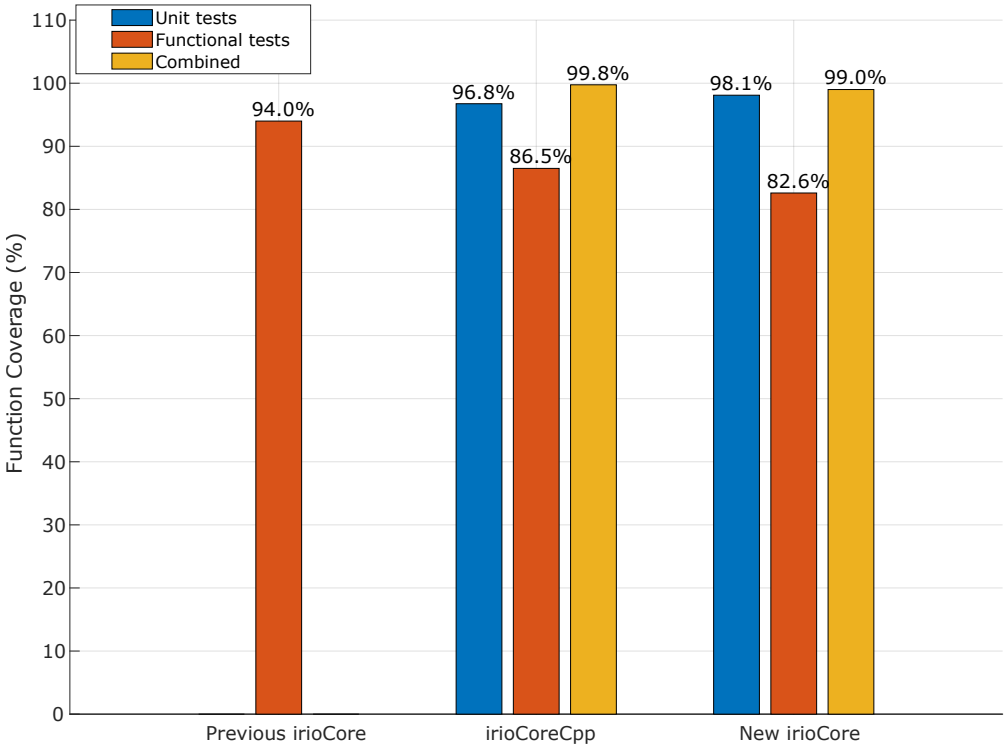

Figure 5.5: Function coverage comparison between previous IrioCore and IrioCoreCpp

5.2.2 Size

Figure 5.6 illustrates the difference between LOC and several functions between the source code of the previous IrioCore and IrioCoreCpp. The values represented in the graphs have been obtained from the coverage reports.

In the LOC section, it can be observed how the number of lines of code of IrioCoreCpp is reduced from the previous library. This is due to the use of features C++ allows, such as OOP and template meta-programming. Comparing the previous IrioCore to the new version, the number of LOC is reduced significantly, to less than half; this is due to the new version being a C wrapper to IrioCoreCpp, so the code is mainly calls to IrioCoreCpp classes and their methods.

From the function section, it can be observed how there is a big difference compared to the previous IrioCore. In both cases, IrioCoreCpp and the new IrioCore, the number of functions increased drastically; in the case of IrioCoreCpp, it is a 3.4x increase. In both cases, this is due to template meta-programming, where several are created at compile time from one definition of a function.

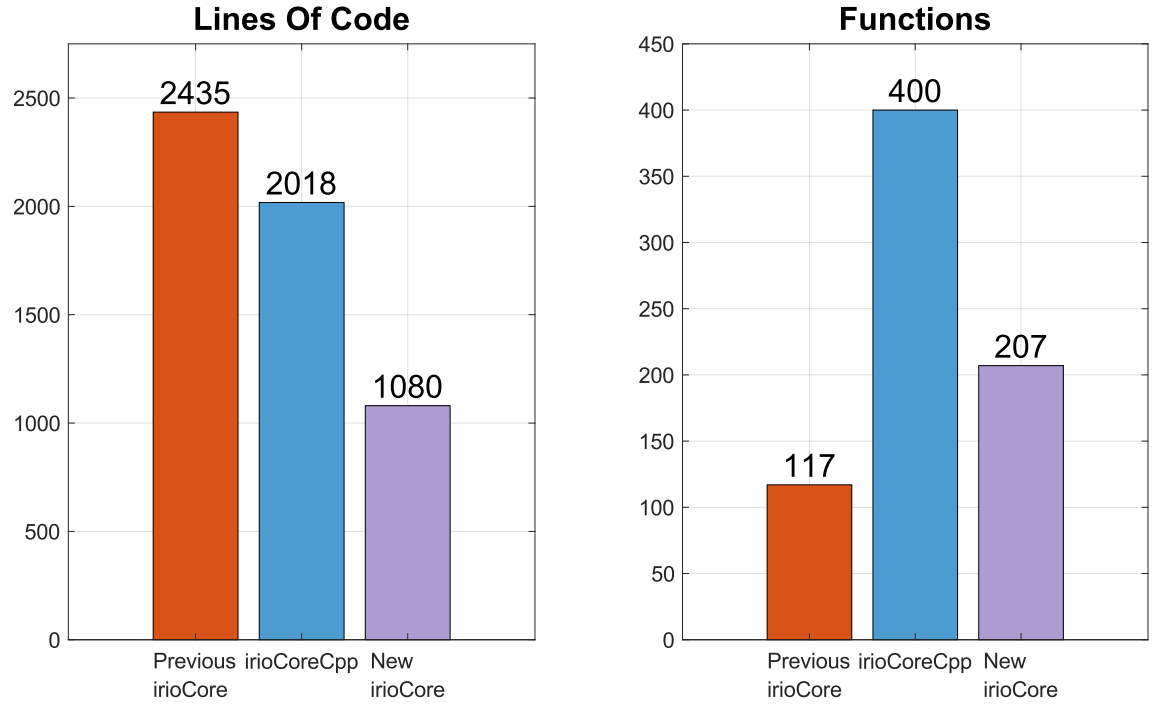

Figure 5.6: LOC and number of functions comparison between previous IrioCore and IrioCoreCpp

5.2.3 Tests

Figure 5.7 illustrates the difference in the number of tests and the LOC of these tests between the previous IrioCore and IrioCoreCpp and its C wrapper. The number of functional tests remains almost the same between them. However, the number of unit tests triples the number of functional tests for the case of IrioCoreCpp and its C wrapper, the new IrioCore.

Regarding LOC, the previous and the new IrioCore remain almost the same in functional tests, with the new one increasing by approximately 200 lines, primarily due to the extra two functional tests it has. IrioCoreCpp has the lower number of LOC of the three.

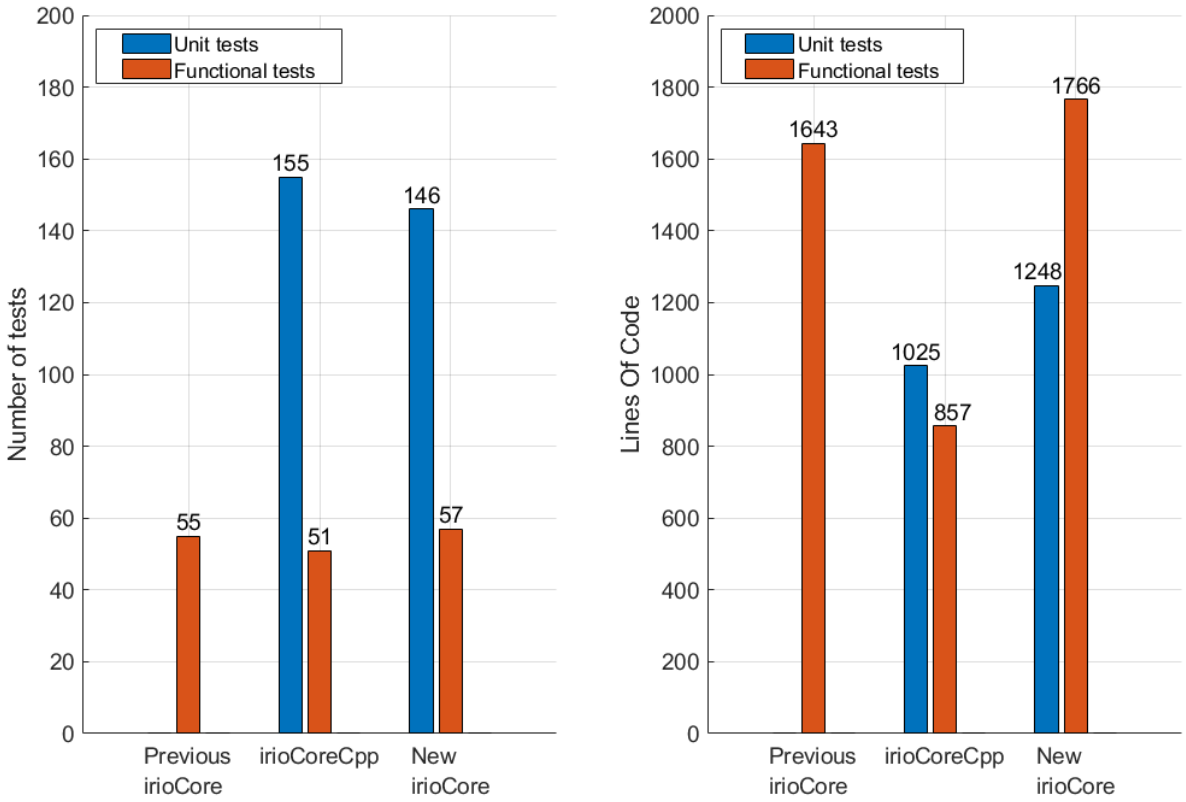

Figure 5.7: Number of tests and LOC comparison between previous IrioCore and IrioCoreCpp

Figure 5.8 represents the average number of LOC per test for unit tests, functional tests and their combined total across the previous IrioCore and IrioCoreCpp and its C wrapper, the new IrioCore. The average was calculated by dividing the total number of tests by the lines of code in each group.

The results indicate a significant difference between the previous and the new versions. With the average of the combined IrioCoreCpp and the new IrioCore being a third and a half, respectively, from the previous IrioCore. One crucial factor for this is the large number of unit tests, decreasing the average significantly, as in the

case of functional tests for IrioCore, the average of tests increases slightly compared to the previous version. However, in the case of IrioCoreCpp, the average remains low in both unit and functional tests.

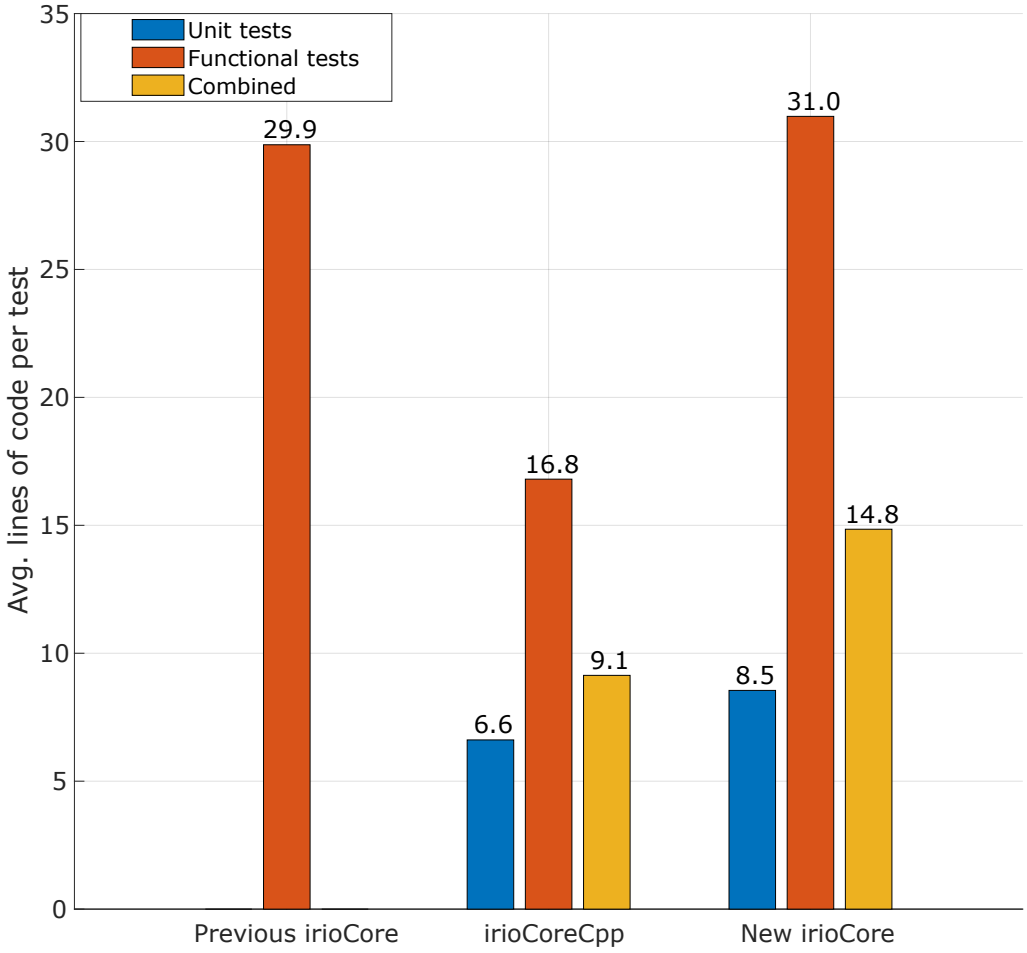

Figure 5.8: Average number of lines of code per test. Comparison between IrioCore and IrioCoreCpp

5.2.4 Static Analysis

Figure 5.9 shows the static analysis results using Sonar on the previous IrioCore library. The results have been compared against the same quality gate (Table 4.6) as IrioCoreCpp (subsection 5.1.3). This results in failing the analysis due to having 2 reliability issues, and thus rating lower than A, having more than 100 maintainability issues, with an accumulated debt of more than 1 day, and more than 3% duplicated lines.

Some points about the static analysis results of the previous IrioCore need to be considered. Coverage in this case is 0% as the project is not configured for a complete integration with SonarCloud. The security hotspots have not been mentioned as they do not necessarily mean issues and have not been revised. Lastly, the maintainability issues detected have not been completely revised, and there may be some false positives.

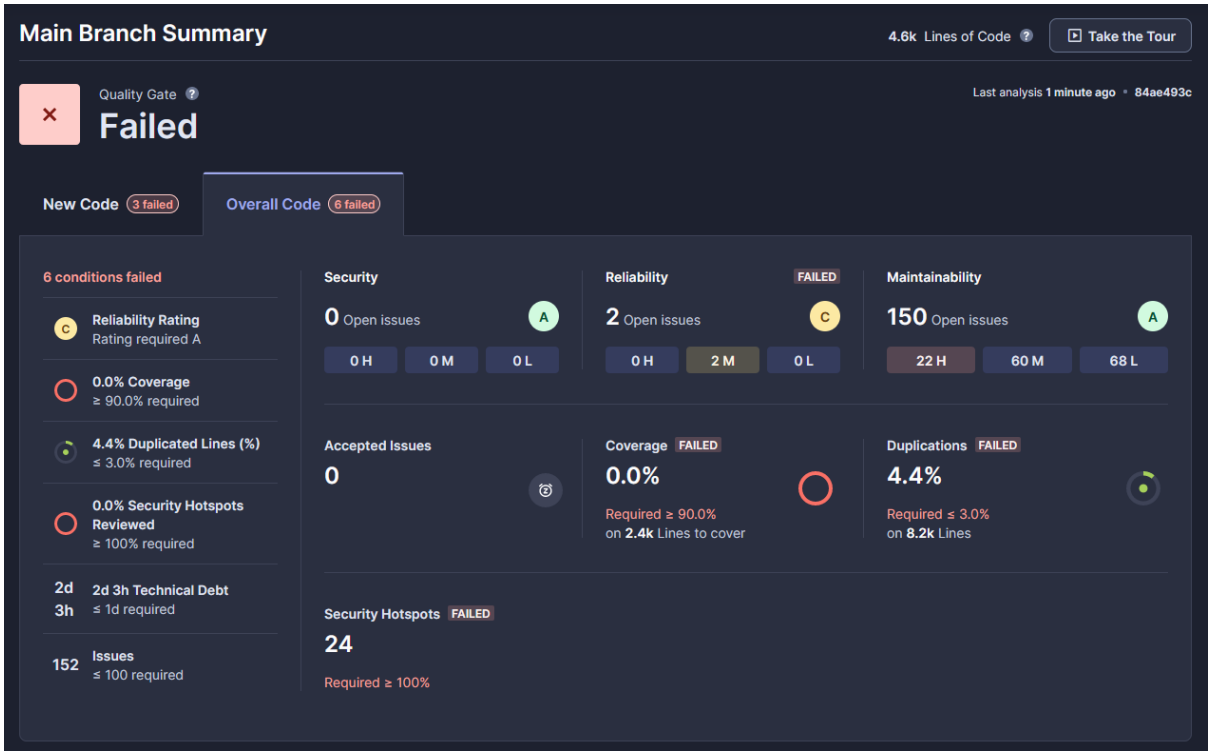

Figure 5.9: Sonar static analysis results for IrioCoreCpp and the new IrioCore

Table 5.4 compares the static analysis results of the previous IrioCore and IrioCoreCpp. The table shows how IrioCoreCpp exhibits fewer issues overall. Specifically, IrioCoreCpp shows no reliability issues and significantly fewer maintainability issues than the previous IrioCore. Additionally, IrioCoreCpp eliminates code duplications, whereas the previous IrioCore has a duplication rate of 4.4%.

	Previous irioCore	irioCoreCpp (including C wrapper)
Reliability issues	(0H, 2M, 0L)	(0H, 0M, 0L)
Maintainability issues	(22H, 60M, 68L)	(1H, 2M, 11L)
Duplications	4.4%	0%

Table 5.4: Comparison static analysis

Figure 5.10 compares cyclomatic and cognitive complexity between the previous IrioCore and IrioCoreCpp. The cyclomatic complexity is noticeably lower in IrioCoreCpp. However, the most significant difference lies in the cognitive complexity, where IrioCoreCpp shows a 4.7-fold reduction, indicating that it should be easier to understand for developers. Also, as expected, the new IrioCore has the lowest complexity scores, primarily serving as a wrapper with most functionality residing in IrioCoreCpp.

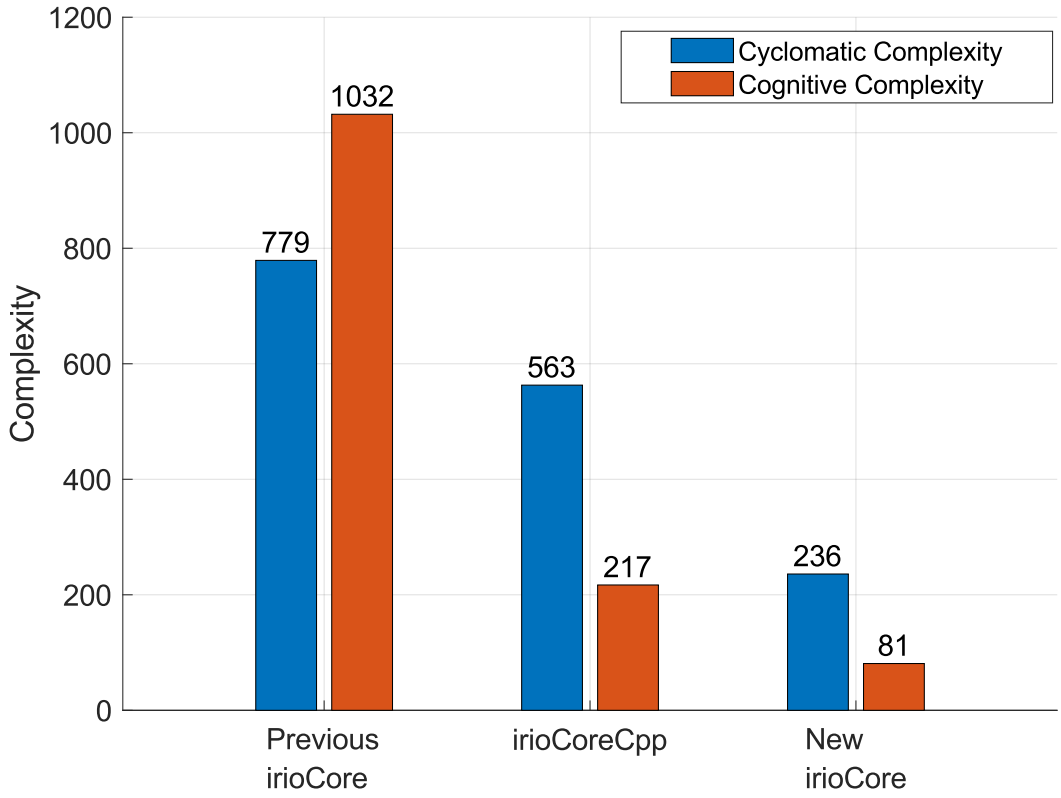

Figure 5.10: Cyclomatic and Cognitive complexity comparison

Chapter 6

Conclusions

The reimplementation of the code in C++ has been largely positive, successfully achieving the objectives set to varying extents. The code has been published as an open source software project, under a GPL-3 licence, in the following GitHub repository, `https://github.com/i2a2/irioCoreCpp`.

The primary objective was to create a product with the same functionality, which has been accomplished. The functional tests provided with the C wrapper library fulfil this role, as they are the same as the ones used in the previous library. However, this method of verifying functionality should be reconsidered and revised, as it depends on another part of the project, making it harder to detect problems. A proper requirement matrix would be desirable; however, due to these requirements not being public, it is not possible at the moment.

Support for R series RIO devices has also been successfully implemented, with the program's modularity now allowing changes like this to be made much more easily.

The use of C++ has enabled the usage of objects such as smart pointers and RAII for improved memory management. Nonetheless, the methodology could be enhanced, as it relies on static analysis tools, in this case, SonarCloud, to detect potential issues such as code smells. The use of dynamic memory analysis tools, like Valgrind, may have been preferable.

The dependency on Windows systems has been significantly reduced. While it is still mandatory to develop FPGA applications using LabVIEW, it is not necessary to use the `C API Generator`, as the `BFP` library has been developed. This makes it much more easy for developers to generate the required bitfile and use the IrioCoreCpp library with only that file.

The interaction with IrioCoreCpp and the C wrapper IrioCore has been improved by providing sufficient methods to interact with the different parameters in a controlled way. However, this was not always possible for the new version of IrioCore. While new methods have been added to manage some parameters, some previous applications used those parameters directly, necessitating maintaining this direct approach for compatibility reasons.

One of the objectives of this project was to make the code more straightforward to add functionality and help with maintenance. The measurements provided by the use of SonarCloud contributed to this goal, as there is now a considerable reduction in complexity between IrioCoreCpp and the previous IrioCore. The logical complexity of the code has been reduced by more than 25% according to cyclomatic complexity measurements and almost 80% in SonarCloud's cognitive complexity measurement. This can be attributed partly to using C++ features, such as OOP and templates. These features allowed for the simplification of operations, reducing the LOC and the types of functions required. This can be observed in Figure 5.8, where the average number of LOC per test is measured. The average LOC per test in the previous IrioCore is three times the average in IrioCoreCpp, while also obtaining higher coverage results (Figure 5.4).

The DevOps process has been significantly improved by establishing CI/CD pipelines with several stages covering different parts of the project, thus enhancing its overall quality.

One of these stages is the documentation stage, which has been modified compared to the previous IrioCore to now treat undocumented public methods as errors, prompting developers to fix them instead of being silent and accumulating as before.

Other significant areas of improvement have been in testing; the introduction of unit tests and the use of test doubles has been highly beneficial, as the time required to run tests has been significantly reduced. This has made it possible to identify and address error situations that were previously more challenging to reproduce reliably. Functional testing also verifies that the requisite functionality of the RIO boards is functioning as intended.

The reduction in test execution time, combined with the decoupling of the need to use specific hardware, has enabled the implementation of robust CI/CD pipelines that were previously undefined. These pipelines provide developers with a mechanism to detect issues earlier and more easily, automating most processes that previously had to be done manually, such as coverage reports, running tests, or static code analysis. This speeds up the development process and provides more robust and efficient mechanisms for quickly generating releases.

Using CI/CD pipelines allows for quicker coverage reports, resulting in high coverage percentages in both line and function coverage, both over 95%. In both unit and functional tests, the coverage results of IrioCoreCpp are higher than the previous IrioCore, especially in line coverage, showing how tests now cover more functionality.

As previously mentioned, the introduction of CI/CD pipelines has significantly improved the project's quality. The development pipeline allows developers to easily verify the code being implemented, while the staging pipeline ensures that the code being pushed to the repository meets the different quality goals while also offering mechanisms to guarantee quality releases.

It was also mentioned that a C wrapper was developed to provide backward compatibility with existing applications using IrioCore. The wrapper offered the same API, thus eliminating the necessity for source code modifications. However, to enhance the library's safety, internal resources were modified in the new IrioCore, making it not ABI compatible. This necessitates recompiling the projects to ensure proper functionality.

These points demonstrate how IrioCoreCpp generally represents an enhancement over the previous IrioCore. Its backward compatibility makes it an optimal alternative for existing Big Science projects that utilize IrioCore and future ones.

Chapter 7

Future Work

While this project successfully implemented IRIO software tools for the management of FPGA RIO devices into C++, several areas could be improved further and refined.

The current implementation on IrioCoreCpp of RIO boards' modules consists of classes for a series of specific modules that are supported. There are many possible modules, and depending on the application, others than the supported ones may be preferable. However, this would not be possible with the current implementation without modifying the code. Therefore, defining a more generic method to implement "custom" modules would be one potential way to make the library more flexible and future-proof. Nevertheless, it is also essential to consider that in large-scale scientific projects, introducing new hardware necessitates validation and potential changes to multiple systems. As such, it is not unreasonable that when new modules are accepted, modifications to the code could be made to accommodate new modules, as defining the various module parameters is a relatively straightforward process.

Currently, the NI System Configuration library is used to search for RIO devices connected to the system. During the search process, this library scans for all types of devices that may be connected through multiple interfaces, such as PXI, PCI or Ethernet. This leads to the call to search the boards to take some type to complete. While this is not a critical issue, as it only occurs when the Irio object is created, it would be desirable to attempt to add some configuration to filter interfaces and thus reduce the search time. Further research would be required into the library documentation to identify whether this is possible and, if so, how it should be implemented.

Another area of potential improvement is related to the CI/CD pipelines, precisely the packaging stage. At the moment, the process requires specific Makefiles for each package to be created. While Makefiles are a helpful tool, they lack the user-friendliness necessary for a more efficient workflow. Therefore, it would be beneficial to explore alternative methods for creating new packages or modifying existing ones more straightforwardly.

Although the code complexity has been significantly reduced from the previous IrioCore, some sections could still be refactored and simplified to achieve further reduction. The continued use of static analysis tools could prove beneficial in this regard.

Finally, the IRIO Design Tools are expected to be updated soon, and changes may be required to match the new specifications. Thanks to improvements in the development process, a new version supporting the new changes can be released much faster than before.

Bibliography

[1] J. Ousterhout, *A Philosophy of Software Design*, 1st. 2018, ISBN: 1732102201.

[2] Kharnagy. "Illustration showing stages in a devops toolchain." (2016), [Online]. Available: https://commons.wikimedia.org/wiki/File:Devops-toolchain.svg.

[3] Red Hat, Inc., *What is CI/CD?* 2023. [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[4] Netflix Technology Blog, *Global Continuous Delivery with Spinnaker*. [Online]. Available: http://techblog.netflix.com/2015/11/global-continuous-delivery-with.html.

[5] B. Homès, *Fundamentals of Software Testing*. eng, 1st ed. 2012, ISBN: 9781118603093.

[6] *Mock Dependencies in Tests*, MathWorks. [Online]. Available: https://es.mathworks.com/help/matlab/mock-dependencies-in-tests.html.

[7] H. Zhu, L. Wei, M. Wen, *et al.*, "Mocksniffer: Characterizing and recommending mocking decisions for unit tests," eng, in *2020 35TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE 2020)*, IEEE Comp Soc, LOS ALAMITOS: ACM, 2020, pp. 436–447, ISBN: 9781450367684.

[8] S. Pittet and Atalassian, *What is code coverage?* [Online]. Available: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.

[9] C. Arguelles, M. Ivanković, and A. Bender. "Code Coverage Best Practices." (2020), [Online]. Available: https://testing.googleblog.com/2020/08/code-coverage-best-practices.html.

[10] N. Uitenbroek, *Safety Critical Flight Software*. [Online]. Available: https://ntrs.nasa.gov/api/citations/20170006507/downloads/20170006507.pdf.

[11] J. Cox, *IV&V Coverage of NASA Software Guidelines*. [Online]. Available: https://www.nasa.gov/wp-content/uploads/2016/10/01-07_ivv_coverage_of_nasa_software_guidelines_0.pdf.

[12] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at Google," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 955–963.

[13] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008. DOI: 10.1109/MS.2008.130.

[14] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic Notes in Theoretical Computer Science*, vol. 217,

pp. 5–21, 2008, Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008), ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2008.06.039. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066108003824.

[15] S. Marathe. "Centralized Version Control Systems (CVCS) and Distributed Version Control Systems (DVCS)." LinkedIn article. (2023), [Online]. Available: https://www.linkedin.com/pulse/centralized-version-control-systems-cvcs-distributed-dvcs-marathe/.

[16] S. Kerola, Stannered, and ReneLeonhardt. "Subversion project visualization image." (2008), [Online]. Available: https://commons.wikimedia.org/wiki/File:Subversion_project_visualization_corrected.svg.

[17] S. I. Feldman, "Make — a program for maintaining computer programs," *Software: Practice and Experience*, vol. 9, no. 4, pp. 255–265, 1979. DOI: https://doi.org/10.1002/spe.4380090402. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380090402. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090402.

[18] *GNU Make*, GNU Operating System. [Online]. Available: https://www.gnu.org/software/make/.

[19] Google Inc., *GoogleTest - Google Testing and Mocking Framework*, 2008. [Online]. Available: https://github.com/google/googletest.

[20] *Fake Function Framework (fff)*. [Online]. Available: https://github.com/meekrosoft/fff.

[21] *cpplint - static code checker for C++*. [Online]. Available: https://github.com/cpplint/cpplint.

[22] Google Inc., *Google C++ Style Guide*. [Online]. Available: https://google.github.io/styleguide/cppguide.html.

[23] *Doxygen*. [Online]. Available: https://www.doxygen.nl/.

[24] *git*. [Online]. Available: https://git-scm.com/.

[25] GitHub, *Understanding github actions*. [Online]. Available: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions.

[26] Docker, Inc., *Docker: Get started*. [Online]. Available: https://docs.docker.com/get-started/.

[27] Docker, Inc., *What is a Container?* [Online]. Available: https://www.docker.com/resources/what-container/.

[28] Intel, *FPGA Basics*. [Online]. Available: https://www.intel.com/content/www/us/en/support/programmable/support-resources/fpga-training/getting-started.html.

[29] Intel, *Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide*. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683521.

[30] A. Fernandes, N. Cruz, B. Santos, *et al.*, "Fpga code for the data acquisition and real-time processing prototype of the iter radial neutron camera," *IEEE Transactions on Nuclear Science*, vol. 66, no. 7, pp. 1318–1323, 2019. DOI: 10.1109/TNS.2019.2903646.

[31] C. Yang, W. Zheng, M. Zhang, T. Yuan, G. Zhuang, and Y. Pan, "A real-time data acquisition and processing framework based on flexrio fpga and iter fast plant system controller," *IEEE Transactions on Nuclear Science*, vol. 63, no. 3, pp. 1715–1719, 2016. DOI: 10.1109/TNS.2016.2542858.

[32] M. Ruiz González, J. Vega, D. Castro, *et al.*, "ITER fast plant system controller prototype based on PXIe platform," in *Proceedings of 8th IAEA Technical Meeting on Control, Data Acquisition, and Remote Participation for Fusion Research*, Holanda: Elsevier, 2011. [Online]. Available: https://oa.upm.es/12992/.

[33] E. Barrera, M. Ruiz, A. Bustos, *et al.*, "Implementation of iter fast plant interlock system using fpgas with compactrio," *IEEE Transactions on Nuclear Science*, vol. 65, no. 2, pp. 796–804, 2018. DOI: 10.1109/TNS.2017.2783243.

[34] M. Ruiz, A. Carpeño, D. Rivilla, M. Astrain, A. Piñas, and V. Costa, "Using fpga-based amc carrier boards for fmc to implement intelligent data acquisition applications in mtca systems using opencl," *IEEE Transactions on Nuclear Science*, vol. 70, no. 6, pp. 993–1000, 2023. DOI: 10.1109/TNS.2023.3255554.

[35] National Instruments, *NI LabVIEW for CompactRIO Developer's Guide - Recommended LabVIEW Architectures and Development Practices for Control and Monitoring Applications*, National Instruments. [Online]. Available: https://www.ni.com/pdf/products/us/fullcriodevguide.pdf.

[36] S. Esquembri Martínez, "Methodology for the integration of high-throughput data and image acquisition systems in EPICS," Ph.D. dissertation, ETSIS Telecomunicacion, 2017. DOI: 10.20868/UPM.thesis.48304. [Online]. Available: https://oa.upm.es/48304/.

[37] T. Hakulinen, F. Valentini, F. Havart, and P. Ninin, "Building an interlock: Comparison of technologies for constructing safety interlocks," 2015.

[38] I2A2, *IRIO Design Rules for LabVIEW FPGA v1.2.0*, Nov. 2016. [Online]. Available: http://www.i2a2.upm.es/wp-content/uploads/2016/11/IRIO_Design_Rules_for_LabVIEW_FPGA_v1.2.0.pdf.

[39] D. S. Chang, S.-K. Kim, and D. W. Lee, "Study of an fpga-based temperature measurement using open source software,"

[40] *CODAC Core System Version 7.2 Release Notes*, ITER, Mar. 2024. [Online]. Available: https://static.iter.org/codac/cs/CODAC_Core_System_Version_7.2_Release_No_9UQ2CL_v1_1.pdf.

[41] S. Esquembri, J. Nieto, M. Ruiz, A. de Gracia, and G. de Arcas, "Methodology for the implementation of real-time image processing systems using fpgas and gpus and their integration in epics using nominal device support," *Fusion Engineering and Design*, vol. 130, pp. 26–31, 2018, ISSN: 0920-3796. DOI: https://doi.org/10.1016/j.fusengdes.2018.02.051. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920379618301522.

[42] M. Astrain, M. Ruiz, A. Carpeño, S. Esquembri, E. Barrera, and J. Vega, "A methodology to standardize the development of fpga-based high-performance daq and processing systems using opencl," *Fusion Engineering and Design*, vol. 155, p. 111561, 2020.

[43] R. Herrero, A. Carpeño, S. Esquembri, M. Ruiz, and E. Barrera, "Fpga-based solutions for analog data acquisition and processing integrated in area detector using flexrio technology," *IEEE Transactions on Nuclear Science*, vol. 65, no. 2, pp. 781–787, 2018. DOI: 10.1109/TNS.2017.2782827.

[44] R. C. Martin, *Clean architecture : a craftsman's guide to software structure and design*, eng, 1st edition. 2018.

[45] *RAII*, cppreference. [Online]. Available: https://en.cppreference.com/w/cpp/language/raii.

[46] National Instruments, *NI System Configuration API Reference for LabWindows™/CVI™*. [Online]. Available: https://www.ni.com/docs/en-US/bundle/ni-system-configuration-c-ref/page/nisyscfgcvi/bp_help_file_title.html.

[47] ITER Organization, *CODAC Core System*, 2024. [Online]. Available: https://www.iter.org/mach/codac/coresystem.

[48] SonarSource, *Code metrics & SonarQube*, 2024. [Online]. Available: https://docs.sonarsource.com/sonarqube/latest/user-guide/code-metrics/metrics-definition/.

[49] G. A. Campbell, "Cognitive complexity: An overview and evaluation," in *Proceedings of the 2018 International Conference on Technical Debt*, ser. TechDebt '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 57–58, ISBN: 9781450357135. DOI: 10.1145/3194164.3194186. [Online]. Available: https://doi.org/10.1145/3194164.3194186.

[50] A. Fertig, *Notebook C++ Tips and Tricks with Templates*, 3. Fertig Publications, Feb. 2024, ISBN: 978-3-949323-07-2. [Online]. Available: https://andreasfertig.com/books/notebookcpp-tips-and-tricks-with-templates/.

[51] A. Fertig, *Notebook C++ About Move Semantics*, 1. Fertig Publications, Sep. 2022, ISBN: 978-3-949323-03-4. [Online]. Available: https://andreasfertig.com/books/notebookcpp-about-move-semantics/.

[52] M. Gregoire, *Professional C++*. Wiley, 2021, ISBN: 9781119695400. [Online]. Available: https://books.google.es/books?id=A_2JzAEACAAJ.

[53] D. Sanz, M. Ruiz, R. Castro, *et al.*, "Advanced Data Acquisition system implementation for the ITER neutron diagnostic use case using EPICS and FlexRIO technology on a PXIe platform," in *2014 19th IEEE-NPSS Real Time Conference*, 2014, pp. 1–2. DOI: 10.1109/RTC.2014.7097486.

[54] E. Barrera, M. Ruiz, D. Sanz, *et al.*, "Test bed for real-time image acquisition and processing systems based on FlexRIO, CameraLink, and EPICS," *Fusion Engineering and Design*, vol. 89, no. 5, pp. 633–637, 2014, Proceedings of the 9th IAEA Technical Meeting on Control, Data Acquisition, and Remote Participation for Fusion Research, ISSN: 0920-3796. DOI: https://doi.org/10.1016/j.fusengdes.2014.02.010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920379614000945.

[55] J. Nieto Valhondo, "Contribuciones en sistemas de adquisición de datos inteligentes para entornos de fusión por confinamiento magnético," Ph.D. dissertation, Telecomunicacion, 2016. DOI: 10.20868/UPM.thesis.39451. [Online]. Available: https://oa.upm.es/39451/.

[56] Á. Bustos Benayas, "CompactRIO: advanced data acquisition systems integration in CODAC Core System," M.S. thesis, ETSIS Telecomunicacion, Jul. 2015. [Online]. Available: https://oa.upm.es/38064/.

[57] D. Sanz, M. Ruiz, R. Castro, *et al.*, "Implementation of intelligent data acquisition systems for fusion experiment using EPICS and FlexRIO technology," in *2012 18th IEEE-NPSS Real Time Conference*, 2012, pp. 1–8. DOI: 10.1109/RTC.2012.6418166.

[58] I2A2, *IRIO EPICS Device Support User Manual v1.2.0*, Nov. 2016. [Online]. Available: http://www.i2a2.upm.es/wp-content/uploads/2016/11/IRIO_EPICS_Device_Support_User_Manual_v1.2.0.pdf.

[59] I2A2, *IRIO-v2 Repository*. [Online]. Available: https://github.com/i2a2/irio-v2.

[60] National Instruments, *FPGA Fundamentals: Basics of Field-Programmable Gate Arrays*. [Online]. Available: https://www.ni.com/en/shop/electronic-test-instrumentation/add-ons-for-electronic-test-and-instrumentation/what-is-labview-fpga-module/fpga-fundamentals.html.

[61] National Instruments, *The LabVIEW RIO Architecture: A Foundation for Innovation*, Jan. 2024. [Online]. Available: http://www.ni.com/white-paper/10894/en/.

[62] National Instruments, *NI R Series Multifunction RIO User Manual*, Oct. 2023. [Online]. Available: https://www.ni.com/docs/en-US/bundle/multifunction-rio-user-manual/resource/multifunction-rio-user-manual.pdf.

[63] National Instruments, *NI R Series Multifunction RIO Specifications*, Oct. 2023. [Online]. Available: https://www.ni.com/docs/en-US/bundle/multifunction-rio-specifications/resource/multifunction-rio-specifications.pdf.

[64] National Instruments, *NI LabVIEW for CompactRIO Developer's Guide*, 2014. [Online]. Available: https://www.ni.com/pdf/products/us/fullcriodevguide.pdf.

[65] National Instruments. "What Is FlexRIO?" (2023), [Online]. Available: https://www.ni.com/en/shop/electronic-test-instrumentation/flexrio/what-is-flexrio.html.

[66] National Instruments. "FlexRIO Custom Instrumentation." (2023), [Online]. Available: https://www.ni.com/pdf/product-flyers/flexrio-custom-instrumentation.pdf.

[67] National Instruments. "NI FlexRIO FPGA Module Installation Guide and Specifications." (2011), [Online]. Available: https://www.ni.com/docs/en-US/bundle/ni-flexrio-fpga-module-getting-started/resource/373047b.pdf.

[68] National Instruments. "FlexRIO Modular Help." (Feb. 2023), [Online]. Available: https://www.ni.com/docs/en-US/bundle/flexrio/page/flexriomodularmerge/flexriomodular_help.html.

[69] R. C. Martin, *Design principles and design patterns*.

[70] R. C. Martin, *The clean coder : a code of conduct for professional programmers*, eng, 1st edition. 2011, ISBN: 0-13-254295-1.

[71] *Design patterns : elements of reusable object-oriented software* (Addison-Wesley professional computing series), eng. Boston [etc.]: Addison-Wesley, 1995, ISBN: 0-201-63361-2.

[72] F. G. Pikus, *Hands-on design patterns with C++ : solve common C++ problems with modern design patterns and build robust applications*, eng, Second edition. 2019, ISBN: 1-80461-727-X.

Appendix

Results Coverage Through The Project's Commits

Index	Commit	Line Cov.	Fun. Cov.	Total lines	Total Fun.
1	2f31642	0.0%	0.0%	0	0
2	adca403	0.0%	0.0%	0	0
3	799dadd	0.0%	0.0%	0	0
4	bfce6ed	0.0%	0.0%	0	0
5	731281c	75.6%	52.4%	127	21
6	e898278	75.6%	52.4%	127	21
7	777ef31	75.6%	52.4%	127	21
8	deb67f1	74.6%	50.0%	130	22
9	1791fc0	74.6%	50.0%	130	22
10	f755654	45.1%	27.5%	215	40
11	20717df	45.1%	27.5%	215	40
12	11e8905	45.1%	27.5%	215	40
13	e4ae8bd	45.1%	27.5%	215	40
14	a49ba37	41.5%	25.0%	234	44
15	0ca21f3	35.1%	20.8%	276	53
16	541a383	35.1%	20.8%	276	53
17	145cc15	34.5%	19.6%	281	56
18	f9a0e04	34.2%	19.3%	284	57
19	8a27b97	31.8%	18.0%	305	61
20	e46e3f6	29.6%	16.7%	328	66
21	3ed9d34	27.6%	15.5%	351	71
22	ac07f96	27.6%	15.5%	351	71
23	b40db70	27.6%	15.5%	351	71
24	89780ef	27.6%	15.5%	351	71

Continued on next page

25	41c555c	26.4%	14.9%	367	74
26	5d05f73	25.5%	14.3%	380	77
27	763d9eb	25.3%	13.9%	384	79
28	bde84fd	25.3%	13.9%	384	79
29	98c3168	21.2%	11.6%	458	95
30	2986a67	21.2%	11.6%	458	95
31	475cf66	21.4%	11.6%	459	95
32	7535b9a	21.4%	11.6%	459	95
33	e2f9bcb	21.4%	11.6%	459	95
34	c63f743	21.4%	11.6%	459	95
35	5433428	20.7%	11.1%	474	99
36	21c7a4f	20.4%	11.0%	480	100
37	b984b1d	20.4%	11.0%	480	100
38	80e2b83	20.4%	11.0%	480	100
39	34462fb	0.0%	0.0%	0	0
40	eaa8eef	0.0%	0.0%	0	0
41	ffdf37d	0.0%	0.0%	0	0
42	9677336	0.0%	0.0%	0	0
43	3f15258	0.0%	0.0%	0	0
44	66ecc87	0.0%	0.0%	0	0
45	fcea702	0.0%	0.0%	0	0
46	b6fb24f	0.0%	0.0%	0	0
47	2432ab6	0.0%	0.0%	0	0
48	8657358	0.0%	0.0%	0	0
49	0f4b309	0.0%	0.0%	0	0
50	9d6bb71	0.0%	0.0%	0	0
51	a2df3fe	0.0%	0.0%	0	0
52	47e7b3c	0.0%	0.0%	0	0
53	514feca	0.0%	0.0%	0	0
54	4a2b559	0.0%	0.0%	0	0
55	5fe8780	0.0%	0.0%	0	0
56	f4e8922	0.0%	0.0%	0	0
57	f397540	0.0%	0.0%	0	0
58	bb0cb6d	0.0%	0.0%	0	0
59	b2c8a43	0.0%	0.0%	0	0

Continued on next page

(Continued)

60	74b5fe0	45.2%	39.1%	538	110
61	4b9a633	44.4%	37.4%	549	115
62	367182c	44.6%	37.4%	552	115
63	f76b561	44.6%	37.4%	552	115
64	75b8233	44.0%	37.4%	564	115
65	4b927cd	44.0%	37.4%	564	115
66	8747e10	44.0%	37.4%	568	115
67	f51183d	44.0%	37.4%	568	115
68	70a760e	44.0%	37.4%	568	115
69	9427fac	38.7%	29.1%	576	117
70	581c47c	38.7%	28.9%	579	121
71	cfb803b	38.7%	28.9%	579	121
72	8535977	38.0%	27.8%	589	126
73	b25cfd3	37.5%	27.3%	597	128
74	27a19cd	44.6%	36.4%	605	129
75	dc12fd1	19.2%	11.5%	613	131
76	842399f	19.2%	11.5%	613	131
77	26d85f2	19.0%	11.2%	622	134
78	0e793d9	18.6%	10.7%	634	140
79	a601505	18.6%	10.7%	634	140
80	926a6da	18.6%	10.7%	634	140
81	a55b627	18.6%	10.7%	634	140
82	122d7aa	55.4%	48.6%	686	148
83	b5b75c3	55.1%	49.0%	690	147
84	d77a247	55.1%	49.0%	690	147
85	81cbb99	55.1%	49.0%	690	147
86	986d928	55.1%	49.0%	690	147
87	cda287f	55.1%	49.0%	690	147
88	2f3e3f0	53.5%	49.0%	708	147
89	caf9f91	52.9%	45.6%	717	158
90	f9c80c0	52.9%	45.6%	717	158
91	78d756c	52.9%	45.6%	717	158
92	3434cc1	52.9%	45.6%	717	158
93	e94b554	52.9%	45.6%	717	158
94	b3c62f8	52.9%	45.6%	717	158

Continued on next page

(Continued)

95	0ba3cb3	52.9%	45.6%	717	158
96	cf75e28	52.9%	45.6%	717	158
97	0e74b70	52.9%	45.6%	717	158
98	6f2be0c	52.9%	45.6%	717	158
99	8a19220	52.9%	45.6%	717	158
100	4cef6cf	52.9%	45.6%	717	158
101	257bbf8	52.9%	45.6%	717	158
102	f63c0c8	52.9%	45.6%	717	158
103	99689ae	52.9%	45.6%	717	158
104	140522e	52.9%	45.6%	717	158
105	34e1ce7	52.9%	45.6%	717	158
106	999dc3f	52.9%	45.6%	717	158
107	9c5d491	41.7%	37.5%	908	192
108	2da2cf8	41.6%	37.3%	911	193
109	c0749d8	41.7%	37.5%	909	192
110	df87cb7	46.5%	41.8%	916	194
111	36baab1	46.5%	41.8%	916	194
112	82a1610	46.6%	41.5%	919	193
113	f8fd951	46.5%	41.2%	921	194
114	67df112	46.5%	41.2%	920	194
115	a61ae98	46.5%	41.2%	920	194
116	c0b821f	46.4%	39.8%	915	201
117	3d9edc5	46.4%	39.8%	915	201
118	3f6a387	0.0%	0.0%	0	0
119	feffebc	0.0%	0.0%	0	0
120	02e8094	47.8%	41.3%	920	201
121	ff1eb0d	47.1%	41.3%	916	201
122	373f4d2	47.1%	41.3%	916	201
123	0ad2228	48.8%	43.3%	916	201
124	09efdb2	48.9%	43.6%	917	202
125	ad32af9	48.9%	43.6%	917	202
126	8204f1c	54.6%	48.0%	917	202
127	c4c230a	54.9%	48.3%	913	201
128	f7fc9b6	53.7%	47.1%	937	206
129	02d43e2	56.0%	49.5%	937	206

Continued on next page

(Continued)

130	62a04d0	56.0%	49.5%	937	206
131	6f7c159	0.0%	0.0%	0	0
132	590b974	0.0%	0.0%	0	0
133	08c311a	0.0%	0.0%	0	0
134	d5cf5cd	58.9%	52.9%	936	206
135	0e34041	58.9%	52.9%	936	206
136	b218e7c	58.8%	52.9%	935	206
137	611683f	58.8%	52.9%	935	206
138	deb6d49	58.8%	52.9%	935	206
139	e1b23b3	59.5%	53.4%	935	206
140	a352597	0.0%	0.0%	0	0
141	83d5159	0.0%	0.0%	0	0
142	260a3e1	59.5%	53.4%	935	206
143	f542986	59.5%	53.4%	935	206
144	f617264	68.4%	62.1%	935	206
145	21306db	68.4%	62.1%	935	206
146	cce85eb	68.3%	62.1%	935	206
147	d0d5eeb	64.9%	59.0%	1041	229
148	9f07a53	64.9%	59.0%	1041	229
149	c94326b	64.9%	59.0%	1041	229
150	48cbf12	0.0%	0.0%	0	0
151	8fcec78	64.9%	58.7%	1041	230
152	df5ef82	68.2%	65.2%	1041	230
153	6947d57	68.2%	65.2%	1042	230
154	14621a7	68.3%	65.2%	1044	230
155	14032ea	68.3%	65.2%	1044	230
156	84d4dfb	68.5%	66.1%	1038	224
157	860b2c1	0.0%	0.0%	0	0
158	97dce46	0.0%	0.0%	0	0
159	31bc91f	68.5%	66.1%	1038	224
160	4a1ce8c	68.5%	66.1%	1038	224
161	0d0b5bc	68.5%	66.1%	1038	224
162	0490197	67.8%	65.2%	1256	313
163	6cc2afa	67.8%	65.2%	1256	313
164	c74dae0	67.8%	65.2%	1256	313

Continued on next page

(Continued)

165	aee60da	67.8%	65.2%	1256	313
166	4f32360	67.7%	64.9%	1241	308
167	074eb77	67.7%	64.9%	1241	308
168	d6755a6	67.7%	64.9%	1241	308
169	67ca7bc	67.7%	64.9%	1241	308
170	9bb5fd2	67.8%	64.9%	1239	308
171	7a461a4	67.9%	65.0%	1242	309
172	d4199a0	67.9%	65.0%	1242	309
173	dc51a33	67.9%	65.0%	1242	309
174	a419b00	67.9%	65.0%	1242	309
175	4a784ed	67.9%	65.0%	1242	309
176	ca05065	67.9%	65.0%	1242	309
177	21a189e	67.9%	65.0%	1242	309
178	629c781	67.9%	65.0%	1242	309
179	59c8493	67.9%	65.0%	1242	309
180	01b5ca9	67.9%	65.0%	1242	309
181	b927982	67.9%	65.0%	1242	309
182	4b56db4	67.9%	65.0%	1242	309
183	fc37636	67.9%	65.0%	1242	309
184	07b1bc8	67.9%	65.0%	1242	309
185	d5bb185	67.9%	65.0%	1242	309
186	6d6bd6d	67.9%	65.0%	1242	309
187	1580917	67.9%	65.0%	1242	309
188	61aad87	67.9%	65.0%	1242	309
189	285ef47	0.0%	0.0%	0	0
190	d597e13	0.0%	0.0%	0	0
191	ce1c5f2	67.7%	64.6%	1248	311
192	a19804f	67.7%	64.6%	1248	311
193	4d57262	67.6%	64.6%	1247	311
194	1ae35ba	67.6%	64.6%	1247	311
195	6ed2065	67.6%	64.6%	1247	311
196	c6b60e7	0.0%	0.0%	0	0
197	decf68f	0.0%	0.0%	0	0
198	0f1ea03	0.0%	0.0%	0	0
199	50f148e	0.0%	0.0%	0	0

Continued on next page

(Continued)

200	81e313c	0.0%	0.0%	0	0
201	dfb0a78	0.0%	0.0%	0	0
202	0f11de6	0.0%	0.0%	0	0
203	df5c1c4	67.6%	64.6%	1247	311
204	e555416	67.6%	64.6%	1247	311
205	ba0ecc8	67.6%	64.6%	1247	311
206	df3a00d	0.0%	0.0%	0	0
207	0ec9359	0.0%	0.0%	0	0
208	b28d1f0	0.0%	0.0%	0	0
209	6884180	79.2%	79.7%	1417	311
210	fd7e019	79.2%	79.7%	1417	311
211	b626e62	79.9%	81.0%	1418	311
212	bf719e3	79.9%	81.0%	1418	311
213	a376fe1	79.9%	81.0%	1418	311
214	6f5de45	79.3%	80.4%	1417	311
215	21fa885	79.4%	80.4%	1416	311
216	c6dd827	79.4%	80.4%	1416	311
217	4be5b47	79.3%	80.4%	1417	311
218	5018f1d	79.4%	80.4%	1414	311
219	50cc0ad	79.4%	80.4%	1414	311
220	99de110	79.2%	79.7%	1417	311
221	789a92c	79.4%	80.4%	1414	311
222	f509784	79.6%	80.4%	1411	311
223	90b3ca6	0.0%	0.0%	0	0
224	bd6b3af	0.0%	0.0%	0	0
225	dd7a638	0.0%	0.0%	0	0
226	4ba162c	0.0%	0.0%	0	0
227	60c55d4	0.0%	0.0%	0	0
228	24d553d	93.6%	92.9%	1402	308
229	4cef869	93.6%	92.9%	1401	308
230	467d03f	93.5%	92.9%	1403	308
231	c4dfdfd	93.5%	92.9%	1403	308
232	1a1edd2	93.6%	92.9%	1402	308
233	dc222fc	93.5%	92.9%	1403	308
234	fab046e	93.6%	92.9%	1402	308

Continued on next page

(Continued)

235	9452c55	93.5%	92.9%	1403	308
236	d6fa636	93.6%	92.9%	1402	308
237	6732d85	93.5%	92.9%	1403	308
238	00be1d5	96.5%	94.9%	1417	313
239	e02f29e	96.5%	94.9%	1417	313
240	d7b0225	96.5%	94.9%	1417	313
241	6d49aed	97.1%	95.9%	1403	316
242	fd3c4d8	97.1%	95.9%	1403	316
243	dab2fba	97.1%	95.9%	1403	316
244	10e96fa	97.1%	95.9%	1403	316
245	b926e1a	97.1%	95.9%	1403	316
246	b7fe99f	97.1%	95.9%	1403	316
247	5ce59bb	97.1%	95.9%	1403	316
248	f1de1ab	96.9%	95.3%	1407	318
249	6f4890e	96.9%	95.3%	1407	318
250	b41253e	96.9%	95.3%	1407	318
251	557893c	96.5%	94.7%	1412	320
252	9ccba42	96.5%	94.7%	1413	320
253	8d776b4	97.2%	95.9%	1413	320
254	962d772	97.8%	96.6%	1413	320
255	31c54fd	97.8%	96.6%	1416	320
256	334f420	97.8%	96.6%	1416	320
257	4732fba	97.8%	96.6%	1416	320
258	dc20844	0.0%	0.0%	0	0
259	b1d1d98	0.0%	0.0%	0	0
260	3a7843e	0.0%	0.0%	0	0
261	4bcfac0	0.0%	0.0%	0	0
262	4dd76c3	0.0%	0.0%	0	0
263	6d4e3c3	0.0%	0.0%	0	0
264	e72e645	0.0%	0.0%	0	0
265	039927b	95.1%	97.7%	2255	472
266	5538d7e	95.1%	97.7%	2255	472
267	f81f02e	95.1%	97.7%	2255	472
268	78a8ea6	95.1%	97.7%	2256	472
269	bfbf019	95.1%	97.7%	2255	472

Continued on next page

(Continued)

270	e5dca26	95.1%	97.7%	2255	472
271	a9bf43c	95.6%	97.9%	2268	473
272	ac3a54c	95.7%	97.9%	2324	476
273	ad4b9f2	95.7%	97.9%	2325	476
274	cf4307a	95.5%	97.9%	2325	476
275	e991dbf	0.0%	0.0%	0	0
276	f5a28a7	0.0%	0.0%	0	0
277	bd30323	0.0%	0.0%	0	0
278	487ddf7	0.0%	0.0%	0	0
279	c57471f	0.0%	0.0%	0	0
280	ed4c477	0.0%	0.0%	0	0
281	013ce48	0.0%	0.0%	0	0
282	0e31c82	0.0%	0.0%	0	0
283	893c9cc	0.0%	0.0%	0	0
284	c1f76c8	0.0%	0.0%	0	0
285	91ec068	0.0%	0.0%	0	0
286	dc3f76e	0.0%	0.0%	0	0
287	314bd53	0.0%	0.0%	0	0
288	05c3f8b	0.0%	0.0%	0	0
289	470fe67	0.0%	0.0%	0	0
290	075d55e	0.0%	0.0%	0	0
291	94e12ce	0.0%	0.0%	0	0
292	e1ea90c	0.0%	0.0%	0	0
293	d3b7f58	0.0%	0.0%	0	0
294	674c0a1	0.0%	0.0%	0	0
295	5cca38d	0.0%	0.0%	0	0
296	13440cb	0.0%	0.0%	0	0
297	e6e4d39	0.0%	0.0%	0	0
298	5a20b35	0.0%	0.0%	0	0
299	75c0ea5	0.0%	0.0%	0	0
300	414fcd9	0.0%	0.0%	0	0
301	9fb87b7	0.0%	0.0%	0	0
302	bb52c2f	0.0%	0.0%	0	0
303	713acc0	0.0%	0.0%	0	0
304	2990c68	0.0%	0.0%	0	0

Continued on next page

(Continued)

305	c2efccf	0.0%	0.0%	0	0
306	562fdef	0.0%	0.0%	0	0
307	e1bdf51	95.3%	97.9%	2334	475
308	43538b6	95.3%	97.9%	2335	475
309	f5c2f06	87.5%	93.1%	2335	475
310	3a4718c	87.5%	93.1%	2335	475
311	c92f638	95.3%	97.9%	2335	475
312	4269721	95.3%	97.9%	2335	475
313	13888ee	95.3%	97.9%	2335	475
314	2820408	95.3%	97.9%	2335	475
315	38f54a3	95.3%	97.9%	2335	475
316	9253bb6	95.3%	97.9%	2335	475
317	bf591ea	94.5%	97.6%	2493	505
318	dde680e	94.5%	97.6%	2493	505
319	fadb8f7	94.5%	97.6%	2493	505
320	9469ec2	86.8%	92.3%	2713	534
321	5684aad	86.3%	91.6%	2729	538
322	7ff80e0	86.3%	91.6%	2729	538
323	2d7cb12	86.1%	91.6%	2735	538
324	644a6ad	95.1%	97.8%	2735	538
325	47d4265	95.0%	97.8%	2736	538
326	5ec7200	92.3%	94.8%	2817	555
327	55c3ace	92.1%	94.4%	2825	558
328	8f8c7cb	91.9%	94.4%	2830	558
329	18100d3	91.9%	94.4%	2830	558
330	1e0c8e2	91.9%	94.4%	2830	558
331	d0acd70	91.9%	94.4%	2830	558
332	9ac61af	95.1%	97.5%	2830	558
333	5922075	95.1%	97.5%	2846	565
334	38bdfec	95.1%	97.5%	2848	565
335	6b0d53d	95.2%	97.6%	2897	577
336	7abecf8	95.2%	97.6%	2897	577
337	60e5f3b	95.2%	97.6%	2897	577
338	348c400	95.2%	97.6%	2897	577
339	99ae2ad	95.2%	97.6%	2897	577

Continued on next page

(Continued)

340	12aab3f	95.2%	97.6%	2897	577
341	b47e294	95.2%	97.6%	2897	577
342	af9c2db	95.2%	97.6%	2897	577
343	d8087fd	95.2%	97.6%	2906	577
344	9cfbb68	95.2%	97.6%	2912	578
345	3ef6c44	95.2%	97.6%	2912	578
346	2f131dc	95.2%	97.6%	2920	578
347	150cb0f	95.2%	97.6%	2929	579
348	ce569b3	95.2%	97.6%	2932	579
349	4cfd599	95.0%	97.6%	2932	579
350	d1db96d	95.0%	97.6%	2926	580
351	38f5882	0.0%	0.0%	0	0
352	d24a9b8	0.0%	0.0%	0	0
353	e38c981	0.0%	0.0%	0	0
354	ad17954	95.4%	97.9%	2926	580
355	f9ff72a	95.4%	97.9%	2926	580
351	38f5882	0.0%	0.0%	0	0
352	d24a9b8	0.0%	0.0%	0	0
353	e38c981	0.0%	0.0%	0	0
354	ad17954	95.4%	97.9%	2926	580
355	f9ff72a	95.4%	97.9%	2926	580
356	9f37cc1	0.0%	0.0%	0	0
351	38f5882	0.0%	0.0%	0	0
352	d24a9b8	0.0%	0.0%	0	0
353	e38c981	0.0%	0.0%	0	0
354	ad17954	95.4%	97.9%	2926	580
355	f9ff72a	95.4%	97.9%	2926	580
356	9f37cc1	95.2%	97.6%	2959	582
357	80eaa4c	95.2%	97.6%	2959	582
358	c22f378	95.2%	97.6%	2959	582
359	e405875	95.2%	97.6%	2959	582
360	70764e2	95.2%	97.6%	2959	582
361	56ff4ce	95.2%	97.6%	2959	582
362	d9321bd	95.2%	97.6%	2959	582
363	56d183a	95.2%	97.6%	2959	582

Continued on next page

(Continued)

364	b4167a0	95.2%	97.6%	2959	582
365	55b7c8f	95.2%	97.6%	2959	582
366	32bfd9a	95.2%	97.6%	2959	582
367	9cdc76f	95.2%	97.6%	2959	582
368	8d56266	95.2%	97.6%	2959	582
369	8615eb2	95.2%	97.6%	2959	582
370	589f52d	95.2%	97.6%	2959	582
371	e7a6626	95.2%	97.6%	2959	582
372	420bc1d	95.2%	97.6%	2959	582
373	0c4c83e	95.2%	97.6%	2959	582
374	b61f8d4	95.2%	97.6%	2959	582
375	0a46f4a	95.2%	97.6%	2959	582
376	c627020	95.2%	97.6%	2959	582
377	e7270f0	95.2%	97.6%	2959	582
378	4084e61	95.2%	97.6%	2959	582
379	bec0e69	95.2%	97.6%	2959	582
380	1f5f67e	95.2%	97.6%	2959	582
381	deae55b	95.2%	97.6%	2959	582
382	2443efe	95.2%	97.6%	2959	582
383	32a0dbe	95.2%	97.6%	2959	582
384	6b37b9c	95.2%	97.6%	2959	582
385	ac25a68	95.2%	97.6%	2959	582
386	3ae5a07	95.2%	97.6%	2959	582
387	43dba9e	95.2%	97.6%	2959	582
388	c8254cf	96.5%	94.9%	1417	313
389	481474c	95.2%	97.6%	2959	582
390	a0f3f7a	95.2%	97.6%	2959	582
391	0def186	95.2%	97.6%	2959	582
392	4ecf8b9	95.2%	97.6%	2959	582
393	0681011	95.2%	97.6%	2959	582
394	8088468	95.2%	97.6%	2959	582
395	708c194	95.2%	97.6%	2959	582
396	c2669e6	95.2%	97.6%	2959	582
397	404dfa8	95.2%	97.6%	2959	582
398	7a6d257	95.2%	97.6%	2959	582

Continued on next page

(Continued)

399	e21d279	95.2%	97.6%	2959	582
400	916ee31	95.2%	97.6%	2959	582
401	81b2eef	95.2%	97.6%	2959	582
402	ba793cb	95.2%	97.6%	2959	582
403	15aab87	95.2%	97.6%	2959	582
404	a7a6fa1	95.2%	97.6%	2959	582
405	c61f787	95.2%	97.6%	2959	582
406	ed4ca14	95.2%	97.6%	2959	582
407	397510b	95.2%	97.6%	2959	582
408	aa2249d	95.2%	97.6%	2959	582
409	d1f1033	95.2%	97.6%	2959	582
410	3711369	95.2%	97.6%	2959	582
411	abb85d7	95.2%	97.6%	2959	582
412	11f6182	95.2%	97.6%	2959	582
413	9ebd9ff	95.2%	97.6%	2959	582
414	a1906a4	95.2%	97.6%	2959	582
415	5df5784	95.2%	97.6%	2959	582
416	87980c7	95.2%	97.6%	2959	582
417	b7ac95a	95.1%	97.6%	2964	582
418	77f7e3d	95.1%	97.6%	2964	582
419	45f6b43	95.2%	97.6%	2978	582
420	950339b	95.2%	97.6%	2978	582
421	0fd718b	95.2%	97.6%	2978	582
422	c13b0d8	95.2%	97.6%	2978	582
423	c7a7371	95.2%	97.6%	2978	582
424	e14e304	95.2%	97.6%	2980	582
425	d64e0fa	95.2%	97.6%	2980	582
426	488a4ad	95.2%	97.6%	2959	582
427	9d8e47b	95.2%	97.6%	2959	582
428	a412d68	95.2%	97.6%	2980	582
429	17be0d8	95.5%	97.6%	2972	582
430	15cc8c9	95.2%	97.6%	2959	582
431	c0d5904	95.2%	97.6%	2959	582
432	8e59c68	95.5%	97.6%	2971	582
433	5269784	95.2%	97.6%	2959	582

Continued on next page

(Continued)

434	ed0af50	95.5%	97.6%	2971	582
435	1a7e8ef	95.5%	97.6%	2969	582
436	952f4c3	95.5%	97.6%	2969	582
437	fe363c9	95.5%	97.6%	2970	582
438	9ad9296	95.5%	97.6%	2970	582
439	d3bcceb	95.5%	97.6%	2970	582
440	fe8c946	95.5%	97.6%	2970	582
441	9ac2678	95.5%	97.6%	2970	582
442	f7ecddc	95.5%	97.6%	2970	582
443	65e714f	95.5%	97.6%	2970	582
444	31ee955	95.5%	97.6%	2970	582
445	018fedd	95.5%	97.6%	2970	582
446	792eea0	95.5%	97.6%	2970	582
447	ea0cd0d	95.5%	97.6%	2970	582
448	6ff7c03	95.5%	97.6%	2970	582
449	c894364	95.5%	97.6%	2970	582
450	cbccdc6	95.5%	97.6%	2970	582
451	9a71f50	95.5%	97.6%	2970	582
452	49afbdb	95.5%	97.6%	2970	582
453	87d6da7	95.5%	97.6%	2970	582
454	2101589	95.5%	97.6%	2970	582
455	32ed75b	95.5%	97.6%	2970	582
456	365c817	95.5%	97.6%	2970	582
457	95d5390	95.5%	97.6%	2970	582
458	e87b021	95.5%	97.6%	2970	582
459	79db606	95.5%	97.6%	2970	582
460	bfa52ef	95.5%	97.6%	2970	582
461	1be9b90	95.5%	97.6%	2970	582
462	9ad8462	95.5%	97.6%	2970	582
463	27a850c	95.5%	97.6%	2970	582
464	d1d7188	95.5%	97.6%	2970	582
465	a20125b	95.5%	97.6%	2970	582
466	65b715f	95.5%	97.6%	2970	582
467	0e71050	95.6%	97.7%	3023	599
468	6286753	95.5%	97.5%	3026	600

Continued on next page

(Continued)

469	77dfcb2	95.5%	97.5%	3026	600
470	b364833	95.5%	97.5%	3026	600
471	bacf45b	95.5%	97.5%	3026	600
472	e0e66e2	95.5%	97.5%	3026	600
473	b290e1d	95.5%	97.5%	3026	600
474	ae7a235	95.5%	97.5%	3026	600
475	0e7dd47	95.5%	97.5%	3026	600
476	cad7d9f	95.5%	97.5%	3026	600
477	ca2050b	95.5%	97.5%	3026	600
478	adf3655	95.7%	97.5%	3026	600
479	58db397	95.2%	97.4%	3103	614
480	f40069b	95.9%	97.7%	3103	614
481	1e60685	95.9%	97.7%	3103	614
482	1c92da4	95.9%	97.7%	3103	614
483	da107b8	95.9%	97.7%	3103	614
484	20067d7	95.9%	97.7%	3103	614
485	0a06f44	95.9%	97.7%	3103	614
486	71d8c09	95.9%	97.7%	3103	614
487	808ae50	95.9%	97.7%	3103	614
488	e363fae	95.9%	97.7%	3103	614
489	9bdc2ed	95.9%	97.7%	3103	614
490	284679d	95.9%	97.4%	3107	616
491	4968ff4	95.9%	97.4%	3107	616
492	b158b3e	95.9%	97.4%	3107	616
493	b65a27a	95.9%	97.4%	3107	616
494	2934753	95.9%	97.4%	3107	616
495	f760fba	95.9%	97.4%	3107	616
496	21cdc5c	95.9%	97.4%	3107	616
497	ce3b14c	95.9%	97.4%	3110	617
498	3d92b66	95.8%	97.4%	3110	617
499	8422f78	95.8%	97.4%	3110	617
500	d0e321a	95.8%	97.4%	3110	617
501	75fc9ff	95.8%	97.4%	3110	617
502	f2f4543	0.0%	0.0%	0	0
503	5f34119	0.0%	0.0%	0	0

Continued on next page

(Continued)

504	3021ae5	0.0%	0.0%	0	0
505	8660863	0.0%	0.0%	0	0
506	3de4196	95.8%	97.4%	3110	617
507	2483e7f	0.0%	0.0%	0	0
508	158795e	95.8%	97.4%	3110	617
509	a817d7c	95.7%	97.4%	3114	617
510	8614d1f	95.7%	97.4%	3114	617
511	6391a31	95.5%	97.1%	3114	617
512	5e1324b	95.6%	97.1%	3114	617
513	375b970	95.5%	97.1%	3114	617
514	ece3980	95.5%	97.1%	3114	617
515	9c229c7	95.6%	97.1%	3114	617
516	6a09081	95.7%	97.1%	3114	617
517	ed4a9f4	95.7%	97.1%	3114	617
518	c8d57f5	95.7%	97.1%	3114	617
519	f1d3d2c	95.7%	97.1%	3114	617
520	5658f32	95.7%	97.1%	3114	617
521	01e8b71	95.7%	97.1%	3114	617
522	19cdbc6	95.7%	97.1%	3114	617
523	9a8d252	95.7%	97.1%	3107	618
524	d34f204	95.7%	97.1%	3108	619
525	9b7494a	96.1%	98.7%	3095	606
526	56e2f8d	96.1%	98.7%	3094	606
527	bde46a0	96.6%	99.2%	3094	606
528	9539c39	96.8%	99.2%	3094	606
529	6c76790	96.8%	99.2%	3094	606
530	2ebbb23	96.8%	99.2%	3094	606
531	7dec17e	96.8%	99.2%	3094	606
532	1795e52	96.8%	99.2%	3094	606
533	87be14b	96.8%	99.2%	3094	606
534	f621ac8	96.8%	99.2%	3094	606
535	ab76126	96.8%	99.2%	3094	606
536	e74e606	96.8%	99.2%	3094	606
537	58a6b93	96.8%	99.2%	3094	606
538	5a7d5f2	96.8%	99.2%	3094	606

Continued on next page

(Continued)

539	99334cc	96.8%	99.2%	3094	606
540	0d491ea	96.8%	99.2%	3094	606
541	46d22e5	96.8%	99.2%	3094	606
542	cab0235	96.8%	99.2%	3094	606
543	ae5cfbc	96.8%	99.2%	3094	606
544	a79d079	96.8%	99.2%	3098	607
545	c439cbe	96.8%	99.2%	3098	607
546	800449f	96.8%	99.2%	3098	607
547	1c15c35	96.8%	99.2%	3098	607
548	1c89592	96.8%	99.2%	3098	607
549	5b9d376	96.8%	99.2%	3098	607
550	c84da54	96.8%	99.2%	3098	607
551	3c96cad	96.8%	99.2%	3098	607
552	2fc0c1f	96.8%	99.2%	3098	607
553	2eb4002	96.8%	99.2%	3098	607
554	6ff677a	96.8%	99.2%	3098	607
555	830b703	96.8%	99.2%	3094	606
556	43866d7	96.8%	99.2%	3098	607
557	320d47e	96.8%	99.2%	3098	607
558	36b757c	96.8%	99.2%	3098	607
559	3db191c	96.8%	99.2%	3098	607
560	0c87c64	96.9%	99.2%	3098	607
561	155b81b	96.9%	99.2%	3098	607