Design and implementation of a new Controller for CERN's sub-ppm stable Current source

Tesi di Laurea Magistrale in
Electronics Engineering - Ingegneria Elettronica

Author: **Valerio Nappi**

Student ID: 970674
Advisor: Prof. Federica Villa
Co-advisors: Miguel Cerqueira Bastos, Slawosz Uznanksi, Adrian Byszuk
Academic Year: 2022-23

Abstract

This work describes the design and development of the controller of the CERN DCTT Calibrator (CDC), a critical component in the power converter controls system in CERN's facilities. The CDC is responsible for providing an accurate and reliable current that can be used to calibrate current sensors or other devices. The development of the controller of such an instrument is described in the following pages, following a hardware upgrade that was carried out in the recent past.

The research begins with an introduction to CERN and the LHC, setting the stage for understanding the significance of the CDC within this context. The thesis then explores the CDC's working principles, its recent upgrade, and the control system requirements for its operation. Special attention is given to the explanation of the working principle of DCCTs (Direct-Current Current Transformers) and the stringent calibration requirements they must meet.

A key focus of the thesis is the hardware and software aspects of the CDC. It presents a comparative study of different system-on-chip (SoC) solutions, highlighting the adoption of the Zynq system and the use of FPGA (Field-Programmable Gate Array) technology. The thesis also addresses the application of Continuous Integration/Continuous Deployment (CI/CD) methodologies in FPGA development, leading to a leap in development efficiency and reliability.

The software development section describes the Petalinux operating system and the development of the Python application and user-space drivers.

Lastly, the thesis presents application and performance testing methodologies, including continuous integration and deployment strategies, to ensure the CDC's robustness and reliability. The performance assessment section provides an overview on the challenging metrological validation of the instrument's performance and the results of the tests carried out.

Keywords: CERN, current source, calibration, Zynq, DCCT, FPGA

Abstract in lingua italiana

Questo lavoro descrive la progettazione e lo sviluppo del controller CERN DCCT Calibrator (CDC), un componente cruciale nel sistema di controllo dei convertitori di potenza nell'infrastruttura del CERN. Il CDC è responsabile della fornitura di una corrente di riferimento precisa e affidabile che può essere utilizzata per calibrare sensori di corrente e altri dispositivi. Le seguenti pagine descrivono lo sviluppo del controller di tale strumento, a seguito di un aggiornamento hardware che è stato effettuato di recente.

La ricerca inizia con un'introduzione al CERN e al LHC, preparando il terreno per comprendere l'importanza del CDC in questo contesto. La tesi esplora quindi i principi di funzionamento del CDC, il suo recente aggiornamento e i requisiti del sistema di controllo per il suo funzionamento. Particolare attenzione viene dedicata alla spiegazione del principio di funzionamento dei DCCT (Trasformatori di Corrente a Corrente Diretta) e ai rigorosi requisiti di calibrazione che devono soddisfare.

Un focus chiave della tesi è sugli aspetti hardware e software del CDC. Presenta uno studio comparativo di diverse soluzioni system-on-chip (SoC), risultato nell'adozione del sistema Zynq e l'uso della tecnologia FPGA (Field-Programmable Gate Array). La tesi affronta anche l'applicazione delle metodologie di Continuous Integration/Continuous Deployment (CI/CD) nello sviluppo FPGA, portando a un salto in termini di efficienza e affidabilità dello sviluppo.

La sezione dello sviluppo software descrive il sistema operativo Petalinux e lo sviluppo dell'applicazione Python e dei driver in user-space.

Infine, la tesi presenta metodologie di test di applicazione e di *performance*, inclusi strategie di integrazione e distribuzione continue, per garantire la robustezza e l'affidabilità del CDC. La sezione di valutazione delle performance fornisce una panoramica sulla sfida della validazione metrologica delle prestazioni dello strumento e sui risultati dei test effettuati.

Parole chiave: CERN, sorgente di corrente, calibrazione, Zynq, DCCT, FPGA

Contents

1 | Introduction

1.1. CERN

Conseil Européen pour la Recherche Nucléaire (CERN) is the world's biggest organization for research in high-energy physics. Most of the laboratory research activity is carried out with the use of particle accelerators. Particle accelerators are organized in the so-called "accelerator complex", where they are connected for operation.

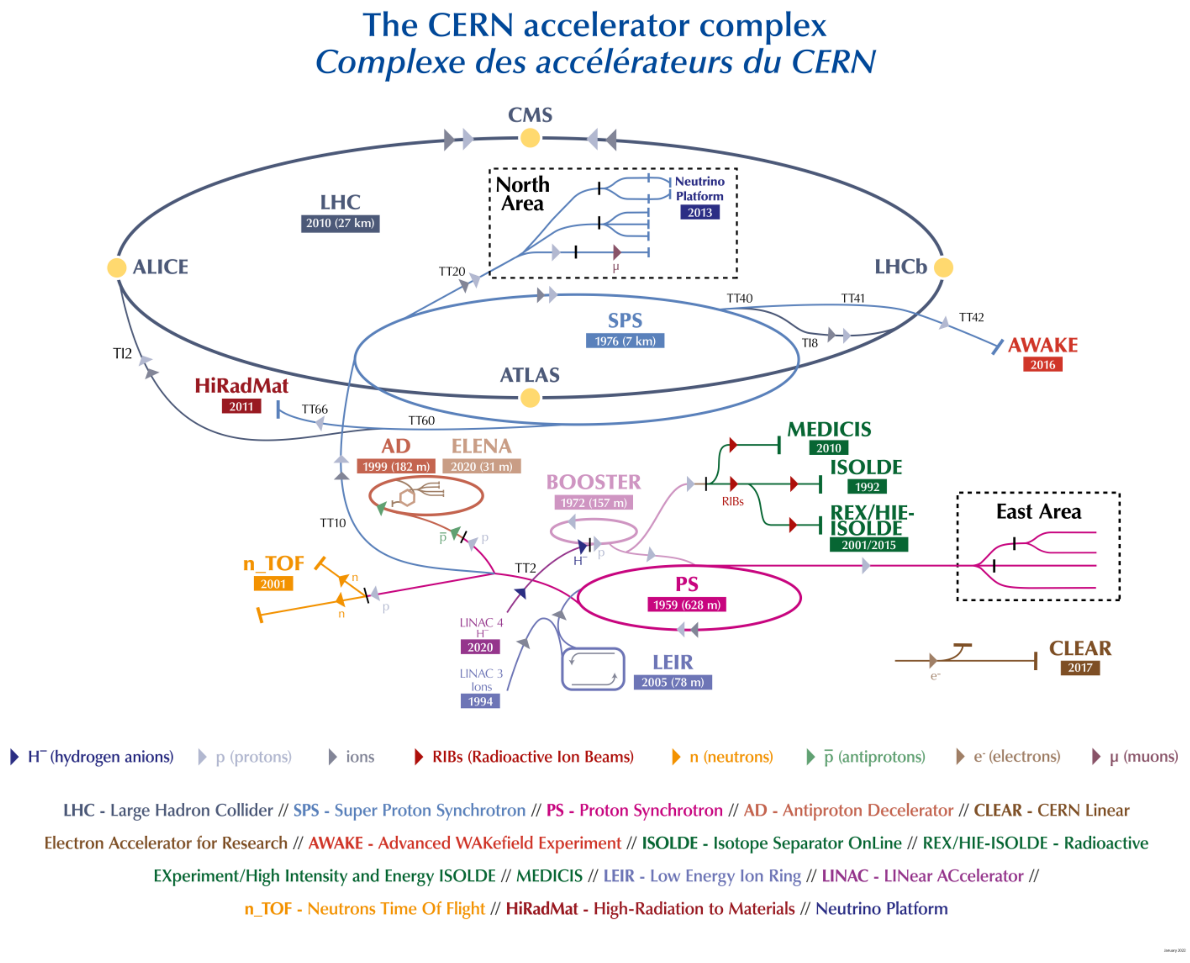

The CERN accelerator complex
Complexe des accélérateurs du CERN
CMS
LHC
2010 (27 km)
North Area
Neutrino Platform
2013
ALICE
LHCb
TT20
TT40
TT41
TT42
AWAKE
2016
SPS
1976 (7 km)
TI8
TI2
ATLAS
HiRadMat
2011
TT66
TT60
AD
1999 (182 m)
ELENA
2020 (31 m)
BOOSTER
1972 (157 m)
MEDICIS
2010
ISOLDE
1992
REX/HIE-ISOLDE
2001/2015
RIBs
East Area
TT10
n_TOF
2001
TT2
PS
1959 (628 m)
LINAC 4
2020
LINAC 3
Ions
1994
LEIR
2005 (78 m)
CLEAR
2017
H⁻ (hydrogen anions)
p (protons)
ions
RIBs (Radioactive Ion Beams)
n (neutrons)
p̄ (antiprotons)
e⁻ (electrons)
μ (muons)
LHC - Large Hadron Collider // SPS - Super Proton Synchrotron // PS - Proton Synchrotron // AD - Antiproton Decelerator // CLEAR - CERN Linear Electron Accelerator for Research // AWAKE - Advanced WAKefield Experiment // ISOLDE - Isotope Separator OnLine // REX/HIE-ISOLDE - Radioactive EXperiment/High Intensity and Energy ISOLDE // MEDICIS // LEIR - Low Energy Ion Ring // LINAC - LINear ACcelerator // n_TOF - Neutrons Time Of Flight // HiRadMat - High-Radiation to Materials // Neutrino Platform

Figure 1.1: CERN accelerator complex diagram, Source: [32]

The CERN accelerator complex is composed of several accelerators, that most of the time run with protons. The starting point of the protons is the Linear Accelerator 4 (LINAC4), a recent upgrade from the Linear Accelerator 2 (LINAC2), that injects protons at the energy of 160 MeV. The LINAC4 injects into the Proton Synchrotron Booster (PSB), followed by the Proton Synchrotron (PS), the Super Proton Synchrotron (SPS) and finally the Large Hadron Collider (LHC).

The LHC experiments are not the only scientific users of the complex, in fact, their usage accounts for only a minor percentage of the total proton production, many other experiments exist and receive particle beams from the various accelerators in the complex.

The complex is run 24 hours a day, 7 days a week, for most of the year. This implies that all systems in the accelerator are requested to be highly reliable and available.

Accelerators, in general, are constituted by magnets, used to bend and shape the beam and by RF cavities, used to accelerate the beam. The magnets are powered by power converters, that get AC power from the power grid and convert it to DC power, controlled with considerable precision.

1.2. The LHC

The LHC is a superconducting ring-shaped particle accelerator. It is the largest in CERN's complex and the largest in the world, with a circumference of 27 km. It is located underground, across the border of Switzerland and France, at about 100m depth. Two particle beams circulate in two beam pipes, in opposite ways, to meet in a head-to-head collision at four points along the accelerator, where the four main experiments are located: A Toroidal LHC Apparatus (ATLAS), Compact Muon Solenoid (CMS), A Large Ion Collider Experiment (ALICE) and Large Hadron Collider beauty (LHCb). The ATLAS and CMS experiments are the ones that confirmed experimentally the existence of the Higgs boson in 2012.

1.3. Powering for the accelerator complex

The LHC is largely constituted by dipole magnets. The dipole magnets are used to curve the beam and keep it orbiting in the accelerator. They are made of two opposing superconducting coils, that are cooled down to 1.9 K, to reach the superconducting state. A large current, in the order of 10 kA, is passed in the coil to produce a magnetic field in the order of 8 T, that is used to curve the beam. Since the turn radius is a function of the current, it is critical to control the current with high precision. Moreover, since the LHC dipoles are powered in eight separate sectors, it is necessary that the sector currents are equal to each other, to avoid a step in the magnetic field at the sector boundaries. This is achieved by using a Direct-Current Current Transformer (DCCT), which is a current sensor that measures the current in the coil, and a control loop implemented digitally in the power converter.

The CERN accelerator complex is powered by a network of power converters, that are located in the various underground and above-ground facilities. The power converters are connected to the power grid, and convert the AC power into DC power. The DC power is then used to power the magnets in the accelerator.

1.3.1. The Hi-Lumi upgrade

An upgrade for the LHC is currently under development. The upgrade is called High Luminosity LHC (Hi-Lumi) and will increase the luminosity of the accelerator by a factor of 10. By luminosity we mean the number of collisions per second per unit of area that an accelerator can produce. This will allow to increase the number of collisions in the experiments, and therefore increase the probability of observing rare events.

The new inner triplet magnets

The upgrade will include, among other upgrades, the replacement of the inner triplet magnets in the LHC for the ATLAS and CMS experiments. The inner triplet magnets are the magnets that are located at both ends of the interaction points of the accelerator, where the experiments are located. They take their name from the fact that they are functionally made of three quadrupole magnets. In practice, one of the quadrupoles is split in two. Two configurations of quadrupole magnets exist: focusing and defocusing. One is focusing the beam on the horizontal plane, but defocusing it in the vertical plane, and they are called focusing quadrupoles. The defocusing quadrupoles do the opposite, defocusing the beam on the horizontal plane and focusing it in the vertical plane. Other than the quadrupole magnets, the inner triplet also contains a dipole magnet.

The inner triplet upgrade will be key to increasing the luminosity of the accelerator. The new inner triplet's magnets will require increased precision in the current control, for which new upgraded DCCTs are required.

1.3.2. Current control

Since the magnetic field in an electromagnet depends on the current in the coil, the current is controlled to adjust the magnetic field. To meet the strict specification for current control on the beam magnets, a closed loop system is in use. In general, a power converter control system will be constituted of a voltage source (the power converter itself), a current measurement system, and a controller. While the power converter topologies can greatly vary between converters, in CERN's installations the controller is implemented in a modular platform called Regulating FGC (RegFGC) [3] [15]. The RegFGC platform supports the Function Generator Controller (FGC), on which the control is implemented. Various modules are then used to interface the FGC to the power converter. The current control loop needs an accurate measurement of the output current to control it effectively. Many solutions exist to measure current in this kind of application, including shunt resistors, Hall effect sensors and current transformers. The described systems combine the requirement for high precision and the need to measure high currents, leaving DCCTs as the only option [6].

1.3.3. DCCTs

The DCCTs are current sensors that exploit the core saturation of a transformer to measure DC currents. DCCTs were initially developed at CERN for beam current monitoring in particle accelerators, for which they are still used to this day, including in the LHC [36]. At CERN, they have been in use to measure the current output of the power converters for control and regulation, from several generations of accelerators [1].

A current transformer, in general, is not able to operate in the DC regime, since it is based on the principle of induction. The DCCTs can operate in the DC regime, by exploiting a modulation and demodulation signal in the magnetic core.

Principle of operation

The DCCTs are constituted by a toroidal transformer core, on which is wound a conductor, called primary, that carries the DC current to be measured, I. Since I is DC, it cannot induce an AC magnetic flux in the core that can be picked up by another coil. The core is therefore modulated using a winding, that is excited by an AC voltage. The AC voltage is generated by an oscillator. The current in the excitation winding is then measured. Since the current measured by the primary can often be very large, the primary is made of a single bus bar going through the center of the transformer's core, which makes it a one turn winding.

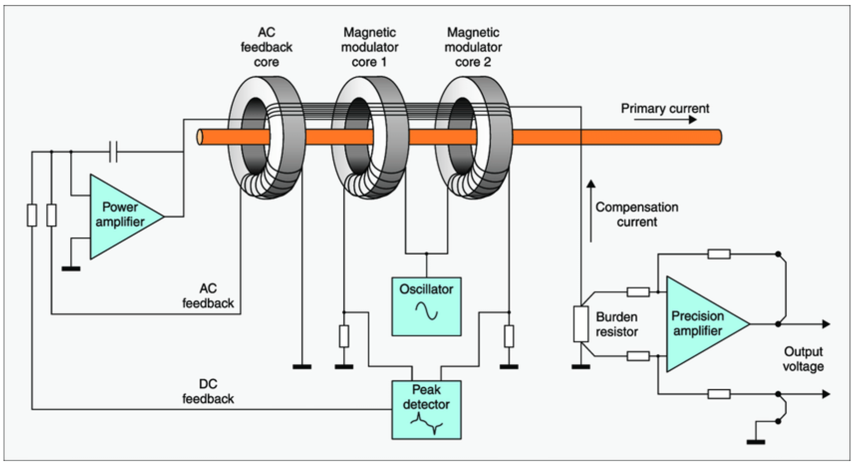


Figure 1.2: DCCT block diagram, Source: [9]

Zero flux detector Magnetic saturation is a phenomenon that occurs in magnetic materials, where the magnetic flux density B stops increasing with the magnetic field H. At that point, the ratio between voltage and current in the winding stops being linear, and exhibits peaks in the current when the core reaches saturation. When there is even a minor DC offset in the core, the peaks in the current are asymmetric, as shown in Figure 1.3.

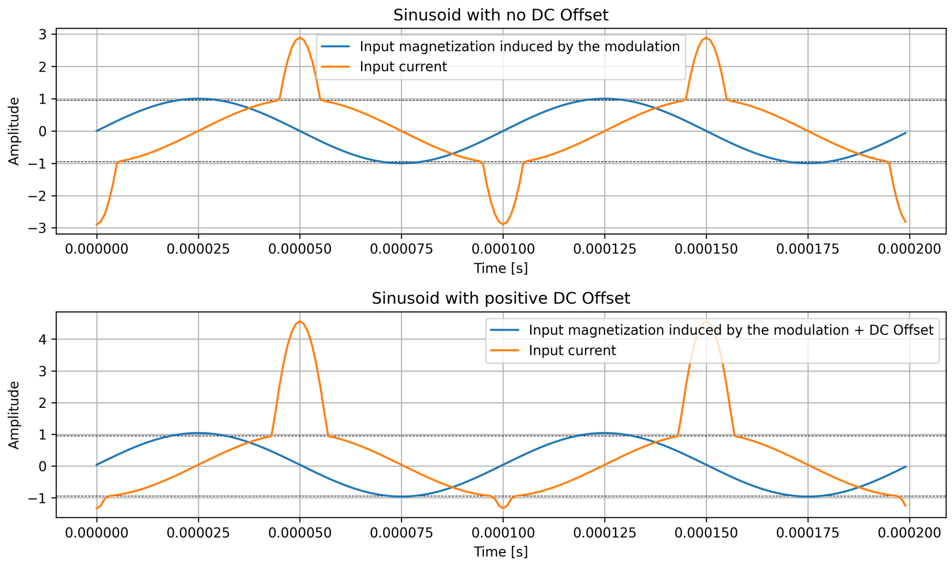


Figure 1.3: Core saturation in time

This allows to easily measure any DC offset in the core with great sensitivity. The DC offset is then used to null the flux in the core, by injecting a secondary current in a secondary winding, that is wound in the opposite direction with respect to the primary winding.

Harmonics analysis Studying the distorted waveforms of the currents in Figure 1.3 in the frequency domain, it is possible to see that the current is composed of a fundamental component and odd harmonics when the DC offset is zero, while it is composed of a fundamental component and all harmonics when the DC offset is not zero. This is shown in Figure 1.4.

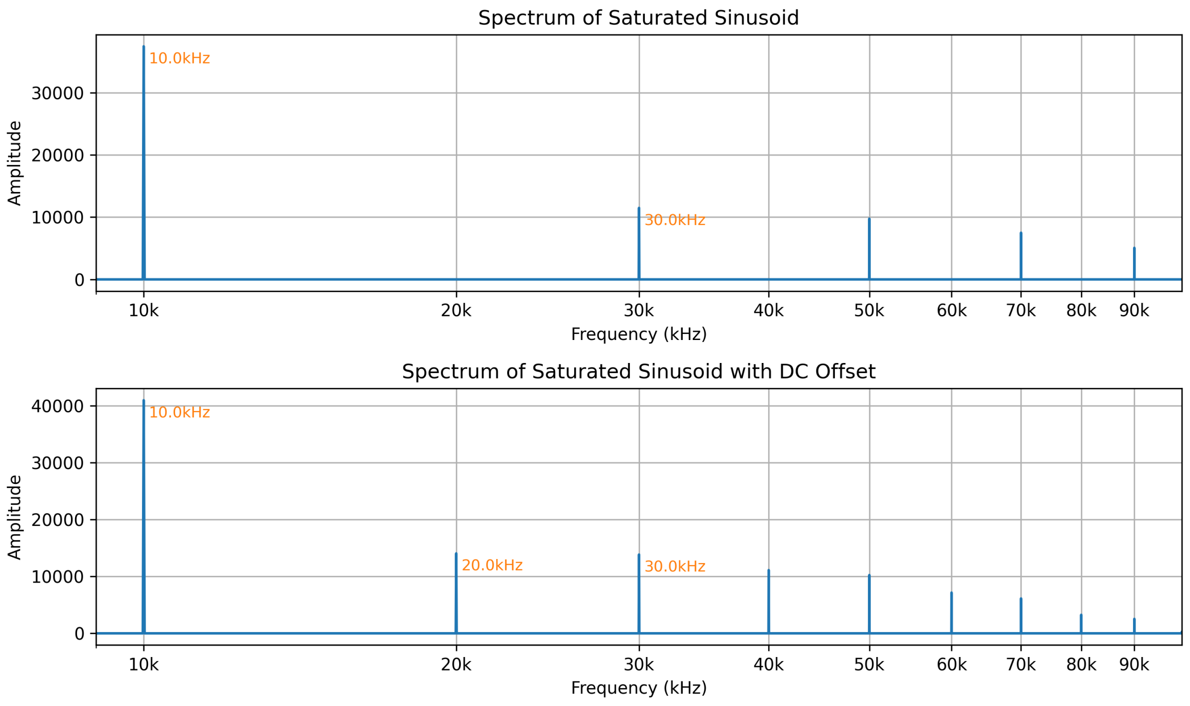


Figure 1.4: Core saturation in frequency

The second harmonic is then used as an error signal so that an amplifier injects a secondary current such that the error is nulled. This means that the DC offset has been removed from the core, so the secondary current matches with the primary to a high degree of precision. To extract the second harmonic for the signal, a synchronous demodulation is used.

Secondary current measurement The secondary current can then be measured with a current sensor, without perturbing the original current. Since the turn ratio between the secondary and the primary determines the ratio between the secondary and the primary current, the secondary current can be much smaller, thus more easily measurable with high accuracy, for example with a shunt resistor, often referred to as a burden resistor.

Since transformers are reciprocal electric machines, in a DCCT constructed as described, the AC modulation would induce an AC component both in the primary and in the secondary, disturbing the measurement. Instead, the core is effectively divided into two symmetrical cores and the AC modulation is injected in two identical windings, one on each core, that are wound in opposite directions. The other windings are then wound on both cores. This way, the AC modulation is greatly reduced as long as the two cores match, and the DCCT is able to measure DC currents without affecting the primary or secondary currents. This is shown in Figure 1.2.

1.3.4. Calibration requirements

To ensure the precision of the current control, the DCCTs need to be calibrated. The calibration is performed by injecting a known current in the coil and measuring the output of the DCCT. The calibration is performed periodically, to ensure that the DCCTs are still within the required precision.

Since producing a very large and very precise current is not trivial, a calibration winding is sometimes used. The calibration winding is a winding that is placed on the DCCT core and has a certain turn ratio with respect to the primary winding. This turn gain is used to scale the injected current, to emulate the flux produced by the current in the primary winding. The calibration current is provided, in CERN's installations, by the CERN DCCT Calibrator (CDC).

1.4. The CERN DCCT Calibrator (CDC)

The CDC is a device that is used to calibrate the DCCTs. It is a programmable $\pm 10\ A$ current source. It can be used in various configurations, both to check and calibrate the DCCTs, as well as to test in-situ current references, as Figure 1.5 shows.

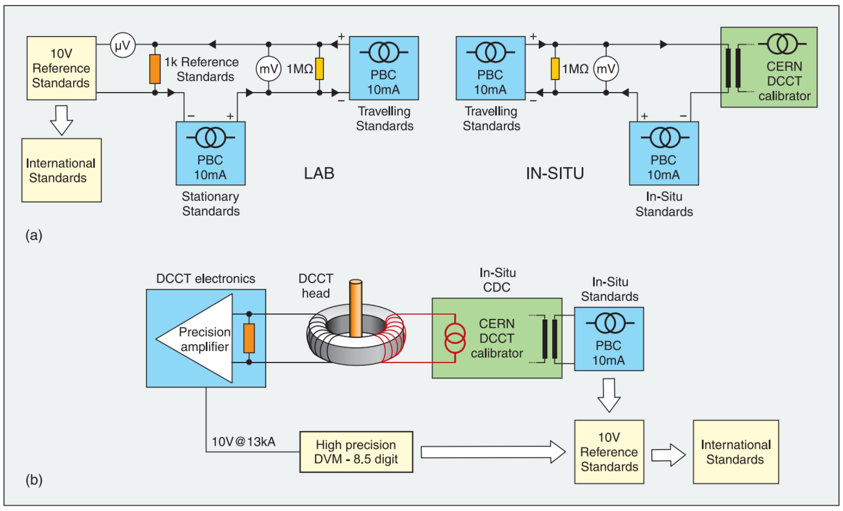


Figure 1.5: CDC configurations, Source: Fig. 4 in [9]

The original CDC was designed in the early 2000s [18] and is now hard to manufacture as it contains many obsolete components. It is based on a modular architecture, where the modules are connected to a backplane. It is in use in the LHC and in some test facilities at CERN, as well as in other laboratories around the world.

1.4.1. CDC working principle

The CDC system itself is based on the DCCT principle. Reference Figure 1.6 for a basic functional block diagram of the CDC.

The system works by injecting a very stable reference current - called the primary current - in a winding, which creates a non-null magnetization in the core. The magnetization is picked up by the zero-flux detector module, which creates an error signal, that is then fed in a Proportional Integral (PI) loop which generates a secondary current, in the opposite direction, such that the error (and so the flux in the core) is nulled. The secondary current is then output from the instrument. This way, the CDC is able to output a current that mirrors the current reference.

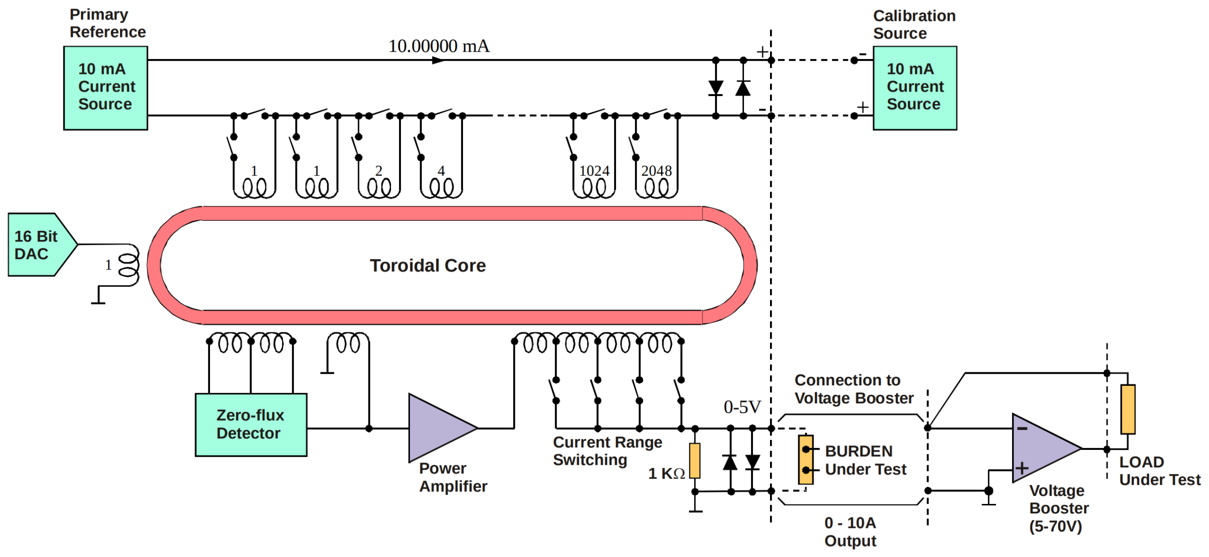


Figure 1.6: CDC basic structure, Fig. 6 in [9]

By varying the number of turns of the primary winding it is possible to vary the output current, in steps multiple of the primary current. The winding modification is achieved through a binary encoding system, where a set of discrete windings with turns ratios of 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, and 1 are selectively inserted or removed from the current path. This selective inclusion and exclusion mechanism allows for a highly flexible and precise adjustment of the total number of turns, enabling the customization of the output current in incremental steps that are multiples of the primary current.

Each specific combination of included windings corresponds to a unique total number of turns, providing a wide range of output current options to precisely mirror the reference current. Finer tuning is allowed by a second single-turn winding that is connected to a Digital to Analog Converter (DAC) and is used to fine-tune the output current. The DAC is controlled by the CDC's control system.

Zero-flux detector

The zero-flux detector is at the core of the instrument, and it is very similar to the one used in the DCCTs. It is constituted by twin inner cores that are modulated in opposite directions, and the result is then picked up by two windings that are wound on both cores. The twin cores are then used to cancel out the AC modulation so that the secondary current is not affected. The pickup coils are wired in anti-series to not cancel out the two modulations but rather to add them up. This is shown in Figure 1.7.

Current reference

The CDC needs a very stable current reference to work. The current reference is provided by a commercial module called PBC. It is based on an ovenized zener diode, the LTZ1000, that is used to generate a very stable voltage reference. Upgraded units, called V3, exploit the even better-performing ADR1000. The voltage reference is then used to generate a very stable current reference, that is used as the primary current in the CDC. The device has been described in [19]. The PBC is able to generate a current reference of $10mA$ with stability in the order of a few parts per million per year, detailed in Table 1.1

Table 1.1: Detailed comparison between PBC Current Reference V3 and V2, from [33] [34]

Specification	PBC V3	PBC V2
Temperature Stability	±0.1ppm/°C	±0.2ppm/°C
Time Stability (first year typical)	±3ppm/year	±3ppm/year
Time stability - ongoing (typical)	±1.5ppm/year	±2ppm/year
Noise (typical, RMS)	150pA	200pA

PBC x5 Chassis

As part of the upgrade, the possibility of using five 10 mA current references in parallel instead of one was introduced. This is achieved by using a PBC x5 chassis, that is able

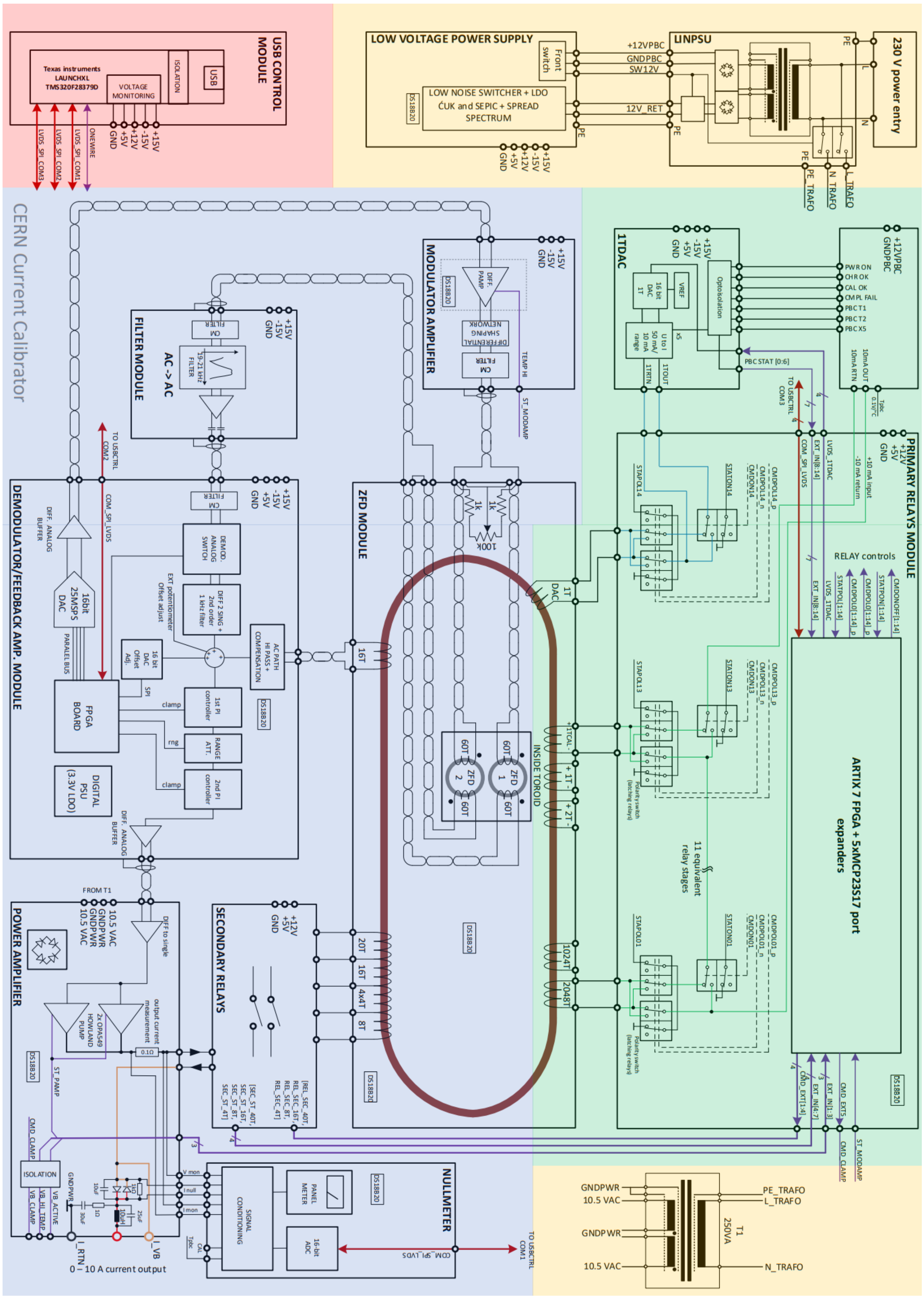


Figure 1.7: CDC detailed schematic, slide 18 in [30]

to host five PBCs. The PBC x5 chassis is able to generate a current reference of $50mA$. This introduces two main advantages: the first is that the current reference is more stable, since the variance is reduced by $\sqrt{5}$, so the stability is improved by a factor of 2.2. The second advantage is that the ampere-turns in the primary winding are increased by a factor of 5 in the 1 A range while being able to operate effectively as in the 5 A range. This way, the contribution of the magnetic noise in the core is reduced.

A third advantage is that being the PBC x5 chassis a separate assembly, it can be placed further away from the CDC itself, reducing the thermal influence of the CDC on the reference current.

Voltage booster

Since the compliance voltage of the CDC is limited to $5V$, a voltage booster is used to increase the compliance voltage to $70V$. This allows to use the CDC with DCCTs that have a higher resistance in the calibration winding.

The Voltage Booster (VB) is an amplifier that is connected between the output terminals of the CDC, as shown by Figure 1.6. The VB acts so that the current being output by the output terminals enters the load and is then sunk by the VB. The VB is able to sink a current of up to $5A$, so the range is limited to $5A$ when the VB is used. TheVB is controlled by the CDC's control system, which is able to set the output current and the compliance voltage.

1.4.2. Upgrade overview

The CDC is currently being upgraded. The upgrade is being carried out by the SY-EPC High Precision Measurements section (SY-EPC-HPM) section, which is in charge of the high-precision measurements for power converters. The purpose of the upgrade is to improve the performance of the CDC, to meet the new requirements for the Hi-Lumi upgrade. The requirements for the upgrade were specified in [7] and are summarized in Table 1.2.

Table 1.2: Comparison of Specifications between the Target and Existing Current Sources.

Parameter	**Reference Current Source**	**Existing Current Calibrator**
Initial uncertainty after calibration (2xrms ppm)	0.5	1
Linearity (max ppm)	0.3	0.5
Stability during a 12h fill (max ppm)	0.2	1.2
Short term stability during 20min (2xrms ppm)	0.1	0.1
Noise at <500Hz (2xrms ppm)	1.5	1.0
Fill to fill repeatability (2xrms ppm)	0.1	0.5
Long term fill to fill stability (max ppm)	1.5	2.0
Temperature coefficient (2xrms ppm/C)	0.2	approx. 0.2

The upgrade is being carried out in two phases. The first phase is the upgrade of the CDC's hardware, while the second phase is the upgrade of the CDC's controls. The present work is about the second phase of the upgrade. The hardware upgrade consists of the redesign of the CDC's modules with new, better-performing, ones.

First phase: hardware upgrade

The hardware upgrade was carried out between 2021 and 2022. The main focus points were:

- **Improve on linearity**;
- **Improve on 12h stability**;
- Identification of 1/f noise source;
- Better thermal management;
- Reduction of modulation noise;
- Improved Electromagnetic Interference (EMI) performance;
- Replacement of the obsolete components;
- Research on magnetic materials.

The upgrade was completed in June 2022, with two development units manufactured and tested in all the parts. The upgraded CDC featured a USB controller based on a Digital Signal Processor (DSP) from Texas Instrument. This was a temporary solution, to allow for testing of the new hardware. The final control system will be based on a Zynq System-on-Chip (SoC). The new CDC was tested in the laboratory with good results. The results of the upgrade are shown in [30].

Second phase: control upgrade

The control upgrade consists of the replacement of the upgraded CDC's temporary controls with a new control system. The control of the old units was based on a Renesas M32C/87 microcontroller. During the first phase of the upgrade, the control was substituted with a DSP-based temporary solution, for testing purposes. The new control system will be based on a Zynq DSP, which will be used to implement the control.

The new control system will adopt the Network Command-Response Protocol (NCRP), to standardize the interface and uniform it to the one already in place for power converters. The old CDC control was based on the serial version of the Command-Response Protocol (CRP), while the new one will be based on the network version of the protocol.

1.4.3. CDC control system

The duties of the control system are:

- Provide a user interface to the user, to allow for the control of the CDC. The user will control the CDC through the network, by sending commands via Transmission Control Protocol (TCP) to the control system;
- Control the various modules of the CDC, following the commands received from the user;
- Monitor the status of the CDC, in particular, the status of the modules and the power supply rails, and raise alarms if necessary;

1.4.4. Modules

The CDC is constituted by various modules, that are connected to a backplane. The modules are:

- Power Supply Unit (PSU) module;
- Modulation Amplifier (MODAMP) module;

- Secondary Relays (SECREL) module;
- Primary Relays (PRIMREL) module;
- One-Turn DAC Driver (1TDAC) module;
- Active Filter (ACTFILT) module;
- Feedback and Demodulation Amplifier (FADEM) module;
- 10mA current reference module (PBC);
- Network Controller (NETCTRL) module;
- Zero-Flux Detector (ZFD) module;
- Nullmeter (NULLMETER) module.

1.4.5. Control specificaiton

The focus for the development is ease of development, flexibility and maintainability. Since the CDC is not manufactured in large quantities, cost is not a primary concern. The lifespan of the instrument design is expected to be in the order of decades, so the control system is required to be easily maintainable and extensible, to allow for future upgrades and modifications. Testability and ease of deployment are also design goals.

The control system is implemented in the NETCTRL module, and it is required to control the following:

- The FADEM module via Serial Peripheral Interface (SPI);
- The PRIMREL module via SPI;
- The NULLMETER module via SPI;
- The network interface, where it receives commands from the user and sends responses;

The NETCTRL module is also required to monitor the status of the CDC, in particular the status of the modules and the power supply rails, and raise alarms if necessary.

1.4.6. NCRP protocol

The FGC NCRP is a protocol commonly used at CERN for the control of power converters. It is specified by [4] [29].

FGC device classes

Every device controlled by the NCRP protocol must adhere to a device class. Device classes are defined by the SY-EPC Converter Control Software section (SY-EPC-CCS) group at CERN. Each device class specifies which commands are available for the device, and how the device must respond to the commands. The device classes are defined in [4].

The CDC is implemented in *Class 150: CALSYS_CDC*, which is the class that is used for all the CDC devices in the accelerator complex.

NCRP properties

Each device class is characterized by a series of properties that are common to all devices of that class. The properties are defined by a name, a type, and a range. Typically, FGC properties are structured in a hierarchy, where each level may contain other levels or properties.

Properties for the CDC controller are defined in [8]. In the CDC, 33 properties are implemented in total.

In general, a property name is a string or a series of strings separated by dots, where the dots define the hierarchy. For example, CALSYS.CDC.ADC.GAIN_1A is a property that is defined in the CALSYS.CDC.ADC hierarchy. Inside CALSYS.CDC.ADC many other properties are found, all related to the ADC, such as I_MEAS, I_NULL, PBC_TEMP, RAW_ZERO, V_MEAS. In CALSYS.CDC other properties and groups of properties are found, such as DAC1TW_CAL, DAC1TW, NULL and so on.

Each property is, in principle, an array. The array can be of any dimension, scalar properties are arrays of dimension 1. The array can be of any type, including integer, floating point, string, boolean, and so on.

The properties are detailed in Appendix 8.

NCRP commands

The NCRP command is the means of interaction with properties. It is based on a set-get paradigm, where the user can set a property to a certain value, or get the value of a property. The commands are defined in [4] and [29].

Each command should start with an exclamation mark and should end with a new line. It should contain a verb and a property name, where the verb can either be set (`s`) or get (`g`). A minimum command is, therefore:

```
! g CALSYS.CDC.ADC.GAIN_1A
```

The command can also contain a value, in the case of a set command. The value is a string that is converted to the property type. For example, the command:

```
! s CALSYS.CDC.ADC.GAIN_1A 1.0
```

The commands are case-insensitive.

To get and set values of properties that are arrays, the command can contain an index or a range. Supported ranges are in the form of `[index]`, `[from, to]`, `[from, ]` (with the meaning of "from index to the end of the array") and `[from, to, step]`.

The values to set can either be targetted individually by index or they can be set in a range by specifying values separated by commas. For example, to set the first 2 values of the CALSYS.CDC.DAC.RAW property to 100 and 200, the command would be:

```
! s CALSYS.CDC.DAC.RAW[0,1] 100,200
```

To get the first 2 values of the CALSYS.CDC.DAC.RAW property, the command would be:

```
! g CALSYS.CDC.DAC.RAW[0,1]
```

NCRP responses

The FGC devices respond to commands with a response. The response is a string that contains the result of the command. The response is defined in [4] and [29].

A successful response to a set would be:

```
$ .


;
```

A successful response to a get would be:

```
$ .
value
;
```

If the command was unsuccessful, the response would be an error code and a short string, such as:

```
$ !
15 no delimiter
```

```
;
```

The error codes are defined in the FGC protocol specification.

FGC Gateways

In the NCRP protocol, no device is standalone. Each device is connected to the network through a gateway, that groups together a number of devices. The connection between the gateway and the devices is arbitrary and can be implemented in any way. Multiple types of gateways, implemented as FGC devices with their own set of properties, are available. The gateways are defined in [5], along with the rest of the protocol and the device classes with their properties.

To be able to interact with the standard tools that are available for the NCRP protocol, the CDC will need to be connected to the network through a gateway. Instead of implementing a real gateway, the CDC will feature a gateway emulator, where all the relative properties will exist formally in the CDC controller, but they will not be implemented. The CDC logic will then route commands either to the CDC controller or to the gateway emulator, depending on the specified device. This way, the CDC will be able to interact with the standard tools that are available for the NCRP protocol.

The NCRP protocol is defined in such a way that each command addresses a specific device in the gateway. The device is specified by a device number, that is a number that uniquely identifies the device in the gateway. The device number is specified in the command, after the gateway name, separated by a colon. For example, a valid command looks like:

```
! \verb> <device-number>:<property-name>\n
```

If the device number is not specified, the gateway is assumed to be the target of the command.

1.5. Programmable electronics for the controls

1.5.1. Network

The control system of the Current Calibrator benefits greatly from an embedded processor, in particular regarding the networking capabilities. One of the standout features of utilizing embedded processors is their support for Linux. With the Linux operating system, network support and utilities are available out of the box on an embedded processor.

The embedded processor simplifies the integration of the control system into the network. Linux provides a mature and robust network stack ensuring reliability and efficiency in communication, facilitating seamless and complete remote control capabilities. Moreover, because Linux is a well-known, widely-used, and open-source operating system, it provides a wealth of resources, including a global community of developers and extensive documentation, aiding in rapid development and troubleshooting.

1.5.2. Hardware interfaces

The control system of the Current Calibrator requires a number of hardware interfaces to control the various modules of the CDC. These interfaces can be implemented in an FPGA, in particular considering that the required IP blocks are readily available.

The FPGA solution to implement interfaces allows for a high degree of flexibility and customization. The FPGA can be programmed to implement any interface, and it can be reprogrammed to implement a different interface. This speeds up the development process.

1.5.3. The Zynq system

The Zynq system is a SoC that combines an ARM Cortex-A9 processor with a Xilinx 7-series Field Programmable Gate Array (FPGA). The Zynq system is a very popular choice for embedded systems, due to its flexibility and performance. The Zynq system is also used in many CERN projects, both for the experiments and for the machine.

A natural choice for the control system of the CDC is the Zynq system. The Zynq system allows for the implementation of the control system in a single chip, with the ARM processor running Linux and the FPGA implementing the hardware interfaces.

1.5.4. The CI/CD methodology for FPGA development

The development of the controller exploited the Continuous Integration and Continuous Deployment (CI/CD) methodology. It is a methodology that is used in software development to ensure that the software is always in a working state and that it can be deployed at any time. The CI/CD methodology is based on the use of a version control system, such as Git, and on the use of automated testing and deployment.

The CI/CD methodology is not commonly used in FPGA development, but it is a very powerful tool that can greatly improve the development process. The CI/CD methodology is based on the use of a version control system, such as Git, and on the use of automated testing and deployment. CI/CD for FPGAs is a relatively new concept, but it is gaining traction in the industry.

This paradigm allows for testing the code at every commit, and to prepare and package the code for deployment at every commit. This allows for a very fast development cycle, where the developer can test the code at every commit, and the code is always ready for deployment. This greatly speeds up the development process.

2 | Hardware

2.1. Introduction

This chapter describes the hardware design of the system. The system is designed to be modular so that parts can be replaced or upgraded without the need to redesign the whole system. The control hardware will constitute a module in the system.

2.2. Requirements

The main points of the requirements for the control hardware are:

- The controller should accept an Ethernet connection for receiving commands;
- The controller should monitor the voltages of the power supply;
- The controller should be able to control the three SPI devices over Low Voltage Differential Signaling (LVDS);
- The controller should be able to connect to the 1-Wire (1-Wire) bus and power the devices on the bus using the parasitic power mode;
- The controller should have a 1-Wire device of its own to identify itself on the bus.

Moreover, the primary targets are:

- Optimizing development time;
- Optimizing maintainability: the system should be designed using technologies that are already known by the team, and that can be easily understood by other developers;
- Optimizing the lifetime of the solution: the system should be designed using technologies that are not going to be obsolete shortly.
- Design for testability: the system should be designed in a way that allows for easy testing of the hardware and software.

The cost of the controller was deemed secondary, as the system will be manufactured in small quantities and the cost of the controller is expected to be negligible compared to the cost of the whole system.

2.3. SoC vs SoC with FPGA

Two main paths were studied for the development of the controller: one solution makes use of a SoC and its hardware peripherals, while the other solution makes use of a SoC and an FPGA.

Both solutions have advantages and disadvantages. FPGAs are a very flexible solution, that offers complete customization of the interfaces but requires the developer to spend more effort in the development. SoCs are a simpler solution, that offers less flexibility but requires less effort in the development. At CERN the FPGA technology is widely used, well known and well supported, while the standalone SoC solution is not widespread. Both solutions are analyzed in the following sections.

2.3.1. Operating system vs bare-metal programming

Embedded processors can be programmed using a bare-metal approach or using an operating system. The bare-metal approach is more suitable for real-time applications, as it allows for more control over the execution of the code. The operating system approach is more suitable for multi-tasking applications and for applications that require a high level of abstraction. The operating system already integrates the full network stack, so it is easier to implement the Ethernet interface using an operating system. The bare-metal approach requires the implementation of the network stack, which is a complex task. The bare-metal approach is more suitable for the control logic, as it allows for more control over the execution of the code. The operating system approach is more suitable for the Ethernet interface, as it allows for a higher level of abstraction. As the control logic is not a real-time application, the operating system approach is more suitable for the controller.

2.3.2. Operating systems

A variety of operating systems are available for the embedded processors. The most popular operating systems are Linux and FreeRTOS. Linux is a full-featured operating system, while FreeRTOS is a real-time operating system. The control logic is not a real-time application, and it can benefit greatly from the high level of abstraction and the large number of drivers available in Linux, so Linux is more suitable for the controller.

The Ethernet interface is not a real-time application either, and it can benefit from the fully implemented network stack in Linux, so Linux was the chosen operating system for the controller.

2.4. The Zynq-7000 SoC

The original proposal for the development of the controller was to use a Zynq-7000 SoC. It offers a dual-core ARM Cortex-A9 processor and a FPGA in a single chip. The FPGA can be used to implement the SPI interface, the 1-Wire bus driver and the interface to control an Analog to Digital Converter (ADC) for voltage monitoring. The ARM processor can be used to implement the Ethernet interface and the control logic. The FPGA could also be used to implement the Ethernet interface, but the ARM processor is more suitable for this task.

The SoC is available in a few sizes and configurations, with the cost greatly varying between solutions. The cheapest bare SoCs are available for around 50 CHF, while System on Modules (SoMs) and module boards are available from 100 CHF to 500 CHF.

2.4.1. Lifetime of the solution

The Zynq-7000 SoC was released in 2011, and it is still in production. The Zynq-7000 SoC at launch was expected to be in production until 2027. In 2023, AMD (formerly Xilinx) announced that the lifetime of the 7-Series FPGA family was extended until 2035 [42], due to the high demand for the 7-Series FPGA family. The Zynq-7000 SoC is part of the 7-Series FPGA family, so it is expected to be in production until 2035.

2.4.2. Available boards

Different boards exist that feature a Zynq-7000 SoC. The most suitable boards are the ZedBoard, the Zybo and the Cora Z7 boards, which are manufactured by Digilent.

The Zybo board

The Zybo board uses a Zynq-7000 XC7Z010-1CLG400C SoC, which features a dual-core ARM Cortex-A9 processor and a FPGA with 28k logic cells. The board features 512MB of DDR3 RAM, 16MB of Quad-SPI Flash memory, a microSD card slot, a USB-JTAG port, a USB-UART port, an HDMI port, an audio jack, a VGA port, an Ethernet port, a Pmod connector, and a multitude of General Purpose Input Output (GPIO) pins. The board is powered by a 5V power supply.

The form factor of the board is not ideal for the final system, as it cannot be easily plugged-in onto a carrier board.

The ZedBoard

The ZedBoard uses a Zynq-7000 XC7Z020-CLG484 SoC, which features a dual-core ARM Cortex-A9 processor and a FPGA with 85k logic cells. The board features 1GB of DDR3 RAM, 128MB of Quad-SPI Flash memory, a microSD card slot, a USB-JTAG port, a USB-UART port, an HDMI port, an audio jack, a VGA port, an Ethernet port, a Pmod connector, and a multitude of GPIO pins. The board is powered by a 5V power supply.

Again, the form factor of the board is not ideal for the final system, as it cannot be easily plugged-in onto a carrier board.

The Cora Z7 board

The Cora Z7 uses a Zynq-7000 XC7Z007S-1CLG400 SoC, which features a single-core ARM Cortex-A9 processor and a FPGA with 23k logic cells. The board features 512MB of DDR3 RAM, a microSD card slot, a USB port for UART and JTAG, an Ethernet port, two Pmod connectors, and a multitude of GPIO pins. The board is powered by a 5V power supply.

The form factor of the board is adequate for the final system, as it can be easily plugged-in onto a carrier board. The board is also cheaper than the other boards, as it uses a smaller SoC.

2.5. The Zynq UltraScale+ MPSoC

The Zynq UltraScale+ MPSoC is the successor of the Zynq-7000 SoC. It offers a quad-core ARM Cortex-A53 processor, a dual-core ARM Cortex-R5 processor, a GPU and a FPGA in a single chip. The FPGA can be used to implement the SPI interface, the 1-Wire bus driver and the interface to control an ADC for voltage monitoring. The ARM processors can be used to implement the Ethernet interface and the control logic.

This SoC is much more powerful than the Zynq-7000 SoC, but it is also much more expensive. Moreover, the controller does not need any high-performance features. Handling the complexity of the SoC would also require more effort in the development, although many projects are maintained at CERN that use the Zynq UltraScale+ MPSoC, including some in the Systems department - Electrical Power Converters group (SY-EPC) group.

The MPSoC is available in many sizes and configurations, with the cost greatly varying between solutions. The cheapest bare SoCs are available for around 250 CHF, while the SoMs are available from 350 CHF to 1,500 CHF. It should also be considered that the bare SoCs require many external components to be used and an expensive high-layer count PCB to be designed, while the SoMs are ready to be used and require only a simple carrier board to be designed. The bare SoC approach is usually more suitable for high-volume production, while the SoM approach is more suitable for low-volume production.

2.5.1. Lifetime of the solution

The Zynq UltraScale+ MPSoC was released in 2015, and it is still in production. The Zynq UltraScale+ MPSoC at launch was expected to be in production until 2034, at least. The life expectancy of the Zynq UltraScale+ MPSoC matches the requirements of the project.

2.5.2. Available boards

Different boards and SoMs are available that feature the Zynq UltraScale+ MPSoC. Some are manufactured by Xilinx such as the Kria SoM, while others are available from third-party manufacturers.

2.6. The DI/OT platform

The Distributed I/O Tier (DI/OT) platform is a common platform used at CERN for the development of control systems. It is developed by the Beams department - Controls Electronics and Mechatronics group (BE-CEM) group, which is in charge of establishing common platforms and technologies for the organization. The DI/OT is constituted by a carrier chassis with a backplane that fits different types of boards. The system board is the main board of the DI/OT platform, and it makes use of a Zynq UltraScale+ MPSoC.

This board could be used to develop the controller, it would plug in directly into the backplane of the system, not requiring any carrier board. The board is very expensive and very high-performance. The use of this board would require the redesign of the backplane of the existing CDC, which is a time-consuming task.

Due to step pricing available to the organization, the cost of the board is significantly less than the cost of the Zynq UltraScale+ MPSoC itself. The board can be procured internally for 1,500 CHF, while the Zynq UltraScale+ MPSoC alone costs 2,900 CHF on the regular market.

2.6.1. Lifetime of the solution

This board is part of the DI/OT platform, which is a common platform at CERN. The DI/OT platform is expected to be in production for many years, as it will be supported by CERN directly.

This solution would not require the development of a custom Linux distribution, as the DI/OT platform already provides a Linux distribution that can be used.

2.7. The Toradex SoM

Toradex is a Swiss company that manufactures SoMs for embedded systems. The SoMs are based on ARM processors, and they are designed to be used in embedded systems. The SoMs feature various processors, with different Input Output (IO) characteristics and available features.

A SoM based on the NXP i.MX 8M series would be a reasonable choice for this project, as it features a quad-core ARM Cortex-A53 processor with support for Linux. The SoC also features a SPI interface and an onboard ADC. The SoC does not include a feature to implement the physical LVDS interface, but it can be implemented in external circuitry. The SoC also features a Gigabit Ethernet interface. The SoC is not a high-performance SoC, but it is more than enough for the controller.

Notably, it does not feature any hardware support for the 1-Wire bus, so the 1-Wire bus driver would have to be implemented in software. This is possible, but no kernel driver implementation is readily available for use with the parasitic power mode of the 1-Wire bus. It would have to be implemented using the GPIO interface of the SoC, but this would either require a custom kernel driver or a custom userspace driver that would not be as efficient as a kernel driver and may risk not being able to meet the strict timing requirements of the 1-Wire bus.

The support for a Linux distribution is already available for the SoC from the manufacturer, so the development of a custom Linux distribution would not be required. The support for the SoC is expected to be maintained for many years, as the SoC is part of the NXP i.MX 8M series, which is a widely used series of SoCs. Customization of the Linux distribution would be required to implement the 1-Wire bus driver, and some configuration of the Yocto build system would be required to build the Linux distribution.

2.7.1. Lifetime of the solution

These modules are industrial computing modules, and as such enjoy a long life expectancy. The modules are expected to be in production for 10 years or more [37], and since they are widely used, they are expected to be supported by the manufacturer for many years.

Moreover, the manufacturer states that the modules are designed to be pin-compatible with future modules, so the modules can be replaced with newer modules without the need to redesign the carrier board. This would allow for the controller to be upgraded in the future without the need to redesign the carrier board and with just minor changes to the software.

2.7.2. Available boards

The modules are available in many sizes and configurations, with the cost greatly varying between solutions. The cheapest SoMs are available from 100 CHF to 300 CHF.

2.8. Comparison of the solutions

Table 2.1: Detailed comparison between the FPGA modules.

	Zynq-7000	**Zynq UltraScale+**	**DI/OT**	**Toradex SoM**
SPI	FPGA	FPGA	FPGA	x3
1-Wire	FPGA	FPGA	FPGA	GPIO
ADC	12 bit 17ch	10 bit 17ch	10 bit 17ch	12 bit 4ch
Ethernet	Gigabit	Gigabit	Gigabit	Gigabit
Complexity	Mid	High	High	Low
Form factor	SoM	SoM	Card	SoM
Price	Mid to Low	Mid to High	High	Mid to Low
Use in CERN	Common	Common	Standard	Uncommon
Long-Term	2035+	2030+[1]	CERN support	2034+

[1] The lifespan of Zynq UltraScale+ is expected to be extended, similar to the Zynq-7000 series, although there is no official announcement yet.

Both the bare SoC and the SoC with FPGA solutions are viable, and both present unique challenges. Moreover, there is a great variety of boards and SoMs available for both solutions. The SoC with FPGA solution is more flexible, but it requires working with a niche technology and brewing a custom Linux distribution. The bare SoC solution is less flexible, but it requires less effort in the development and it allows for the use of a standard Linux distribution. On the other hand, the expertise in the FPGA technology is widespread at CERN, while the bare SoC technology is not widespread.

Table 2.1 offers an overview of the different solutions. The Zynq-7000 SoC and the Toradex SoM are the cheapest and have a suitable life cycle ahead, but the Zynq-7000 SoC requires the development of a custom Linux distribution and the Toradex SoM requires the development of a custom 1-Wire bus driver for the kernel. The Zynq UltraScale+ SoC is the most expensive and has a long life cycle ahead, but it is overkill for the controller and it would require the redesign of the backplane of the existing CDC. The DI/OT platform is the most expensive and has a long life cycle ahead, but it would require the redesign of the backplane of the existing CDC with a backplane with inhomogeneous connectors.

2.8.1. Conclusions

The DI/OT solution was discarded both because of the complexity and because of the need to redesign a new *hybrid* backplane. The price of the solution was more than 15 times the price of other solutions, and the project wouldn't have benefited from the high-performance features of the DI/OT platform.

The Zynq UltraScale+ solution was similarly deemed too complex and costly for not making use of the enhanced performance of the SoC.

The Zynq-7000 solution was preferred in the end, both for the familiarity of the group with the tools and technologies and for flexibility. The Toradex SoM solution was also considered, but the lack of hardware support for the 1-Wire bus with parasitic power and the little diffusion of the technology at CERN made it less suitable for the project.

Table 2.2: Detailed comparison between the Zynq-7000 boards.

Features/Boards	**Zybo**	**ZedBoard**	**Cora Z7-07S**
SoC	XC7Z010-1CLG400C	XC7Z020-CLG484	XC7Z007S-1CLG400
Processor	ARM Cortex-A9	ARM Cortex-A9	ARM Cortex-A9
Processor Cores	2	2	1
FPGA Cells	28k	85k	23k
RAM	512MB DDR3	1GB DDR3	512MB DDR3
Flash Memory	16MB Quad-SPI	128MB Quad-SPI	-
MicroSD Slot	Yes	Yes	Yes
Ethernet	Yes	Yes	Yes
USB-UART	Yes	Yes	Yes
GPIO Pins	40 on Pmod	82 on FMC-LPC	49 on pin headers
Power Supply	5V	5V	5V
Form Factor	Not ideal	Not ideal	Adequate
Price	299 USD[1]	589 USD[1]	149 USD[1]

[1] Price from the manufacturer's website at the time of writing

The Cora Z7 board was selected as the preferred Zynq-7000, for the ability to be mounted directly and securely on a carrier board, and due to availability concerns for the other solutions.

2.9. LVDS drivers

To ensure reliable communication, the SPI control in this project adheres to the LVDS physical specification. The LVDS, a differential signaling standard, is particularly favorable for its application in ensuring reliability and performance in data transmissions [23]. However, the native configuration of the Zynq-7000 SoC in the selected module does not support LVDS, necessitating the integration of external LVDS drivers. The challenges and solutions associated with this integration are discussed further in the subsection 3.4.2.

In our design, we face a challenge with the physical layer requirements of the SPI interfaces, which demand the use of LVDS. While a single SPI master interfacing with multiple slaves is generally feasible, the implementation becomes complex with LVDS due to the need for extra circuitry to avoid bus contention, especially for multiple transmitting devices. As will be detailed in subsection 3.4.1, this complexity leads us to a design decision to instantiate three separate SPI drivers on the PCB, ensuring each device has its dedicated interface, thus simplifying the connection and avoiding potential issues associated with bus contention.

The use of a Commercial Off-The-Shelf (COTS) driver chip was preferred, as they are readily available and easy to use. The COTS driver chips are also more reliable than a custom implementation, as they are designed and tested by the manufacturer. They are also known to be more cost-effective than a custom discrete parts implementation, as they are mass-produced and the cost of the chip is negligible compared to the cost of the whole system.

Two Integrated Circuits (ICs) were selected for the application, one as a receiver for the Master In Slave Out (MISO) signals, and the other as a bus driver for the Master Out Slave In (MOSI), Clock (CLK) and Chip Select (CS) signals. The MISO receiver is the SN65LVDT2DBV [25], while the bus driver is the SN65LVDS1DBV [25]. Both ICs are manufactured by Texas Instruments, and they are available in a small SOT-23 package. The ICs are designed to be used together for the same bus.

The LVDS standard requires the use of a termination resistor at the end of the bus. The termination resistor is required to avoid signal reflections on the bus, which would cause the signal to be distorted and the communication to fail. The termination resistor is placed at the receiver end of the bus. The receiving IC features an internal 100Ω termination resistor

2.10. OneWire bus driver

The 1-Wire bus driver in the system has to support the parasitic power mode, as the 1-Wire devices are powered by the bus itself. In this functioning mode, the ID chips are powered by the bus strong pull-up during the communication, where they get the energy for the transaction and then they execute the transaction on the bus, on a single wire.

The 1-Wire bus driver's logic is implemented in the Zynq-7000 FPGA, but it needs hardware support, in particular, to ensure the strong pull-up and the needed drive strength for the bus.

To design the driver, different options were available. Both COTS driver ICs and custom discrete-components driver were considered. Since the COTS drivers are usually translating between two different interfaces, meaning that they would need a different interface to be implemented with its software support, it was decided to adopt the 1-Wire IP from CERN and to instantiate the supporting hardware.

The driver is made of three discrete MOS transistors, one for controlling the low resistance pull-up part required for the parasitic powering of the devices on the bus and two for the pull-down of the voltage level of the bus. Additionally, a resistor between the bus and the positive supply voltage rail pulls up the bus during the communication. During the transactions, the bus is pulled to $3.3V$ by `R3`. The driver connects to the bus via a 100Ω resistor to protect the driver and the system from shorts on the bus.

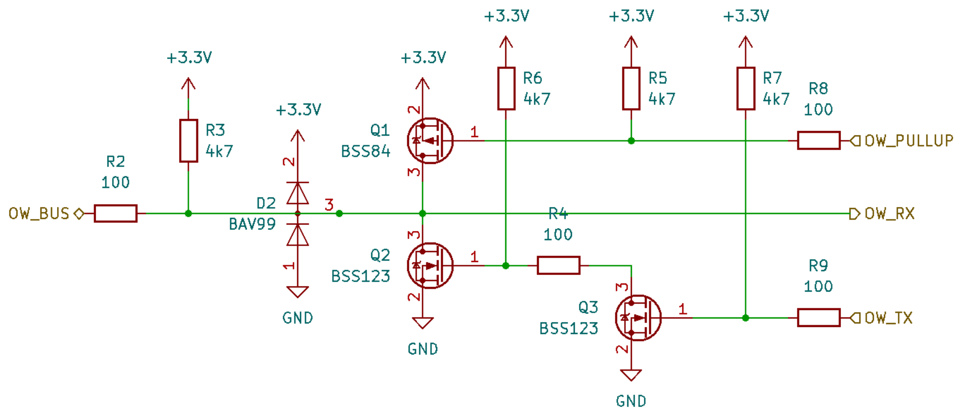


Figure 2.1: The 1-Wire bus driver circuit.

2.11. Voltage monitoring

The supply rails have to be monitored to ensure that the system is operating within the specified limits. The supply rails are monitored using the ADC on the Cora board. The ADC is a 12-bit ADC with 17 channels, 14 of which are available on the Cora pin headers, 6 of which are single-ended and 8 of which are differential. The ADC can accept voltages up to 3.3V on the single-ended channels and up to 1V on the differential channels. The ADC is integrated in the SoC. For the purpose of voltage monitoring, the ADC is configured to operate in single-ended mode.

Since the rails to be monitored are: $+5V$, $+12V$, $+15V$ and $-15V$, the voltages need an analog circuit to be scaled down to the maximum input voltage of the ADC. The voltages are scaled down using a voltage divider. The voltage divider features also a low-pass filter to avoid aliasing in the ADC. The filter is then connected to a buffer to avoid loading the voltage divider, as the native ADC range is $1V$, and the input is scaled on the Cora board with another resistive divider, as shown in [16], the signal sees an impedance of about $3.3k\Omega$. The buffer is then implemented using two low noise dual operational amplifiers, the AD8658 [13].

The $+5V$ rail

The $+5V$ rail must be scaled in the $3.3V$ range. A ratio of ½ is deemed suitable, as it translates the $5V$ into $2.5V$ and allows enough room around it to measure deviations from the nominal value. It is implemented by $R11 = R15 = 2.2k\Omega$. A $1\mu F$ capacitor is used to filter the signal, which yields a pole at $f = {}^{1}/_{[2\pi(R11//R15)C20]} = 114.7Hz$. Finally, the buffer is implemented using the AD8658 [13].

The $+12V$ rail

To scale the $+12V$ rail, a divider with two resistors, $R10 = 12k\Omega$ and $R14 = 2.2k\Omega$, is used, yielding a ratio of ${}^{2.2k\Omega}/_{14.2k\Omega} = 0.1549$. The output voltage is then $V_{out} = G_{12V} \cdot V_{in} = 0.1549 \cdot V_{in}$, so it is scaled down for the nominal $12V$ to $1.86V$. The filter is implemented using a $1\mu F$, that gives a pole at $f = {}^{1}/_{[2\pi(R10//R14)C19]} = 85.5Hz$.

The $+15V$ rail

To scale the $+15V$ rail, a divider with two resistors, $R12 = 12k\Omega$ and $R16 = 2.2k\Omega$, is used, yielding again a ratio of ${}^{2.2k\Omega}/_{14.2k\Omega} = 0.1549$. This is done to reduce the part count on the Bill of Materials (BOM) and to simplify the design. The output voltage is then $V_{out} = G_{15V} \cdot V_{in} = 0.1549 \cdot V_{in}$, so it is scaled down for the nominal $15V$ to $2.32V$. The filter is implemented using a $1\mu F$, that gives a pole at $f = {}^{1}/_{[2\pi(R11//R15)C20]} = 85.5Hz$.

The $-15V$ rail

Since the $-15V$ rail is negative, an offset of $3.3V$ is applied at the voltage divider. This way the division has a ratio offers $G_{-15V} = {}^{2.2k\Omega}/_{14.2k\Omega} = 0.1549$ and the output voltage is $V_{out} = G_{-15V} \cdot (V_{in} - V_{offset}) + V_{offset} = G_{-15V} \cdot V_{in} + V_{offset}\,(1 - G_{-15V})$

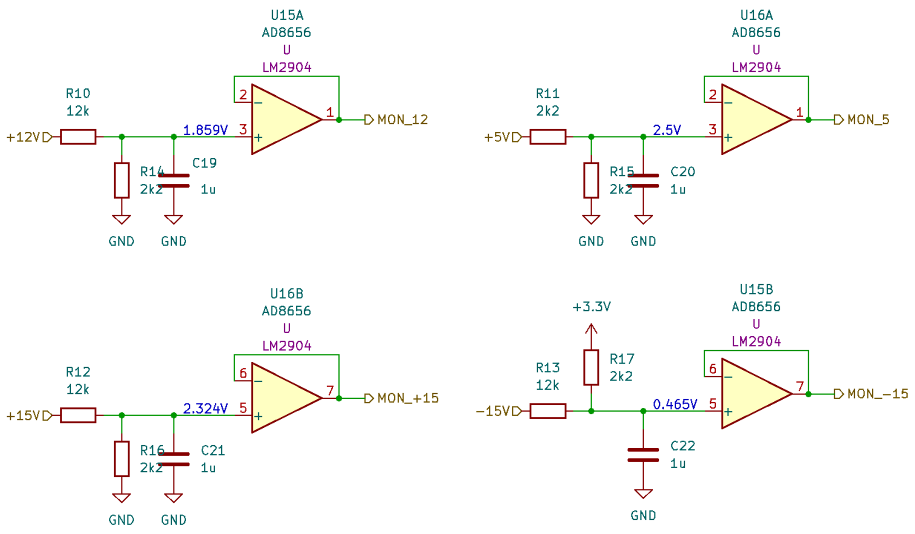


Figure 2.2: The voltage monitoring circuit.

2.12. Ethernet interface

The Ethernet interface is already present on the Cora board, and it is implemented using the SoC integrated Ethernet controller.

The interface is presented to the front panel by the RJ-45 jack on the Cora board, the module is then mounted with the Ethernet side aligned to the edge of the carrier, so that the Ethernet jack is accessible from the front panel.

2.13. Powering

The controller is powered by the CDC power supply. Since the Cora supports $5V$ power input, the $5V$ voltage rail is used to power the module. All the needed voltage in the module are then derived from the $5V$ rail.

The circuitry on the carrier board is powered by $3.3V$ that are generated in the carrier board from the $5V$ rail. Since the current requirement for the onboard circuitry is minor, the $3.3V$ rail is generated using a linear regulator IC. In the first design, the LM1117-3.3 IC was selected, but due to availability and package concerns, the IC was changed to the LT1963A-3.3 IC, which is largely available.

2.14. Power domain separation

Due to the two different 3.3V power domains, some power domain separation was required. In particular, the LVDS interfaces are powered by the carrier board, but receive digital signals from the Cora's 3.3V power domain. Similarly, the 1-Wire bus driver is powered by the carrier board, but it receives digital signals from the Cora's 3.3V power domain.

To avoid conflicts, backfeeding of power and potential cascade failures, a voltage level translator is inserted between the two domains. The voltage level translator is implemented using a COTS IC, the SN74LVC8T245 IC [24]. The IC is a bidirectional voltage level translator with 8 channels. Since each SPI bus needs four signals, and the 1-Wire bus needs four, at least two voltage-level translators are needed. Furthermore, since the chip allows to select the direction of translation only on a chip-wide level, the direction of the signals are to be considered.

Each SPI bus needs three signals from the Cora to the carrier (MOSI, CS and CLK) and one signal from the carrier to the Cora (MISO). The 1-Wire bus needs two signals from the Cora to the carrier (pull-up and transmit) and one signal from the carrier to the Cora (receive).

This accounts to a total of 10 signals from the Cora to the carrier and 6 signals from the carrier to the Cora. This means that more than two voltage-level translators are needed. The solution is to use three voltage level translators, two for the Cora to carrier signals and one for the carrier to Cora signals.

2.15. PCB layout

At last, the circuit was implemented in a PCB. The PCB was designed using KiCad, an open-source Electronic Design Automation (EDA) suite. The PCB was designed to fit inside a 3U (100mm) Eurocard slot, with a length of 220mm. This is the size of all modules in the CDC. The mechanical constraints were the position of the backplane connector, the position of the module that allows access to the Ethernet jack from the front, and some cutouts to allow for some parts of the Cora module that are taller than the connectors so that the Cora can properly connect without interference.

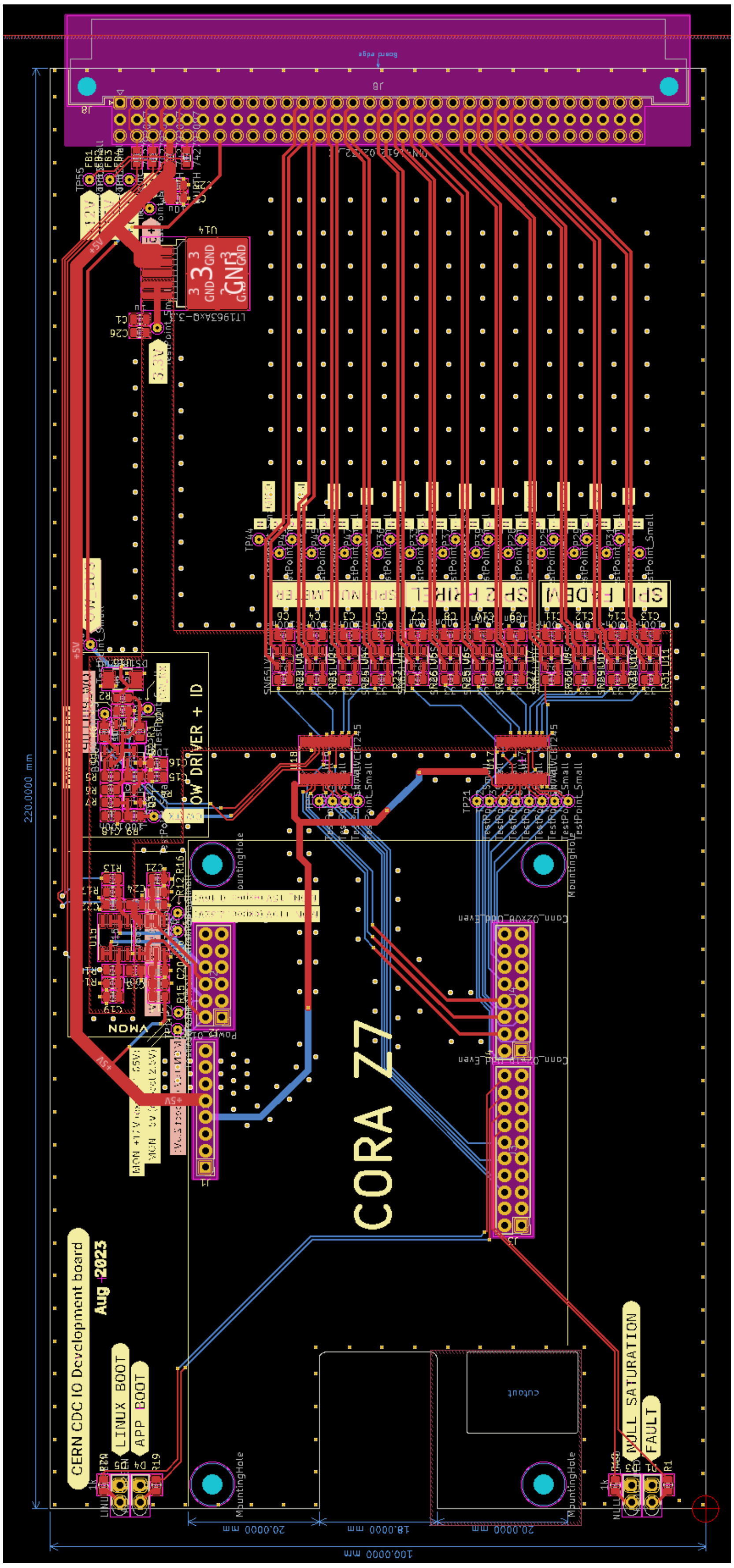


Figure 2.3: The PCB layout that was used for the development, later superseded by the final layout.

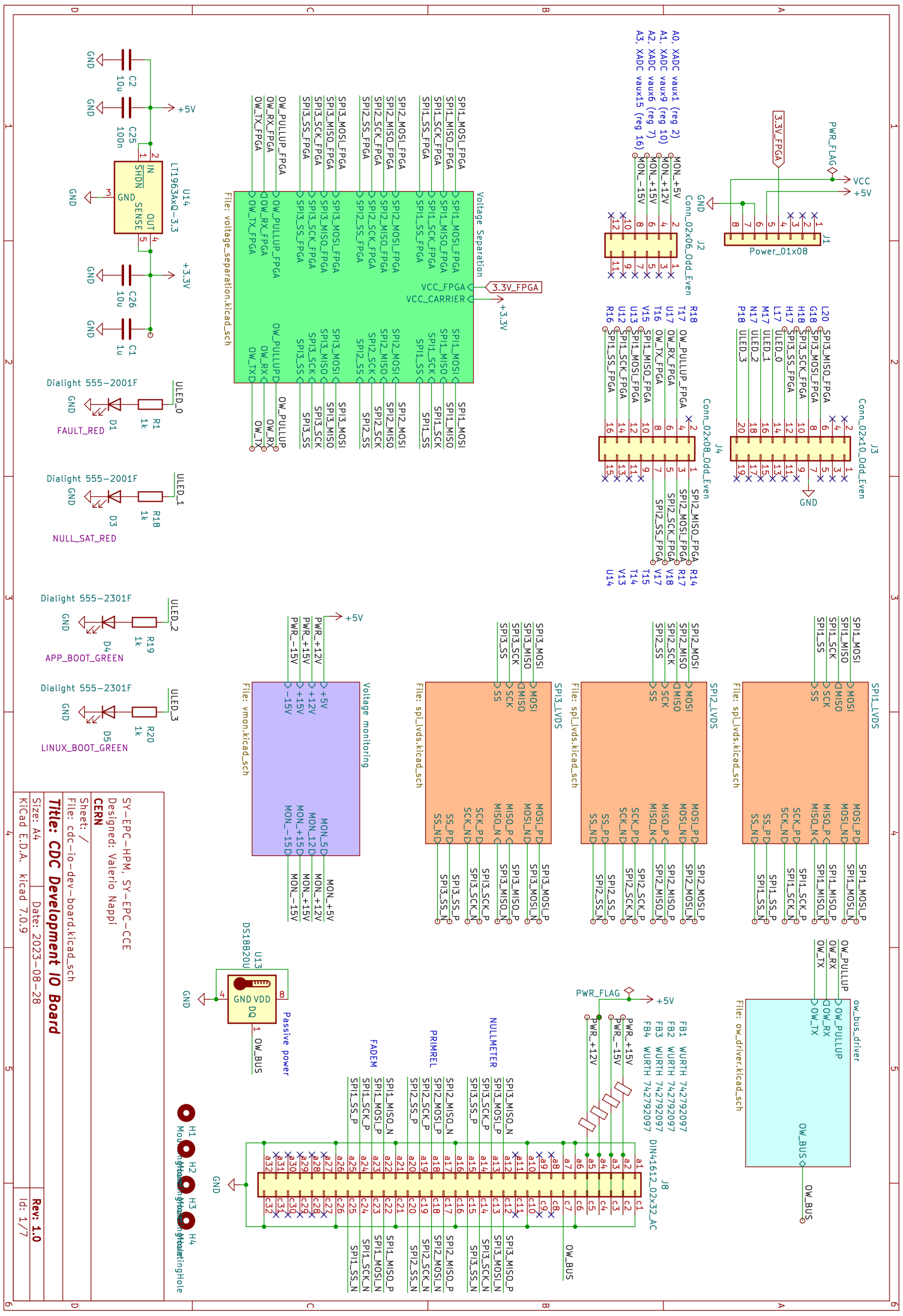


Figure 2.4: Carrier board electrical schematic

3 | Gateware

As defined in subsection 1.4.5, the system will need to be interfaced with the hardware as follows:

- One SPI bus to communicate with the FADEM module, over LVDS.
- One SPI bus to communicate with the PRIMREL module, over LVDS.
- One SPI bus to communicate with the NULLMETER module's ADC, over LVDS.
- One 1-Wire bus to check ID chips and temperature of the modules.
- Four ADC channels to monitor the four power rails.

Some interfaces can be implemented using the FPGA vendor's built-in Intellectual Property (IP) libraries, while others will need to be implemented in VHDL.

VHSIC Hardware Description Language (VHDL) is a hardware description language that is used to describe digital circuits. It is used to describe the behavior of the hardware and not the structure. The structure is inferred from the behavior. The VHDL code is then synthesized into a netlist, which is a list of components and connections between them. The netlist is then mapped to the FPGA resources and routed. The VHDL code can be simulated to verify the behavior of the hardware before synthesis.

The following sections will describe the interfaces and the IP that will be used to implement them.

The IPs will connect to the Processing System (PS) through an Advanced eXtensible Interface (AXI) interface, which is the standard interface for the PS to communicate with Programmable Logic (PL) IPs. The AXI interface is then accessible from the PS in the memory map and can be accessed as simple memory addresses.

3.1. Block Design Overview

The block design is shown in Figure 3.1. The AXI interface is routed from the PS to the IPs through the AXI interconnect. The purpose of the AXI interconnects is to allow one master to communicate with multiple slaves. The AXI interconnect is then connected to all the peripherals in the PL. The AXI bus is connected to three instances of the SPI IP, to the 1-Wire IP, to two instances of the AXI GPIO IP for controlling status LEDs, and to one Xilinx Analog to Digital Converter (XADC) IP.

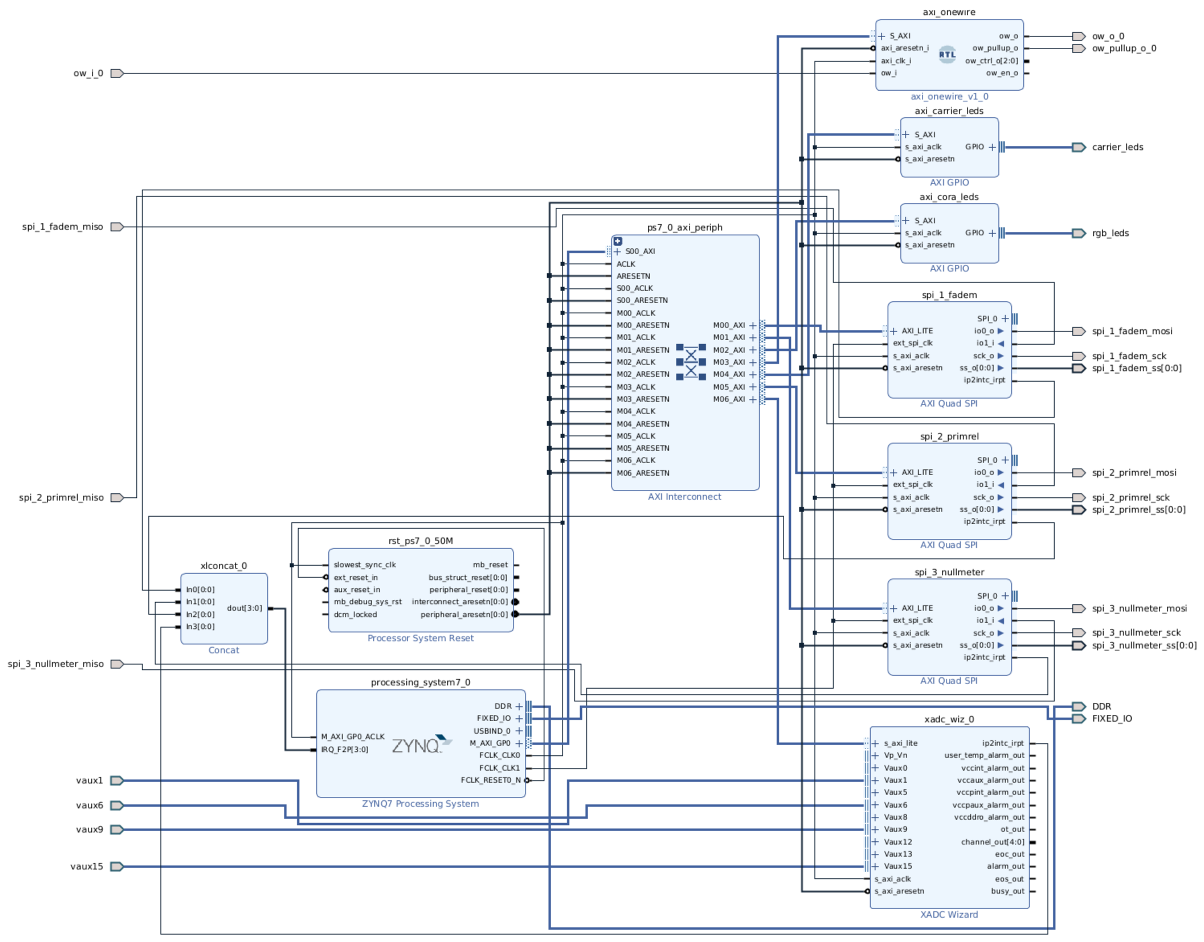
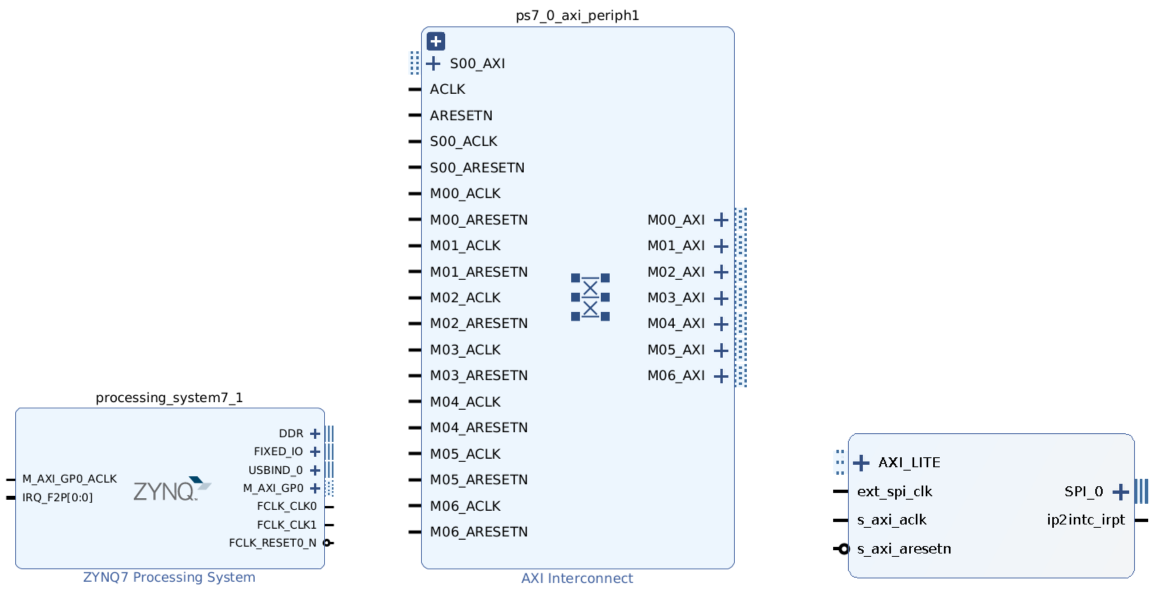


Figure 3.1: Block design overview

3.1.1. IPs

IPs are pre-made modules that can be used in a design. They are usually provided by the FPGA vendor, but can also be provided by third parties. Some of the IPs for this project were developed internally by the CERN SY-EPC - Converter Control Electronics section (SY-EPC-CCE) team, and some were provided by Xilinx.

For this project, there was no need to develop any IPs from scratch, as the interfaces were standard enough that the ones provided by Xilinx and CERN were sufficient. The main tasks associated with the gateware were to configure the IPs and integrate them into the design, by generating the interfaces, memory maps, clocks and resets, and connecting them to the AXI interconnect.



(a) Zynq-7000 PS IP block (b) AXI interconnect IP block (c) SPI IP block

Figure 3.2: Combined figure of IP blocks

3.2. PS IP

The core of the block design is the IP that represents the PS (Figure 3.2a). This building block does not actually get instantiated as programmable logic but rather configures the PS hardware to behave as set.

Many features of the PS are configured from this block, such as:

- Clock generation;
- Definition of the AXI interfaces between the PS and the PL;
- Universal Asynchronous Receiver/Transmitter (UART) configuration for the Linux console;
- Interrupts between the PS and the PL;
- Ethernet interface in the PS;
- Definition of the GPIO interfaces.

3.3. AXI interconnect

The purpose of the AXI interconnect (Figure 3.2b) is to allow multiple AXI peripherals to be connected to the PS AXI interface. The AXI interconnect is configured to have one master and multiple slaves.

The master can select between the slaves by writing to the AXI interconnect's memory map. The AXI interconnect then forwards the read or write request to the selected slave. The AXI interconnect also handles the AXI protocol, which is a standard protocol for AXI peripherals to communicate with the PS.

It is understood that, for this to be possible, the memory interface address space for the various IPs must be reasonably small compared to the available address range. Typically, the IPs used in this application, has a 64kB address space, which is small compared to the 1GB address space dedicated to each AXI port.

The system described here makes use only of the AXI0 port, so all addresses start from `0x4000_0000` as per Figure 3.3

This translation is transparent for the IPs, as they are configured to have a 64kB address space, and the AXI interconnect handles the translation between the AXI address space and the IP address space.

3.4. SPI IP

The SPI requirements were for a simple interface with a single data line, with three devices connected to it. The SPI IP (Figure 3.2c) provided by Xilinx was sufficient for this application.

3.4.1. Multislave SPI

As noted in the hardware chapter, the physical layer requirement for the SPI interfaces to operate with LVDS complicates the use of a multidrop SPI configuration. Although SPI inherently supports multiple slaves, implementing this with LVDS becomes challenging. Particularly, the contention that arises with multiple transmitting devices requires additional circuitry to mitigate.

Recalling the discussions from the previous chapter, to circumvent these complexities and to adhere to the physical layer specifications, we opted to instantiate three separate SPI IPs. Each SPI IP is dedicated to a specific device, facilitating a direct connection to the

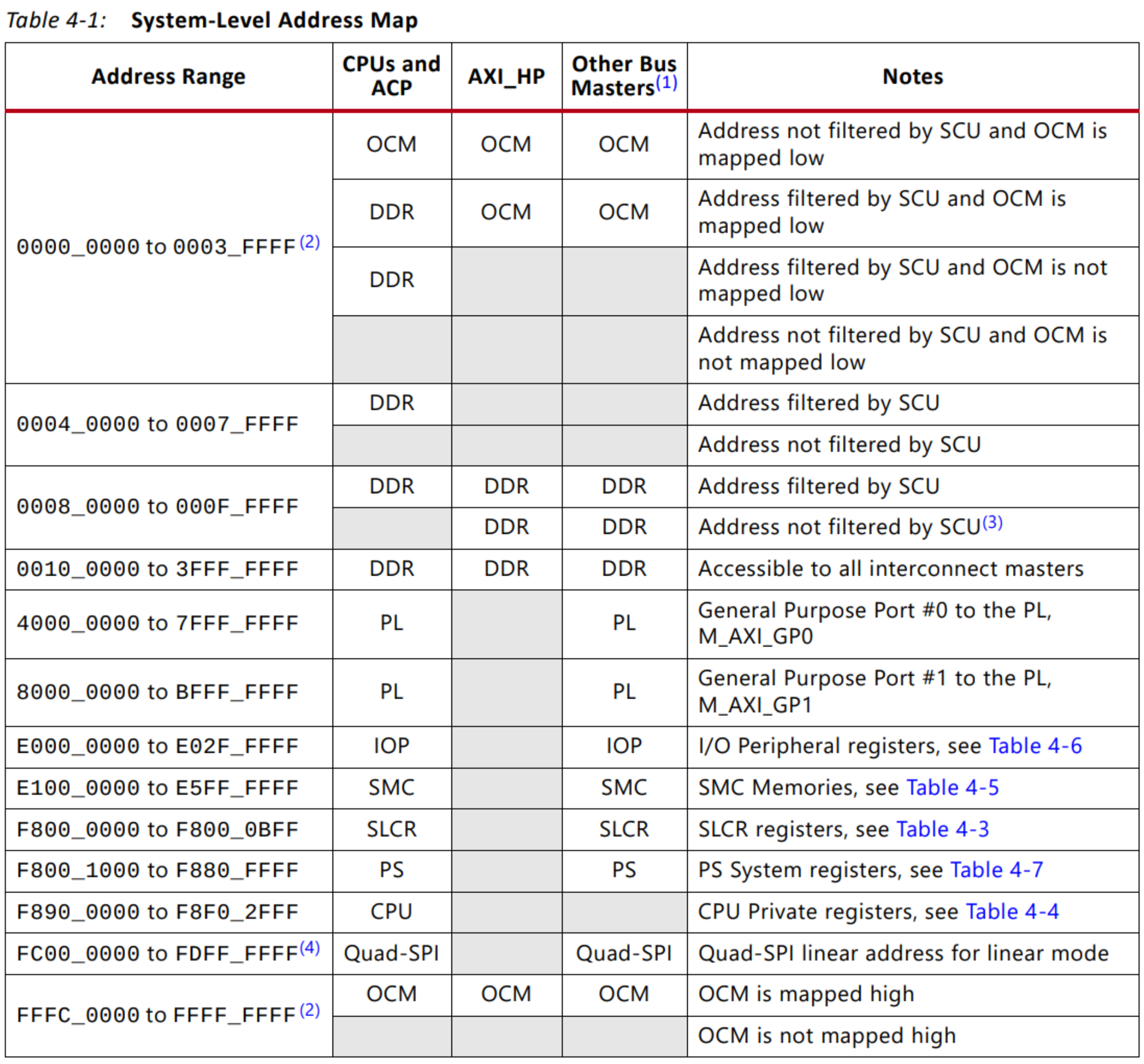


Table 4-1: **System-Level Address Map**

Address Range	**CPUs and ACP**	**AXI_HP**	**Other Bus Masters**(1)	**Notes**
0000_0000 to 0003_FFFF(2)	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU(3)
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF(4)	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF(2)	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Figure 3.3: Zynq-7000 memory map, from the Zynq-7000 Technical Reference Manual [39]

FPGA pins without the need for complex circuitry or bus management protocols.

3.4.2. LVDS Physical Layer

As mentioned in section 2.9, the project employs LVDS for the SPI control, overcoming the Zynq SoC's native limitation with the aid of external LVDS drivers. While the Zynq SoC and FPGAs in general can be tailored to support LVDS signaling through the instantiation of IO transceivers [38], the specific conditions of our project presented additional challenges.

IO bank voltages

FPGAs IO banks are powered externally, offering flexibility in IO voltage levels, a feature detailed in our chosen board's schematic. In our case, the IO banks are powered by a 3.3V supply, restricting the use of native LVDS drivers which require 2.5V or 1.8V supplies for the `LVDS_25` and `LVDS` transceivers respectively.

Hence, the implementation of LVDS translation was accomplished using external LVDS drivers, compatible with the 3.3V supply. These drivers facilitate the required LVDS signaling while conforming to the IO bank's voltage specifications. The IO standard selected is `LVCMOS33`, aligning with the 3.3V IO banks. Configuration details are discussed in the FPGA constraints file, as highlighted in section 3.8.

3.4.3. SPI IP configuration

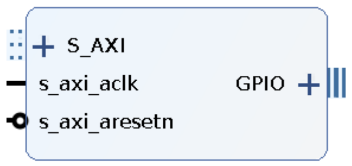
The configuration for this IP is relatively simple, as it only needs to be configured as a master, with a single data line. The configuration is detailed in [41]. The transactions on the SPI bus for the FADEM and PRIMREL modules are detailed in subsection 5.5.1. Both modules use 16-bit words, and the SPI IP is compatible with this word size. The Linux driver for the IP is not compatible with 16-bit transactions, so the IP was configured to use 8-bit words, as two 8-bit words are equivalent to one 16-bit word.

The frequency of the SPI bus is set to 1MHz, which is the frequency supported by the FADEM and the PRIMREL modules. The SPI IP supports frequencies up to several MHz, so this is not a limitation. The frequency is set by generating a PL clock of 2MHz from the Zynq PS IP and setting the minimum divider of 2 in the SPI IP.

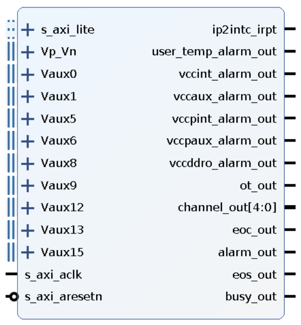
A hardware First In First Out (FIFO) queue is enabled to provide some buffering, with a depth of 16 words.

At last, the interrupt signal is connected from the SPI IP to the AXI interconnect so that the PS can be notified when a transaction is complete. Since the PS interrupt port is for a vector of interrupts and the SPI IP only has one interrupt, the interrupts are fed in a concatenation IP that bundles the interrupts into a vector.

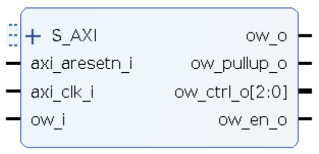
3.5. AXI GPIO IP



(a) AXI GPIO IP block



(b) XADC IP block



(c) OneWire IP block

Figure 3.4: Combined figure of additional IP blocks

Two GPIO IPs (Figure 3.4a) are used in this application, one for the status LEDs on the development board, and one for the status LEDs on the carrier board.

Two Red Green Blue (RGB) LEDs are on the development board, so six GPIOs are instantiated to control those. Four more GPIOs are instantiated to control the four status LEDs on the carrier board.

The GPIO IP are connected to the AXI interconnect, and mapped in the PS memory map. The GPIO IP is configured to have a 64kB address space.

3.6. XADC IP

The Zynq SoC features an ADC that can be used to monitor the voltage rails, which is called XADC. The XADC is implemented in hardware and can be controlled from the PS and the PL.

An IP (Figure 3.4b) is instantiated in the FPGA fabric that connects the XADC to the AXI bus.

The configuration of the XADC IO sets the interface to AXI and enables the necessary channels. The XADC actually contains one ADC, and the channels are multiplexed to the

ADC. The ADC is 12-bit and supports differential inputs. The ADC is configured to have a sampling frequency of 1MHz, which is the maximum sampling frequency supported by the ADC, sequencing through the four channels.

3.7. OneWire IP

The 1-Wire IP (Figure 3.4c) is used to communicate with the 1-Wire bus. The IP is provided by CERN, as it is in use in other projects such as the FGC.

The IP is instantiated in the FPGA fabric and connected to the AXI bus. The configuration for this IP is elementary. The frequency of the clock is given as a parameter, and the IP is configured to have a 100MHz clock. The IP has a few IO ports:

- `S_AXI`: the AXI interface;
- `axi_aresetn_i`: the AXI reset;
- `axi_clk_i`: the clock;
- `ow_i`: the signal from the read buffer of the 1-Wire bus;
- `ow_o`: the signal to the write buffer of the 1-Wire bus;
- `ow_en_o`: the signal to enable the write buffer of the 1-Wire bus;
- `ow_pullup_o`: the signal to enable the *strong pullup* of the 1-Wire bus;
- `ow_ctrl_o`: this signal is used in complex applications where multiple 1-Wire buses are used, and it is not used in this application.

The IP supports triplicated Block RAM (BRAM), which is used in radiation-hardened applications. This feature is not used in this application, so the IP is configured to use a single BRAM.

3.7.1. LFSRs

To generate the timing for the 1-Wire bus, the IP uses a Linear Feedback Shift Register (LFSR). The LFSR is a shift register that is clocked at a high frequency, and the output is a pseudo-random sequence. The LFSR is used to generate the timing for the 1-Wire bus, as they are efficient and simple to implement.

The LFSR is implemented in PL logic, and the IP is configured to have a 100MHz clock, so the correct LFSR polynomial is selected to generate the proper 1-Wire timing from a 100MHz clock.

The calculation of the configuration and seed of the LFSR is quite complex, and the IP simulation suite is designed to calculate the correct configuration and seed for a given clock frequency and output it. At this point, the configuration can be copied in the IP configuration.

3.8. Constraints files

The FPGA constraints file is used to configure the FPGA resources, such as the IO standards, the IO bank voltages, the clock frequencies, and the pin locations. The constraints file is a text file that is read by the FPGA synthesis tool, and it contains the configuration for the FPGA resources.

As no other IO standard is available for the IO banks used in this application, the IO standard is set to `LVCMOS33`, which is the standard for 3.3V IO banks. The IO bank voltages are set to 3.3V, as this is the voltage supplied to the IO banks on the board. The clock frequencies are set to the frequencies used in the application, and the pin locations are set to the pins used in the application.

The pin locations are bound by the hardware, both by the available pins in the development board and by the layout of the carrier board, which is influenced by the pinout of the connectors.

4 | Software

This chapter describes the software architecture of the system. The software is divided into three main parts: the Linux operating system, the bootloader, and the application. The Linux operating system is the core of the system and is responsible for managing the hardware and providing an interface for the application. The boot loader is responsible for loading the Linux kernel into memory and starting the system. The application is the main part of the system and is responsible for controlling the hardware and providing an interface for the user. The application also contains userspace drivers to facilitate access and communication to the hardware.

4.1. Petalinux

Petalinux is a toolchain for building embedded Linux systems. It is based on the Yocto project and was developed by Xilinx. Petalinux is used to build the Linux operating system for the system. The toolchain is based on the Yocto project and uses the OpenEmbedded build system. The build system is based on the BitBake tool, which is used to build the Linux operating system. The build system is configured using recipes, written in Python, which are used to specify the build process. Recipes are organized in layers, where each layer contains a set of recipes.

A Petalinux project serves to integrate Yocto with Xilinx's technology on FPGAs. Typically a Petalinux project is created by importing a Hardware Description File (HDF) which is generated by Vivado. The HDF contains information about the hardware and is used to configure the Linux kernel.

The output product is a system image that contains a bootloader, a compiled kernel, and a filesystem image (rootfs).

4.1.1. Layers

Layers are used to organize the recipes. A layer is a directory that contains recipes and configuration files. A layer can be added to a project to add recipes to the project.

Layers are organized in a hierarchical structure, where the bottom layers contain the base recipes for Yocto, and then each subsequent layer contains recipes that are more fine-grained to the project, for example, silicon vendor layers, board vendor layers, BSP layers, application layers, etc.

A layer can be used to override recipes from another layer. This is useful when a recipe from a layer needs to be modified. The layer is added to the project and the recipe is modified in the layer. This way, the recipe is not modified directly and can be updated without losing the modifications. The default Petalinux project contains a set of layers that are used to build the system. The layers are organized in a hierarchy, where each layer can override recipes from the layers below it.

4.1.2. Petalinux project structure

A typical Petalinux project looks like this, where the tree is truncated to show only the relevant files:

```
petalinux-project/
└── project-spec/
    ├── configs/
    │   ├── config
    │   └── rootfs_config
    └── meta-user/
        ├── conf/
        │   ├── layer.conf
        │   ├── petalinuxbsp.conf
        │   └── user-rootfsconfig
        ├── meta-xilinx-tools/
        ├── recipes-bsp/
        │   ├── device-tree/
        │   └── u-boot/
        ├── recipes-core/
        │   └── dropbear/
        └── recipes-kernel/
            └── linux/
```

The project directory contains the Petalinux project. The project-spec directory contains the configuration files for the project. The configs directory contains the configuration files for the Linux kernel and the rootfs. The meta-user directory contains the recipes for the system. The meta-xilinx-tools directory contains the recipes for the Xilinx tools and customizations. The recipes-bsp directory contains the recipes for the Board Support Package (BSP) layer, in this case, the bootloader and the device tree. The recipes-core directory contains the recipes for the rootfs. The recipes-kernel directory contains the recipes for the Linux kernel. Additionally, application recipes can be added to the project in the recipes-apps directory.

A typical recipe looks like this:

```
recipes-apps/
└── hello-world/
    ├── files/
    │   └── hello-world.c
    └── hello-world.bb
```

The files folder contains the source code for the application. The hello-world.bb file contains the recipe for the application. The recipe specifies the name of the application, the version, the source code and the dependencies. The recipe also specifies the build process and the install procedure for the application. The recipe is written in Python and is used by the build system to build the application.

The recipe for the application in `hello-world.bb` is listed in 4.1.

Figure 4.1: Recipe for the hello-world application

```
SUMMARY = "Hello world application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://hello-world.c"

S = "${WORKDIR}"

do_compile() {
  ${CC} ${LDFLAGS} hello-world.c -o hello-world
}

do_install() {
  install -d ${D}${bindir}
  install -m 0755 hello-world ${D}${bindir}
}

FILES_${PN} += "${bindir}/hello-world"

```

The first section of the recipe specifies the name of the application, the version, the section and the license. Note that it is mandatory, for any recipe, to specify a license. This is because the build system checks that all the licenses of the dependencies are compatible with the license of the application.

The second section specifies the source code files for the application. The third section specifies the build directory, named S, where the source code is copied. The fourth section specifies the build process, in this case the compilation. The fifth section specifies the install process, in this case it is made of two steps: creating the directory where the application will be installed and copying the application to the directory. The sixth section specifies the files to be installed.

When the system build is started, the build system will build each recipe in the project, and bundle it in the system image.

Recipe appends

Another way to customize the build process is to use recipe appends. A recipe append is a recipe that is used to modify the build process of another recipe without modifying the recipe itself. This is useful when the recipe is provided by a layer and cannot be modified. A recipe append is written in Python and is used by the build system to modify the build process of a recipe.

For the CDC network controller, some customization had to be done to the kernel. A bbappend recipe is used to specify configurations - such as drivers and features to include - and patches to apply to the kernel. The bbappend recipe is written in Python and is used by the build system to modify the build process of the kernel.

4.1.3. Petalinux tools

Petalinux includes a set of customization of the Yocto project, including the automatic creation and configuration of a base project that includes the custom Xilinx bootloader and kernel, automatically compiles the device tree from the hardware description files and automatically configures the kernel. The tools also package the output files into a bootable image, which can be loaded onto the system.

The designer is left with the customization of the system to their needs. This includes the configuration of the rootfs, the addition of applications, the addition of custom drivers and the settings of the device tree.

4.1.4. Bootloader

The Zynq-7000 platform makes use of a multi-step boot process. The first step is the bootROM, which is a small piece of code that is stored in the bootROM of the Zynq-7000 platform. The bootROM is responsible for loading the First Stage Boot Loader (FSBL)

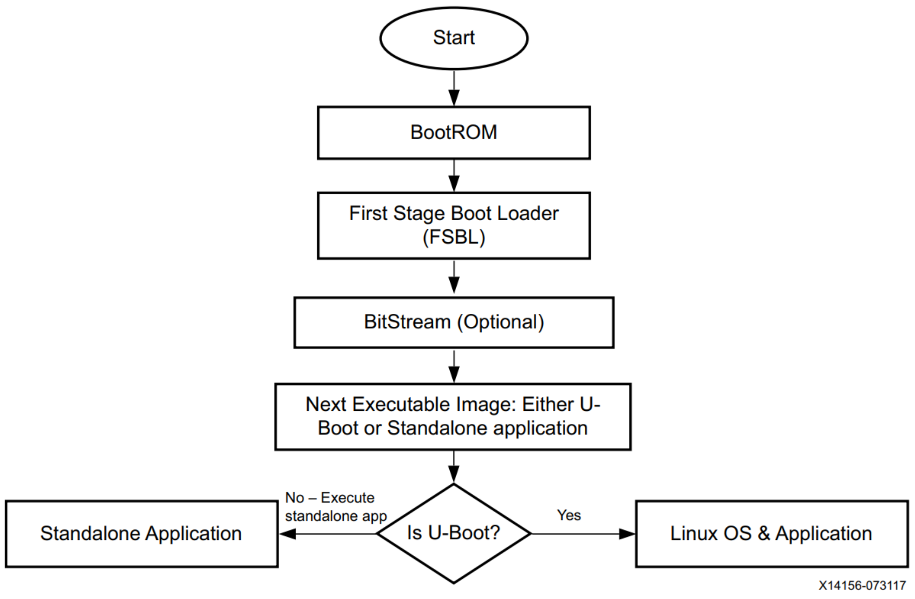

Start
BootROM
First Stage Boot Loader (FSBL)
BitStream (Optional)
Next Executable Image: Either U-Boot or Standalone application
Is U-Boot?
No – Execute standalone app
Standalone Application
Yes
Linux OS & Application
X14156-073117

Figure 4.2: Boot process of the Zynq-7000 platform. Figure 4-6 from [40].

into memory and starting it. The FSBL is a small piece of code that is stored in the boot flash of the Zynq-7000 platform. The FSBL is responsible for loading the bitstream into the PL and either loading the baremetal application or u-boot in the PS.

U-boot is a bootloader that is used to load the Linux kernel into memory and start the system. It is very common in embedded systems and is used in the Zynq-7000 platform. Petalinux makes use of a Xilinx-customized u-boot. U-boot can be configured to load the Linux kernel from a variety of sources, including a Trivial File Transfer Protocol (TFTP) server, a Universal Serial Bus (USB) drive, an Secure Digital (SD) card, etc. The Linux kernel is loaded into memory and started by u-boot.

4.1.5. Linux Kernel

The Linux kernel is configured using a Device Tree Source (DTS) file which is generated by Petalinux from the HDF, and integrated by Device Tree Source Include (DTSI) files, which modify the device tree. It contains information on the peripherals that are available and how they are connected, together with the configuration of the peripherals and the driver to use.

The Device Tree Blob (DTB) is a compiled version of the DTS and is generated by merging the information from the HDF with the information from the recipes, where the designer does the configuration. The DTB is used by the Linux kernel to configure the hardware and to load the appropriate drivers. The DTB is also used by the bootloader to configure the hardware before the Linux kernel is loaded.

In the case of the CDC, the DTSI file that configures the PL side is configuring the SPI devices to be of type *"cern,hpc-cdc-fadem"*, *"cern,hpm-cdc-primrel"* and *"cern,hpm-cdc-nullmeter"*. These strings tell the kernel which drivers to associate to the devices.

Listing 4.1: Device tree snippet for the SPI devices

```
&spi_1_fadem {
    status = "okay";
    spidev@0 {
        compatible = "cern,hpm-cdc-fadem";
        reg = <0>;
        spi-max-frequency = <1000000>;
    };
};

&spi_2_primrel {
    status = "okay";
    spidev@0 {
        compatible = "cern,hpm-cdc-primrel";
        reg = <0>;
        spi-max-frequency = <1000000>;
    };
};

&spi_3_nullmeter {
    status = "okay";
    spidev@0 {
        compatible = "cern,hpm-cdc-nullmeter";
        reg = <0>;
        spi-max-frequency = <1000000>;
    };
};
```

It is evident how the *compatible = "cern,[...]"* strings are custom to the CDC and are not part of the Linux kernel. This is because the *spidev* driver requires the user to patch itself to specify the kind of device for each device. For this reason, a patch is applied to the *spidev* driver to add support for the CDC devices. The patch is applied by a bbappend recipe, which is used to modify the build process of the kernel. The patch adds the strings *"cern,hpc-cdc-fadem"*, *"cern,hpm-cdc-primrel"* and *"cern,hpm-cdc-nullmeter"* to the *spidev* driver. The DTSI file is then modified to use these strings to configure the SPI devices.

Listing 4.2: Device tree snippet for the SPI devices

```
diff --git a/drivers/spi/spidev.c b/drivers/spi/spidev.c
index 1bd73e322b7b..e67522e51e8d 100644
--- a/drivers/spi/spidev.c
+++ b/drivers/spi/spidev.c
@@ -682,6 +682,9 @@ static const struct spi_device_id spidev_spi_ids[] = {
  { .name = "m53cpld" },
  { .name = "spi-petra" },
  { .name = "spi-authenta" },
+ { .name = "hpm-cdc-fadem" },
+ { .name = "hpm-cdc-primrel" },
+ { .name = "hpm-cdc-nullmeter" },
    {},
};
MODULE_DEVICE_TABLE(spi, spidev_spi_ids);
@@ -696,6 +699,9 @@ static const struct of_device_id spidev_dt_ids[] = {
  { .compatible = "menlo,m53cpld" },
  { .compatible = "cisco,spi-petra" },
  { .compatible = "micron,spi-authenta" },
+ { .compatible = "cern,hpm-cdc-fadem" },
+ { .compatible = "cern,hpm-cdc-primrel" },
+ { .compatible = "cern,hpm-cdc-nullmeter" },
  {},
};
MODULE_DEVICE_TABLE(of, spidev_dt_ids);
```

The DTSI also includes configurations for the AXI GPIO IP used for the LEDs and the XADC configuration.

The DTSI also includes a fix for a bug in Petalinux, which generates the DTS always for a two-core Zynq-7000 chip, thus referencing the second core in the DTS. This causes the DTB to fail compiling. The fix is to remove the reference to the second core in the DTSI file.

GPIO pin numbers. The GPIO pin numbers are used to configure the GPIO IP and are different from the PL pin numbers. The DTSI file is modified to use the correct pin numbers.

Listing 4.3: Device tree snippet for the CDC

```
&axi_carrier_leds {
  status = "okay";
};

&amba {
  ptm@f889d000 {
    cpu = <&cpu0>;
  };
};

&xadc_wiz_0 {
  #address-cells = <1>;
  #size-cells = <0>;
  xlnx,channels {
    VmonP5@2 {
       reg = <2>;
    };
    VmonP15@7 {
       reg = <7>;
    };
    VmonP12@10 {
       reg = <10>;
    };
    VmonN15@16 {
      reg = <16>;
    };
  };
};
```

4.1.6. Rootfs

The Root File System (RootFS) is the filesystem that is loaded by the Linux kernel. It contains the applications and libraries that are used by the system. The RootFS is configured using a RootFS configuration file, which is generated by Petalinux and customized by the user through the Petalinux RootFS configuration menu.

Packages and applications included in the RootFS can either be needed by the application, by the system or by the user, for development or to perform system maintenance.

Packages like *busybox* and *dropbear* are needed by the system to provide basic functionality, like a shell and a Secure Shell (SSH) server. Packages like *sudo* and *vim* are needed by the user to perform system maintenance and development. Packages like *python3* and *python3-datetime* or *python3-spidev* are needed by the system to run the application.

In the RootFS configuration, users are created and their passwords are set. This means that the passwords of the users are included in the source code. This raises security concerns, as the passwords are accessible by anyone with access to the source code. This security implication is significant and it will be discussed in subsection 6.2.2

4.2. U-boot

U-boot is the second stage bootloader for the system, it is started by the FSBL and is responsible for loading the Linux kernel into memory and starting it. The configuration needed for u-boot is generated by Petalinux and is customized by the user through the Petalinux u-boot configuration menu.

The customization done in this case is minimal. The only customization consists of enabling the environment variable fetch from the SD card. This is needed to properly set the Media Access Control (MAC) address. Since the Cora board does not have an onboard Electrically Erasable Programmable Read-Only Memory (EEPROM), the MAC address for the system cannot be stored there. It can be specified at compile time, but this is not ideal, as it would require a different image for each board. The MAC address can be specified in the u-boot environment variables, which are stored in the SD card. The MAC address is then read from the SD card, in a file called `uEnv.txt` and it is set in the environment variables. The MAC address is then passed to the Linux kernel to configure the network interface.

5 | Application

Note:

In this chapter, code snippets from the source code are presented. For brevity, the code snippets are not complete and some parts are omitted. The complete source code is available in [35].

The application is the main part of the system, and it is written in Python3. It is responsible for controlling the hardware, providing an interface for the user and bridging the two. The application is divided into many parts: the first part to be developed was a Python representation of the NCRP commands and responses, followed by a representation of the FGC property model. These implemented the core concept and mechanisms of the system. It was followed by the development of the NCRP command parser. A NCRP server was developed to receive commands from the client and send the response back to the client. A handler thread is responsible for interacting with the `CDC class`, which represents the CDC with all the properties and peripherals. Each property is either an instance of the base FGC property class, or a class derived from that one.

Userspace drivers are used to control the FPGA hardware from the Linux system. The SPI driver is responsible for communicating with the FADEM, PRIMREL and NULLMETER modules. The three SPI drivers rely on the `spidev` package of Python. The 1-Wire driver accesses directly the memory through the `/dev/mem` virtual device.

Two more threads, one responsible for the monitoring of the system and an Hypertext Transfer Protocol (HTTP) server complete the application. The monitoring thread is responsible for periodically reading the 1-Wire bus and the XADC voltage monitoring, and moving the CDC to a faulty state if the voltage is too low. The HTTP server is responsible for serving an HTML debug page, that shows the status of the system. The debug page provides a complete overview of the status of the system and its components, as well as the log of the system since the power-on.

```
class Error(Exception):
  def __init__(self, error_code, error_message, error_description):
    self.error_code = error_code
    self.error_message = error_message
    self.error_description = error_description
  def __str__(self):
    return f"{self.error_code} {self.error_message}"

class UnknownCmd(Error):
  def __init__(self):
    super().__init__(9, 'unknown cmd', 'Unknown command: Indicates that the command
    buffer didn\'t start with either "S " or "G " (or the lower case equivalents). This
    will not be seen when using the terminal with the editor enabled, as the editor will
    add "G " by default if neither "S " or "G " are found.')
```

Figure 5.1: Sample of Error classes

5.1. FGC NCRP module

This module contains implementation for all of the basic concepts of the NCRP protocol. The `ncrp.py` file contains the `Command` and `Response` classes, that are used to represent FGC NCRP commands and responses in the project. The `errors.py` file contains the `Error` class and all the derived classes, one for each error. The `fgc_property.py` file contains the `Property` class and a derived class.

5.1.1. NCRP errors

Many errors are defined in the NCRP protocol, some of which are used in the project. The `errors.py` file contains a base `Error` class and many derived classes. Each error is expected to have an error code, a short error string and a detailed error description. The derived classes call the base class' init method with the proper arguments, as can be seen in Figure 5.1.

The `Error` class is derived from the `Exception` class so that it can be raised as an exception. The `Error` class also contains a `__str__` method, that returns a string representation of the error. This method is used by the NCRP server to send the error to the NCRP client.

5.1.2. NCRP commands and responses

The `ncrp.py` file is the file that contains the `Command` and the `Response` classes. These are used to represent FGC NCRP commands and responses in the project. Both classes contain an init method and a convert-to-string method, used to print commands with the proper structure.

Command class

The `Command` class is made of an init method, that takes the tag, the verb, the device, the NCRP property, the transaction id, the user selector, the array and the payload as arguments. The init method validates the arguments and stores them in the class. If the arguments are not valid, an exception is raised. The `Command` class also contains a convert to string method, that converts the command to a string, that can be sent to the NCRP server. The convert to string method is used by the NCRP server to send the command to the FGC.

```
class Command:
  def __init__(self, tag, verb, device, ncrp_property, transaction_id, user_selector,
    array, payload):
    # Validate arguments
    # ...
```

Response class

As the `Command class`, the `Response class` is used to represent the various FGC NCRP responses. The `Response` class is made of an init method, that takes the tag, the error object (one of the `Error` classes) and the data, if any. The data field is used to pass the value back to the user if the command was a `get` command. It is not used for `set` commands.

The init stores the arguments in the class. The `Response` class also contains a convert to string method, that converts the response to a string, that can be sent back to the NCRP client. The convert to string method is used by the NCRP server to send the response to the FGC.

```
class Response:
  def __init__(self, tag, error=None, data=None):
    self.tag = tag
    self.data = data
    self.error = error
  def __str__(self):
    if self.error == errors.OkRsp or self.error is None:
      if self.data is None:
        return f'${self.tag} .\n\n;'
      else:
        data = str(self.data).strip("[]' ")
        return f'${self.tag} .\n{data}\n;'
    else:
      return f'${self.tag} !\n{self.error}\n;'
```

Figure 5.2: Response class

5.1.3. FGC property

The concept of FGC NCRP property was discussed in subsubsection 1.4.6. The `fgc_property.py` file contains the `Property` class and the derived `NotImplementedProperty` class, which is used as a placeholder for a property that must exist but is not yet implemented.

Property class

A property is the terminal node of a branching structure that contains one or more values as part of an array. Each property has many characteristics, such as a name, some access rights, a type, a length, a default value, a minimum and a maximum value.

The property class serves to represent a property in the system. The most minimal implementation would contain an init method, a set method that takes a command as an argument and a get method that takes a command as an argument and returns an array of values. These methods can also raise exceptions if the input is not valid or if the method encounters an error. Many utility private functions ensure the proper functioning of the class.

Getters and setters

To allow for greater flexibility, the base class allows to attach so-called *getters* and *setters*. These items are meant as one per item in the property and are not mandatory. For some properties, only the setters or only the getters can be provided. If provided, the `Property` class expects the getters and setters to be a list of functions, one for each item in the property. The getters and setters are called by the `get` and `set` methods, respectively. A getter is expected to take no argument and to return exactly one value of the type of the property. A setter is expected to take exactly one value and return nothing.

NotImplementedProperty class

The `NotImplementedProperty` class is a placeholder for a property that must exist but is not yet implemented. It is used to avoid raising an `UnknownProperty` exception when a property is not implemented and instead raises a `NotImplemented` exception. The `NotImplementedProperty` class is a subclass of the `Property` class and it overrides the `get` and `set` methods so that they return an error when called. The `NotImplementedProperty` class is very simple, as can be seen in Figure 5.3.

```
class Property:
    def __init__(self,
                 name: str,
                 access: str,
                 val_type: Callable,
                 length: int = 1,
                 getters: Optional[List[Callable]] = None,
                 setters: Optional[List[Callable]] = None,
                 min_val: Optional[Union[Any, List[Any]]] = None,
                 max_val: Optional[Union[Any, List[Any]]] = None,
                 unit: str = "",
                 description: str = "",
                 default: Optional[Union[Any, List[Any]]] = None,
                 configs_path: str = "configs/",
                 default_path: str = "defaults/"
                 ):
      # Validate access
      # Check if val_type is callable
      # Assign the arguments to the class
      # Initialize min and max values with proper length
      # Initialize the value with the default value, either from the config file or from
    the default file

    def _get_scalar_or_list(self, val: Union[Any, List[Any]]) -> List[Any]:
      """Helper method to create a list of correct length from a scalar or a list."""

    def _load_or_get_default_value(self) -> List[Any]:
      """Load the value from the config file if it exists, otherwise return the default
    value."""

    def _write_value(self, new_value: List[Any]) -> None:
      """Write the value to the config file."""

    def _expand_command_array(self, command_array: Union[None, List[Union[int, str]]]) ->
     slice:
      """Expand the array from the command and return a slice object"""

    def _validate_array_slice(self, array_slice: slice) -> None:
      """Test if the array slice is valid, by checking if start and stop are within the
    array length. Also checks that start < stop."""

    def _get(self, array_slice: slice = slice(None, None, None)) -> List[Any]:
      """Internal method to get the property value. This method is called by the get
    method."""

    def _validate_payload_length(self, command_array: Union[None, List[Union[int, str]]],
     payload_length: int) -> None:
      """Validate that the payload length matches the command array."""

    def _validate_and_cast_payload(self, new_values: List[str]) -> List[Any]:
      """Validate and cast the payload values to the correct type."""

    def _set(self, new_values: List[Any], array_slice: slice = slice(None)) -> None:
      """Internal method to set the property value."""

    def get(self, command: Any) -> List[Any]:
      """Get the property value."""

    def set(self, command: Any) -> None:
      """Set the property value."""
```

```python
class NotImplementedProperty(Property):
  "This class is used to create a property that is not implemented yet."

  def __init__(self, name):
    super().__init__(name, 'rw', str, description='This property is not implemented yet')

  def get(self, command):
    """Raise an error if the property is not implemented yet."""
    raise errors.NcrpNotImplemented

  def set(self, command):
    """Raise an error if the property is not implemented yet."""
    raise errors.NcrpNotImplemented
```

Figure 5.3: NotImplementedProperty class

5.2. NCRP command parser module

The NCRP command parser module is responsible for parsing the NCRP commands and converting them into `Command` objects. The module is made of a `parse` function, that takes a string as an argument and returns a `Command` object. The function also takes the reference to a logger object, to log errors and debug data. The `parse` function is used by the NCRP server to parse the commands received from the NCRP client.

Internally, given the regularity of the NCRP language, the `parse` function matches valid command patterns with regular expressions. The regular expressions are defined in the `gen_regex` function. The `parse` function also runs basic checks on the command, such as checking if the command contains invalid characters. The checks on the content of the command are left to the `Command` class, and to the CDC class, as the `parse` function does not have enough information to perform these checks.

Since the regular expression is quite complex in order to foresee all the possible valid commands, the regex string is quite long and complex. It is reported here for completeness, but it is not very useful to understand the code. The regex string is shown in Figure 5.4.

5.3. The CDC Application modules

The CDC application is made of many modules, each of which is responsible for a specific task. The modules are described in the following sections.

5.3.1. NCRP server

The NCRP server is a class that is designed to listen to incoming NCRP commands and to send back the responses. The NCRP server is implemented as a thread, that is started

```python
def gen_regex():
  """Validation of the regex at https://regex101.com/r/aAyMZQ/1"""

  # magic spell begins
  r_preamble = r'!'
  r_tag = r'([a-zA-Z0-9\_]{0,31})\s+'
  r_verb = r'([sgSG])\s+'
  r_device = r'(?:([a-zA-Z0-9]+)\:)?'
  r_property = r'([a-zA-Z0-9\_]*'
  r_additional_property = r'(?:\.[a-zA-Z0-9\_]+)*)'  # non capturing group
  r_transaction_id = r'(?:\<(\d*)\>)?'  # no range provided!
  r_user_selector = r'(?:\((\d\d?)\))?'  # should check range 0-31
  r_array_id = r'(?:\s*\d+\s*)?|'  # non capturing group
  r_array_fromid = r'(?:\s*\d+\s*,\s*)|'  # non capturing group
  r_array_fromid_toid = r'(?:\s*\d+\s*,\s*\d+\s*)|'  # non capturing group
  # non capturing group
  r_array_fromid_toid_step = r'(?:\s*\d+\s*,\s*\d+\s*,\s*\d+\s*)'
  r_payload = r'\s*([a-zA-Z0-9.,_\-\s]*)?'  # can either be values or options
  r_terminator = r'\n'

  r_array = r'(?:\[(' + r_array_id + r_array_fromid + \
      r_array_fromid_toid + r_array_fromid_toid_step + r')\])?'

  regex = r_preamble + r_tag + r_verb + r_device + r_property + r_additional_property + \
      r_transaction_id + r_user_selector + r_array + r_payload + r_terminator

  return regex
```

Figure 5.4: Regex string

by the main thread at the beginning of the program. The NCRP server is listening on the standard NCRP port (1905) and it is responsible for receiving connection from the client, putting the connection in a queue and creating the consumer threads that will handle the connection. The consumer threads are responsible for receiving the commands from the client, parsing them, sending them to the CDC class and sending the responses back to the client. The NCRP server is implemented in the `fgc_ncrp_server.py` file, and the code executed by the worker threads is in the `fgc_ncrp_handler.py` file.

Client queue

The client queue is implemented using a standard Python library, the `queue` library. A queue with a fixed capacity (set by the configuration system) is created at the initialization of the `Server` class and populated with tasks as the connections are received. The tasks are then pulled from the queue by the worker threads. After the task is completed, the worker signals the queue by calling the `task_done()` method. At that point, the task was already removed from the queue when it was started, but the queue is still keeping track of how many tasks are still running. When the main thread calls the `join()` method, it will wait until the queue is empty, that is until all the tasks are completed.

Worker threads

The main logic for the worker threads is implemented in the `handler.handler` function. The function takes the client TCP socket as an argument, as well as references to the CDC object, the gateway object, the logger and the configuration system. The function is responsible for receiving the commands from the client, parsing them, sending them to the CDC class and sending the responses back to the client. If a call to the set or get method of the CDC class raises a NCRP exception, the exception is caught and transformed into a `Response` object, that is sent back to the client. If a different exception is raised, the exception is logged and a generic `LogWaiting` error is sent back to the client. This prevents the system from crashing if an unforeseen error happens, and allows to keep track of the incident. If no exception is raised, an `OkRsp` response is sent back to the client, with the data if the command was a `get` command.

Another important task managed by the handler is the routing of commands to the gateway or to the CDC. As discussed in subsubsection 1.4.6, the gateway needs to be emulated by the NETCTRL module. The gateway emulation will be discussed further in the following subsection 5.3.3.

5.3.2. FGC device

Since the CDC and the gateway share the same principles, both being FGC devices, a common class describes a generic device that contains a set of properties organized in a Python dictionary, that can be accessed by a `set(command)` and a `get(command)` methods.

Furthermore, the `Device` class contains a Mutual Exclusion (MutEx) mechanism, that is implemented as a `threading.Lock` object. The lock is acquired at the beginning of the `set` and `get` methods and released at the end of the methods. The lock is used to prevent concurrent access to the device, as the NCRP protocol does not specify how to handle concurrent access to the same device. The `Device` class is implemented in the `fgcd.py` file.

Set and get methods

Both methods, after acquiring the lock, look for the property specified by the command in the `self.properties` dictionary. If no match is found, an `UnknownProperty` exception is raised, that is to be caught and transformed into a `Response` object by the handlers described in subsubsection 5.3.1. If the property is found, the `set` method calls the `set`

```python
class GatewaySerial(fgcd.Device):
  def __init__(self):
    # create the properties
    properties = {
        'CLIENT': ncrp_property.NotImplementedProperty('CLIENT'),
        'CLIENT.CMDS_HELD': ncrp_property.NotImplementedProperty('CLIENT.CMDS_HELD'),
        'CLIENT.TOKEN': ncrp_property.NotImplementedProperty('CLIENT.TOKEN'),
        # omissis ...
        'TIME.START_A': ncrp_property.NotImplementedProperty('TIME.START_A'),
    }
    super().__init__(properties)
```

method of the property, passing the command as an argument. The `get` method calls the `get` method of the property, passing the command as an argument. Both methods return the `Response` object returned by the property.

5.3.3. Gateway emulation

The official reference for FGC [5] describes many device classes for gateways. In this case, since the previous version of the instrument made use of the `FGC_GW_SERIAL` class, that is the serial gateway, the same class was used for the emulation. The `FGC_GW_SERIAL` class is implemented in the `fgcd_gateway_serial.py` file, in the `GatewaySerial` class. It is defined as containing 66 properties, but in this case, none is implemented.

5.3.4. CDC Class

The CDC class is the main class of the application. It is responsible for creating the CDC object, that contains all the properties of the CDC. The CDC class is implemented in the `fgcd_cdc.py` file, in the `CDC` class. The CDC class contains the properties dictionary along with many device-specific methods.

State machine

The CDC operates in a very simple state machine. The states are:

- **INIT**: the CDC is in the initialization state. It is waiting for the initialization of the hardware modules.
- **IDLE**: the CDC is in the idle state. It is waiting for a command from the user.
- **NULLING**: the nulling process is ongoing, the CDC is waiting for the nulling process to complete.
- **ACTIVE**: the CDC is in a valid range, the user can set the output current.

- **ERROR**: the CDC is in an unrecoverable faulty state.

The __init__

The `__init__` method is responsible for initializing the class. It handles many tasks:

1. Instantiate three `spidev`, for the FADEM, the PRIMREL and the NULLMETER modules.
2. Instantiate the FADEM, the PRIMREL and the NULLMETER modules, passing the `spidev` objects as arguments.
3. Create the system monitor thread, passing the logger and the config object as arguments. Start the thread.
4. Create the properties dictionary, instantiating the base class for the simplest ones and the derived classes for the more complex ones.
5. Pass the properties and the logger to the constructor of the base class.
6. Fetch and set the persistent values for the one turn DAC (fullscale and zero) and set them in the PRIMREL module.

Once the `__init__` method is completed, the CDC object is ready to be used. The `hardware_init` can be called to bring all the hardware modules to a known state, and the calibrator moves to the `IDLE` state. The `hardware_init` method is called by the entry point of the application, as described in section 5.4.

5.3.5. CDC properties

In the simplest case, properties are a simple array of a given type, that are stored on the user set and returned on the user get. In the case of the CDC, many properties are more complex, as they require interaction with the hardware. The CDC properties are described in the following sections. Only a subset of the properties are simple enough to be implemented as just an instance of the base `Property` class described in subsubsection 5.1.3. The other properties are implemented as derived classes of the base `Property` class. Nonetheless, the base class defines several methods that are used by the derived classes. Only a small subset of the properties' implementations are described here. The purpose of this section is to give an overview of the variety of ways in which properties can be implemented. A thorough description of the properties' meanings is available in Appendix A, and the full implementation is available in the source code [35].

DEVICE.VERSION.MAIN_PROG

This is the simplest property to be implemented in the CDC. It is a read-only property that returns the version of the main program. The property is implemented as an instance of the `Property` class, as can be seen in Figure 5.5.

```
properties = {
  'DEVICE.VERSION.MAIN_PROG': ncrp_property.Property('DEVICE.VERSION.MAIN_PROG', 'r', int
    , description='This property indicates the current version of the software', min_val
    =0, default=0),
  # ...
}
```

Figure 5.5: DEVICE.VERSION.MAIN_PROG property

CALSYS.CDC.DAC.VOLT

This property is a read-only property that returns the voltage corresponding to the preset output of the specified DAC channel. In this case, the property is not as straightforward as the previous one. It gets the voltage from the FADEM and the PRIMREL modules, which return just the raw value. Some calculation is required to present the value in Volts. This conversion is done in two separate functions, that are then tied to the property as getters. The property is implemented as a derived class of the `Property` class, as can be seen in Figure 5.6.

```
class CalsysCdcDacVolt(fgc_property.Property):
  def __init__(self, cdc):
    self.cdc = cdc
    # create getters and setters
    getters = [self.get_ch0_volt, self.get_ch1_volt]
    super().__init__(
        name='CALSYS.CDC.DAC.VOLT',
        access='r',
        length=2,
        getters=getters,
        val_type=float,
        unit='V',
        description='This property returns the voltage corresponding to the preset output
     of the specified DAC channel.'
    )
  def get_ch0_volt(self):
    voltage = self.cdc.fadem.get_dac() * 5.0 / 65536.0 - 2.5
    return voltage
  def get_ch1_volt(self):
    voltage = self.cdc.primrel.get_dac() * 5.0 / 65536.0 - 2.5
    return voltage
```

Figure 5.6: CALSYS.CDC.DAC.VOLT property

CALSYS.CDC.OUTPUT

The `CALSYS.CDC.OUTPUT` property is a key property for the functioning of the CDC. It is a read-write property that sets the output current range of the instrument. The property is implemented as a derived class of the `Property` class, it is a complex property, as other than performing a vast amount of checks and interacting with various pieces of hardware, interacts with the `CALSYS.CDC.NULL.AUTO` property that is responsible for the nulling process. The `CALSYS.CDC.OUTPUT` property is implemented in the `fgcd_cdc_output.py` file, in the `CalsysCdcOutput` class. The `CalsysCdcOutput` class is shown in Figure 5.7. In this case, as in the previous ones, the property makes use of the getter-setter paradigm to perform complex tasks in a modular way.

```python
class CalsysCdcOutput(fgc_property.Property):

  def __init__(self, cdc):
    self.cdc = cdc
    super().__init__(
        name='CALSYS.CDC.OUTPUT',
        access='rw',
        val_type=str,
        setters=[self.set_output],
        description='This property gets and sets the range the current calibrator is in',
        default='OFF'
    )

  def _validate_value(self, value):
    # checks that the required range is not in conflict with the current state of the instrument

  def _set_value(self, value):
    # handles the range switch procedure, by preparing the hardware, calling _set_range and starting the nulling process
    # ...
    self.cdc.state = self.cdc.CDCStates.ACTIVE
    self.value[0] = value
    self.cdc.logger.debug('OUTPUT set')

  def _set_range(self, value):
    # sets the range in the hardware modules
    # clamp the fadem
    self.cdc.fadem.set_clamp1(True)
    self.cdc.fadem.set_clamp2(True)
    # wait for the fadem to settle
    time.sleep(0.1)

    self.cdc.fadem.set_range(self.ranges_map[value]['fadem'])
    self.cdc.primrel.set_range(value)
    # wait for the system to settle
    time.sleep(0.1)

  def set_output(self, value):
    # setter
    self._validate_value(value)
    self._set_value(value)
```

Figure 5.7: CALSYS.CDC.OUTPUT property

STATUS.WARNINGS

The `STATUS.WARNINGS` property is a read-only property that returns the list of warnings that are currently active. The property is implemented as a derived class of the `Property` class. It is implemented as a Python `set()`, which is a collection of unique elements. In this case, the property makes use of the getter-setter paradigm in a different way, defining a lambda function that returns the set of warnings. The `STATUS.WARNINGS` property is implemented in the `fgcd_status_warnings.py` file, in the `StatusWarnings` class. The `StatusWarnings` class is shown in Figure 5.8. The property exposes a `set_warning` and a `clear_warning` method, that are to be used by the rest of the application to set and clear the warnings.

```python
class StatusWarnings(fgc_property.Property):
  warnings = ['PBC_TMP', 'CLP_OFF', 'CLP_ON']

  def __init__(self, cdc):
    self.cdc = cdc
    super().__init__(
        name='STATUS.WARNINGS',
        access='r',
        getters=[lambda: ' '.join(self.value[0])],
        val_type=set,
        description='This property indicates the current warnings of the device'
    )

  def set_warning(self, value):
    if value not in self.warnings:
      raise ValueError('Argument must be a Warnings object')
    self.value[0].add(value)

  def clear_warning(self, value):
    if value not in self.warnings:
      raise ValueError('Argument must be a Warnings object')
    self.value[0].discard(value)
```

Figure 5.8: STATUS.WARNINGS property

5.3.6. System monitor

The system monitor is tasked to monitor:

- The voltage rails of the system;
- The presence and temperature of all the modules;
- Check for any fault or warning flags in external hardware;
- Update a cache for the HTTP server described in subsection 5.3.7.

The tasks are run every 5 seconds.

Voltage monitoring

The voltage monitoring of the four supply rails is done through the on-chip XADC. The XADC driver is loaded by the kernel, and the channels are presented as files in the filesystem, under the path: `/sys/bus/iio/devices/iio:device1/`. The channels are:

- `in_voltage8_raw`: 5V rail;
- `in_voltage10_raw`: 12V rail;
- `in_voltage9_raw`: +15V rail;
- `in_voltage11_raw`: -15V rail.

The XADC is 12-bits, so the values are in the range 0-4095. The voltages are scaled in the input voltage range by the hardware, as described section 2.11. The monitoring function reconstructs the original voltages by applying the inverse scaling factors and offsets. A calibration gain is set by setting the `CALSYS.DCM.PSxx.CAL` properties described in appendices A.23, A.25, A.27 and A.28. Those properties are set to the current value of the voltage rails measured by a calibrated voltmeter, and a calibration gain is derived and stored persistently. The calibration gain is then used to calculate the actual voltage of the rails.

The allowed voltage range is expressed as two floating point numbers, `sysmon.voltage_min_alarm` and `sysmon.voltage_max_alarm`. If the voltage is outside the range, the CDC is moved to the `ERROR` state and the event is logged.

Temperature monitoring

The temperature monitoring is done through the 1-Wire bus. Since there is no protection for concurrent use of the bus, a MutEx mechanism is put in place. With the 1-Wire driver logic being custom-made, no kernel space driver is available out of the box. To simplify development, a userspace driver is instead used. The low-level driver is provided together with the IP core.

In the temperature monitoring routine, first of all, the devices are read from the bus. If the devices are a number different than the expected one, the CDC is moved to the `ERROR` state and the event is logged. If the number of devices is correct, the system checks that the device IDs match the ones stored in a configuration file, called `modules.ini`. If the IDs do not match, the CDC attempts to recover. If just one ID is different and all the rest match, the CDC assumes that one module was replaced and it updates the configuration file. If more than one ID is different, the CDC has no way to tell which module is updated to which ID, so it logs the error. Since the failure is not unrecoverable, the CDC is not moved to the `ERROR` state.

External faults

The PRIMREL module register space also contains some flags regarding other modules or external fault and warning flags. A dictionary links every fault and warning to a function that returns a boolean value. If the function returns `True`, the fault or warning is set. If the function returns `False`, the fault or warning is cleared. When necessary, lambda functions are defined to return the boolean value.

If any fault is to be raised, the fault indicator LED is turned on. The LEDs are driven by the AXI GPIO IP described in section 3.5. The GPIOs are controlled by a kernel driver that maps the GPIO registers to the filesystem. GPIOs are presented as files under the `/sys/class/gpio/` path.

5.3.7. HTTP server

An HTTP server is implemented to allow for remote monitoring of the CDC. The server is implemented using the `http.server` Python library. The server is started by the main thread at the beginning of the program. The server is listening on the standard HTTP port (8080) and it is responsible for serving the HTTP requests from the client. The HTTP server is implemented in the `cdc_http.py` file.

The server makes use of the f-string feature of Python, to insert the proper information in a predefined HTML template.

5.3.8. Session logger

The session logger, implemented in the `SessionLogger` class, is a class that is responsible for logging each session data. The session data is logged in a file, whose name is the current date and time. The session is intended as one power cycle of the CDC.

The session logger features two independent outputs for the logs: the console and the file. Four logging levels are available: `DEBUG`, `INFO`, `WARNING` and `ERROR`. The logging level can be set independently for the console and the file.

Each level has a dedicated function that can be called to log a message. The functions are `debug()`, `info()`, `warning()` and `error()`. The functions take a string as an argument and log the message with the proper level. The functions are implemented as shown in

```python
def _log(self, level, message):
  with self.lock:
    getattr(self.logger, level)(message)

def debug(self, message):
  """Log the message at the debug level."""
  self._log('debug', message)

def info(self, message):
  """Log the message at the info level."""
  self._log('info', message)

def warning(self, message):
  """Log the message at the warning level."""
  self._log('warning', message)

def error(self, message):
  """Log the message at the error level."""
  self._log('error', message)
```

Figure 5.9: SessionLogger class methods for logging

Figure 5.9.

5.4. Entry point

The `cdc.py` file is the entry point of the application, which is responsible for fetching the configuration, retrieving system time from Network Time Protocol (NTP), starting the logger and the CDC object, starting the NCRP server and the HTTP server. It monitors the threads and restarts them if they crash.

The entry point takes as an argument the port on which the NCRP server is listening. If not provided, the default port is used.

5.4.1. Configuration system

The configuration system for the current calibrator is based on the INI, which is an easy-to-read, text-based configuration format. The configuration system is implemented by making use of the `configparser` Python library. The file is read as a dictionary and the dictionary is passed to the objects that require configurations.

5.5. Userspace drivers

The application makes use of userspace drivers for controlling the 1-Wire and the SPIs for the FADEM, the PRIMREL and the NULLMETER modules. The 1-Wire driver is bundled with the IP, and it is the work of other engineers, so it will not be discussed here.

The FADEM and the PRIMREL contain instances of the same register logic, so they can be controlled in the same way at the low level. The NULLMETER SPI is connected to a COTS ADC IC, the AD7689 [12].

5.5.1. SPI Controller driver

The PRIMREL and FADEM devices use a word width of 16 bits. Since, for compatibility reasons, the SPI driver is limited to 8 bits, the driver is implemented as a SPI driver that uses two bytes for each word. The driver is implemented in the `spi_controller.py` file, in the `SPIController` class.

This driver must provide the few basic functionalities common to the FADEM and the PRIMREL modules. It offers a `_readWord(addr)` method and a `_writeWord(addr, data)` method, that reads and writes a word to the specified address. The two 8-bit transactions are implemented in these methods. The methods exploit the `spidev.xfer2()` and `spidev.writebytes()` from the `spidev` Python library. The protocol looks like this:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$\overline{R}$W	U	6-bit addr						8-bit data							

Figure 5.10: 16-bit transfer structure

The read and write methods are shown in Figure 5.11.

```python
def _readWord(self, addr):
  addr &= 0x3F  # make sure the address is only 6 bits
  if addr % 2 != 0:
    addr -= 1
  response = self.spi.xfer2([addr, 0, addr + 1, 0])
  return (response[1] << 8) | response[3]

def _writeWord(self, addr, value):
  addr &= 0x3F  # make sure the address is only 6 bits
  if addr % 2 != 0:
    addr -= 1
  value &= 0xFFFF  # make sure the value is only 16 bits
  addr |= (1 << 7)  # set bit 7 to 1 for write
  self.spi.writebytes([addr, value >> 8, addr + 1, value & 0xFF])
```

Figure 5.11: SPI Controller driver read and write methods

The base class also implements utilities to get and set specific bits, as shown in Figure 5.12. The methods are used by the derived classes to implement the specific functionalities of the FADEM and the PRIMREL modules.

```python
def _getBit(self, addr, offset):
  if addr % 2 == 0:
    response = self._readWord(addr)
    return (response >> (offset + 8)) & 1
  else:
    response = self._readWord(addr - 1)
    return (response >> offset) & 1

def _setBit(self, addr, offset, value=1):
  if addr % 2 == 0:
    response = self._readWord(addr)
    if value:
      response |= (1 << (offset + 8))
    else:
      response &= ~(1 << (offset + 8))
    self._writeWord(addr, response)
  else:
    response = self._readWord(addr - 1)
    if value:
      response |= (1 << offset)
    else:
      response &= ~(1 << offset)
    self._writeWord(addr - 1, response)
```

Figure 5.12: SPI Controller driver get and set bit methods

5.5.2. FADEM driver

The FADEM class is derived from the `SPIController` class. It implements the `FADEM` class, which is responsible for the low-level communication with the FADEM module. The `FADEM` class is implemented in the `fadem.py` file. The `FADEM` class implements high-level methods for accessing the various registers, that are documented in Table 5.1.

The class features a `reset()` method that sets all the register to a known state, a `set_range()` and a `get_range()` methods to set the range gain in the FADEM module, a `set_dac()` and a `get_dac()` methods to set the current in the 1T-DAC winding, a `set_clamp1()` and a `set_clamp2()` methods to set the clamps, together with several methods that just write or read a 16 bit value to and from the appropriate register.

State	Value
FADEM_RAMPDOWN	0b100
FADEM_NORMAL	0b011
FADEM_DEGAUSS	0b010
FADEM_IDLE	0b001
FADEM_INIT	0b000

Figure 5.13: FADEM states and values

The `FADEM` class has also a definition for the state machine states, that are used by the

CDC to set the state of the FADEM module. The states are shown in Figure 5.13. A `get_state()` and a `set_state()` methods are implemented to get and set the state of the FADEM module.

5.5.3. PRIMREL driver

The PRIMREL class is derived from the `SPIController` class. It implements the `PRIMREL` class, which is responsible for the low-level communication with the PRIMREL module. The `PrimRel` class is implemented in the `primrel.py` file. The `PrimRel` class implements high-level methods for accessing the various registers, that are documented in Table 5.2.

The PRIMREL module is tasked with controlling the primary relays (setting the output current), the secondary relays (setting the range) and many IOs for external hardware, that are mapped to the registers in the SPI peripheral.

The class features a `reset()` method that sets all the register to a known state, a `set_range()` and a `get_range()` methods to set the secondary relays, a `set_dac()` and a `get_dac()` methods are used to control the DAC that nulls the offset in the control loop.

Table 5.1: Detailed mapping of the FADEM FPGA registers.

A	RW	Name	D7	D6	D5	D4	D3	D2	D1	D0
0	R	STATUS MSB	ACTIVE	0	1	0	1	0	1	0
1	R	STATUS LSB	CLAMP2	CLAMP1	RANGE[1]	RANGE[0]	RANGE_EN	STATE[2]	STATE[1]	STATE[0]
2	R/W	CMD MSB	ACTIVATE	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd
3	R/W	CMD LSB	CLAMP2	CLAMP1	RANGE[1]	RANGE[0]	RANGE_EN	rsvd	rsvd	rsvd
4	R/W	NORMAL GAIN MSB	NG15	NG14	NG13	NG12	NG11	NG10	NG9	NG8
5	R/W	NORMAL GAIN LSB	NG7	NG6	NG5	NG4	NG3	NG2	NG1	NG0
6	R/W	DEGAUSS GAIN MSB	DG15	DG14	DG13	DG12	DG11	DG10	DG9	DG8
7	R/W	DEGAUSS GAIN LSB	DG7	DG6	DG5	DG4	DG3	DG2	DG1	DG0
8	R/W	DEM1 SHIFT MSB	D1S15	D1S14	D1S13	D1S12	D1S11	D1S10	D1S9	D1S8
9	R/W	DEM1 SHIFT LSB	D1S7	D1S6	D1S5	D1S4	D1S3	D1S2	D1S1	D1S0
10	R/W	DEM2 SHIFT MSB	D2S15	D2S14	D2S13	D2S12	D2S11	D2S10	D2S9	D2S8
11	R/W	DEM2 SHIFT LSB	D2S7	D2S6	D2S5	D2S4	D2S3	D2S2	D1S1	D1S0
12	R/W	DAC CH1 OUT MSB	DAC1O15	DAC1O14	DAC1O13	DAC1O12	DAC1O11	DAC1O10	DAC1O9	DAC1O8
13	R/W	DAC CH1 OUT LSB	DAC1O7	DAC1O6	DAC1O5	DAC1O4	DAC1O3	DAC1O2	DAC1O1	DAC1O0
14	R/W	DAC CH2 OUT MSB	DAC2O15	DAC2O14	DAC2O13	DAC2O12	DAC2O11	DAC2O10	DAC2O9	DAC2O8
15	R/W	DAC CH2 OUT LSB	DAC2O7	DAC2O6	DAC2O5	DAC2O4	DAC2O3	DAC2O2	DAC2O1	DAC2O0

Table 5.2: Detailed mapping of the PRIMREL FPGA registers.

A	R/W	Name	D7	D6	D5	D4	D3	D2	D1	D0
0	R	STATUS MSB	rsvd	rsvd	rsvd	rsvd	rsvd	ACTIVE	0	1
1	R	STATUS LSB	0	1	0	1	0	STATE[2]	STATE[1]	STATE[0]
2	R/W	CMD MSB	ACTIVATE	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd
3	R/W	CMD LSB	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd
4	R/W	REL POL CMD MSB	x	x	2048T	1024T	512T	256T	128T	64T
5	R/W	REL POL CMD LSB	32T	16T	8T	4T	2T	1T	1TCAL	1TDAC
6	R/W	NOT USED								
7	R/W	NOT USED								
8	R/W	REL ON/OFF CMD MSB	x	x	2048T	1024T	512T	256T	128T	64T
9	R/W	REL ON/OFF CMD LSB	32T	16T	8T	4T	2T	1T	1TCAL	1TDAC
10	R/W	EXT OUT CMD MSB	x	x	x	x	x	x	x	x
11	R/W	EXT OUT CMD LSB	EXTOUT8	EXTOUT7	EXTOUT6	EXTOUT5	EXTOUT4	EXTOUT3	EXTOUT2	EXTOUT1
12	R	REL POL STAT MSB	0	0	2048T	1024T	512T	256T	128T	64T
13	R	REL POL STAT LSB	32T	16T	8T	4T	2T	1T	1TCAL	1TDAC
14	R	REL ON/OFF STAT MSB	0	0	2048T	1024T	512T	256T	128T	64T
15	R	REL ON/OFF STAT LSB	32T	16T	8T	4T	2T	1T	1TCAL	1TDAC
16	R	EXT IN STAT MSB	EXTIN16	EXTIN15	EXTIN14	EXTIN13	EXTIN12	EXTIN11	EXTIN10	EXTIN9
17	R	EXT IN STAT LSB	EXTIN8	EXTIN7	EXTIN6	EXTIN5	EXTIN4	EXTIN3	EXTIN2	EXTIN1
18	R	NOT USED								
19	R	NOT USED								
20	R/W	1T DAC CH1 OUT MSB	DAC1O15	DAC1O14	DAC1O13	DAC1O12	DAC1O11	DAC1O10	DAC1O9	DAC1O8
21	R/W	1T DAC CH1 OUT LSB	DAC1O7	DAC1O6	DAC1O5	DAC1O4	DAC1O3	DAC1O2	DAC1O1	DAC1O0
22	R/W	1T DAC CH2 OUT MSB	DAC2O15	DAC2O14	DAC2O13	DAC2O12	DAC2O11	DAC2O10	DAC2O9	DAC2O8
23	R/W	1T DAC CH2 OUT LSB	DAC2O7	DAC2O6	DAC2O5	DAC2O4	DAC2O3	DAC2O2	DAC2O1	DAC2O0

6 | Application testing

Testing and automation are two tools instrumental to the maintainability objective of this work, as they allow us to ensure that the system is working as expected and that the system is not broken by future changes. This chapter describes the testing and automation tools that have been put in place to ensure the maintainability of the system.

6.1. Continuous integration

Continuous Integration (CI) is a paradigm that is used to ensure that the code is always in a working state. The idea is that every time a change is made to the code, the code is tested and verified to be working. This is done by running the tests on a server, which is called the CI server. The CI server is configured to run the tests every time a change is made to the code. If the tests fail, the CI server notifies the developers that the code is broken. This allows the developers to fix the code before going forward with the development.

The automation infrastructure is made available from GitLab, which is a web-based version control system, in an instance run by CERN. CERN's GitLab provides a CI server that is integrated with the version control. The CI server is configured to run the tests every time a change is made to the code. The tests run in a clean environment every time, to ensure isolation between tests. The code coverage is also measured at every run. The code coverage is a measurement of the lines of code that are executed by the tests. While a code coverage of 100% does not ensure that the code is exempt from bugs, it is a good indication that there are no pieces of code that are left untested. In practice, it is hard or impossible to reach a code coverage of 100%, so a code coverage of 85-95% is considered a good target. The code coverage is measured with the pytest-cov plugin.

6.1.1. Pytest

Pytest [10] is a testing framework for Python. It allows the user to write tests simply and intuitively. The tests are written in Python, and they can be run from the command line.

Pytest is a very popular testing framework, and it is used in many projects. The tests are written in a way that is very similar to the way the code is written, which makes it easy to write and maintain the tests. For this application, unit tests are written. Unit tests are tests that test the smallest unit of code, which is usually a function or a class. The tests are written to test the functionality of the code, and to ensure that the code is working as expected, as well as testing the handling of errors.

6.1.2. Documentation generation and deployment

The documentation is generated with Sphinx [11]. Sphinx is a documentation generator that is used to generate documentation from reStructuredText files. The documentation is generated from the docstrings in the code. The docstrings are strings that are written in the code to document it. They are written in a way that is very similar to the way the code is written, which makes it easy to write and maintain the documentation. Sphinx generates documentation in an HTML format, and the CI/CD deploys it to a web server. The documentation is generated automatically every time a change is made to the code, and it is deployed automatically to the web server. This ensures that the documentation is always up to date, and readily available to the developers.

6.2. Continuous Deployment

Continuous Deployment (CD) is a paradigm that is used to ensure that the code is always available and ready to be deployed on the field. After the CI testing is run, the system image is built with Petalinux and packaged in a format that makes it practical to deploy.

6.2.1. Building the Petalinux project

Petalinux was not built with compatibility with a versioning system in mind, as it is not possible even to change the location of a project without breaking it. Paths are absolute and hardcoded, so the only way to move a project to a new location, even on the same machine, is to package it and unpackage the project in the new desired location. The packaged project is particularly ill-suited for versioning, as it is a single file that is not intelligible.

The proposed solution is to create a new project every time the build is run, in a clean environment, and to copy the user-modified files over the new project to implement the customizations on the final build. The script that does this is presented in Listing 6.1.

The build produces three files that should be copied onto the SD card to be used in

Listing 6.1: Script for building the petalinux project on the CI machine

```
BUILD_NAME=build
PROJECT_NAME=cdc-petalinux
PETALINUX_USER="petalinux"
CI_PETALINUX_PASSWORD="${PETALINUX\_PASSWORD:-zynqdev}"

# create a "build" project
petalinux-create -t project -n $BUILD\_NAME --template zynq

# run the hw config
petalinux-config --silentconfig --get-hw-description=design\_1\_wrapper.xsa -p $BUILD\
  _NAME

# run the config
petalinux-config --silentconfig --project $BUILD\_NAME/

# set the user name and password from the secrets
CONFIG\_FILE="$PROJECT\_NAME/project-spec/configs/rootfs\_config"
sed -i "s/CONFIG\_ADD\_EXTRA\_USERS=\"petalinux:zynqdev;\"/CONFIG\_ADD\_EXTRA\_USERS=\"
  $PETALINUX\_USER:$CI\_PETALINUX\_PASSWORD;\"/g" $CONFIG\_FILE

# copy over previos config
rsync -av --exclude='hw-description' $PROJECT\_NAME/project-spec $BUILD\_NAME/

# build
petalinux-build -p $BUILD\_NAME && \
# package
petalinux-package --boot --force --fsbl $BUILD\_NAME/images/linux/zynq\_fsbl.elf --fpga
   $BUILD\_NAME/project-spec/hw-description/design\_1\_wrapper.bit  --u-boot --project
  $BUILD\_NAME && \
# do bsp
petalinux-package --bsp -p $BUILD\_NAME -o cdc-petalinux.bsp
```

the FPGA board. The files are the boot script `boot.scr`, the binary of the bootloader `BOOT.BIN` and the system and kernel image `image.ub`. These files are marked as artifacts of the build so that they are stored and passed to the next step, described in subsection 6.2.3.

6.2.2. Protecting the passwords in the CD

In a Petalinux project, the users and their passwords are defined in a source file called `rootfs_config`. This file is part of the source files and as such it is committed to the repository and it is public. This means that the passwords of the users can be seen by anyone, which raises great security concerns.

Instead, the password can be stored as a secret variable in the CI environment, and the file can be overwritten by the CI script. In this way, the password is never exposed to the public, and it is only known to the CI server and to the staff with access to the repository.

6.2.3. Making the SD image

Once the build is complete, the last step is to make the SD image. The SD image is an image file that already contains a partition with the build products generated by subsection 6.2.1, and another partition with the application python files. The SD image is made with a script that is run on the CI machine. The script is presented in Listing 6.2.

The `sdcard.img` file can then be copied bit by bit to an SD card, which can then be used to boot the FPGA board.

Listing 6.2: Script for building the image for the SD

```bash
# 1. Set up the environment for creating SD card images
apt-get update && apt-get install -y parted losetup kpartx git

# 2. Create an empty SD card image (let's say 2GB)
dd if=/dev/zero of=sdcard.img bs=1M count=2000

# Create the partitions on the SD card image
parted sdcard.img --script mklabel msdos
parted sdcard.img --script mkpart primary fat32 1M 500M
parted sdcard.img --script mkpart primary fat32 500M 100%

# check the result
fdisk -l sdcard.img

# create loop device
sudo mknod -m660 /dev/loop10 b 7 10
sudo chown root:disk /dev/loop10

# Generate a MAC address and store it in an environment variable
export TARGET_MAC_ADDRESS="00:18:3e:04:71:c7"

# Create the uEnv.txt file with the generated MAC address
echo "ethaddr=${TARGET_MAC_ADDRESS}" > uEnv.txt

# Report the chosen MAC address to the job's output
echo "Chosen MAC address ${TARGET_MAC_ADDRESS}"

# Setup loop devices for the partitions
sudo losetup -P /dev/loop10 sdcard.img
LOOP_DEV="/dev/loop10"
sudo kpartx -avs $LOOP_DEV
sudo ls /dev/mapper/
sudo partprobe $LOOP_DEV
ls /dev/loop*

# 3. Mount each partition and populate them with the necessary files
# For the 1st partition
sudo mkdir -p /mnt/part1
sudo mkfs.vfat /dev/mapper/loop10p1
sudo fatlabel /dev/mapper/loop10p1 BOOT
sudo mount /dev/mapper/loop10p1 /mnt/part1
sudo cp $BUILD_NAME/images/linux/* /mnt/part1/
sudo cp uEnv.txt /mnt/part1/
sudo umount /mnt/part1

# For the 2nd partition
sudo mkdir -p /mnt/part2
sudo mkfs.vfat /dev/mapper/loop10p2
sudo fatlabel /dev/mapper/loop10p2 APP
sudo mount /dev/mapper/loop10p2 /mnt/part2
sudo cp -r cdc-control-application /mnt/part2
sudo umount /mnt/part2

# 4. Unmount the partitions and detach loop device
sudo kpartx -dvs $LOOP_DEV
sudo losetup -d $LOOP_DEV
```

7 | Performance testing

The last part of this work focuses on the validation of the system from a metrological perspective. Tests are conducted to assess the performance of the system with a particular focus on the stability over 12-48h.

Measuring the performance of such a high-precision system poses significant challenges, as it requires the use of high-end instruments and advanced techniques. A direct measurement of the current would be vastly inadequate. Considering for example the direct measurement of the current with a metrological digitizing multimeter such as the Fluke 8588A in the 1 A range, the uncertainty of the measurement in the ideal case would be no less than ±27.5 ppm over 20 minutes (see [22]). For reference, the specification for the same instrument on the DC 10 V range is ±0.1 ppm over 20 minutes, increased to ±0.55 ppm over 24 hours. Similar specifications hold for the HP 3458A, one of the leading metrology digitizing multimeters [27].

7.1. Performance assessment

The insufficient performance of the COTS Digital Voltmeters (DVMs) is mostly due to the internal working, where the current that is to be measured is run in a shunt resistor and the voltage drop across the resistor is measured. The limiting factor in terms of performance is the stability and the self-heating of the internal shunt resistor. A resistor that is physically large enough to dissipate the heat generated by the current flowing through it would be too large to be integrated into a DVM. Instead, to perform the measurements required by this work, a DVM is used to measure the voltage drop across a large and temperature-controlled precision resistor, which is then used to calculate the current. The precision resistor is an Alpha Electronics FNP 0.200 Ω shunt resistor [17].

The resistor is specified by the manufacturer to have a temperature coefficient within 0±1 ppm/°C and a resistance tolerance of ±0.05%. The resistor is mounted on a temperature-controlled aluminum heatsink, on which a fan ensures constant flow of air. The assembly is then put in a temperature-controlled chamber, where the temperature is set to 30.0 °C.

The temperature stability of the setup allows the temperature coefficient of the resistor to not become significant in the measurement. The voltage drop across the resistor is fed in a x10 gain stable amplifier, that is placed in the same temperature-controlled chamber. The amplifier is a custom design based on the OPA189 precision operational amplifier [26]. This methodology is mutuated from [30].

At the output of the amplifier, for a current of 5 A, a voltage of 10 V is to be measured. This voltage is chosen as it is the voltage range at which the DVMs have the best specified performance. The voltage is measured with two Keysight 3458A DVM [27], which is specified to have a 24-hour accuracy of ±0.55 ppm. Since we want to measure a stability of 0.1 ppm over 24 hours, the DVMs are the limiting factor in the measurement. Nonetheless, this measurement sets the first benchmark for the performance of the system.

7.1.1. Current reversal test

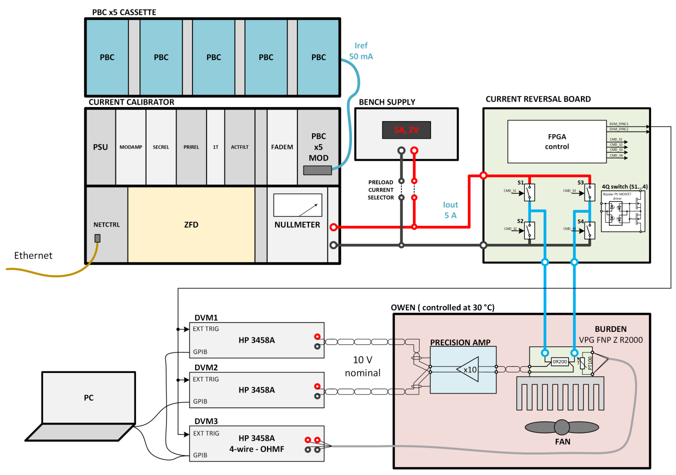


Figure 7.1: Current reversal system, images adapted from [30]

At the levels of precision required by this measurement, the thermoelectric effect of bimetallic junctions becomes significant [28]. Common materials such as nickel and copper, when coupled, can result in thermoelectric voltage coefficients as high as $10^{\mu V}/_{^\circ C}$ per junction. If the junctions or the materials are not perfectly symmetric, or if there is an asymmetric temperature gradient, the thermoelectric voltage adds a variable offset to the measurement that is impossible to extract, which can be significant at the levels of precision required by this work. To mitigate this effect, a common technique is to reverse the measured voltage. The thermoelectric voltage is then reversed, and the difference between the two measurements is taken. The thermoelectric voltage is canceled out, and the measurement is free from the effect of the thermoelectric voltage. This technique is used in this work to measure the stability of the system.

To perform the reversal, a special mosfet bridge board was designed. The bridge inverts the polarity of the current fed into the shunt resistor. The bridge is controlled by a separate FPGA and does the reversing every second. The bridge board outputs a sync signal that is used to synchronize the measurements of the DVMs. The overall system is shown in figure 7.1.

The current reversal board provides a trigger signal to the DVMs to synchronize the measurements. The timing of the current reversal is as follows: 120 ms of short (dead time), 1080 ms of current in verse A, 120 ms of short, and 1080 ms of current in verse B. The voltage is sampled from 240 ms after the short to 240 ms before the next short, for a 600 ms window. The current reversal behavior is shown in Figure 7.2.

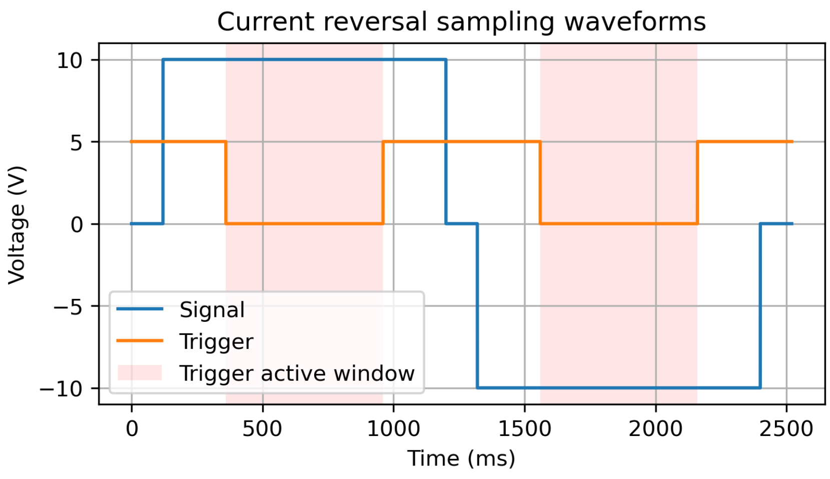


Figure 7.2: Current reversal timing

The test ran over 37 hours and the results are shown in figure 7.3. The results show a worst-case stability of 0.53 ppm over 37 hours. With the DVMs being the limiting factor, this is a good result. The results show that the system is stable enough to outperform the DVMs used in the measurement. A measurement is then run to assess the baseline of the DVMs. The results are shown in section 7.1.2.

7.1.2. HP 3458A Baseline test

To assess the baseline of the DVMs used in the measurement, a test is run with the DVMs directly connected to a voltage reference. To minimize the effects of voltage variations induced by thermal effects, the reference is placed in a temperature-controlled chamber. The reference used is a PBC v3 [34]. The reference is expected to have a one-year stability of 1.5 ppm, for a 24-hour period it is expected to be irrelevant for the purpose of this test. The configuration is shown in figure 7.4.

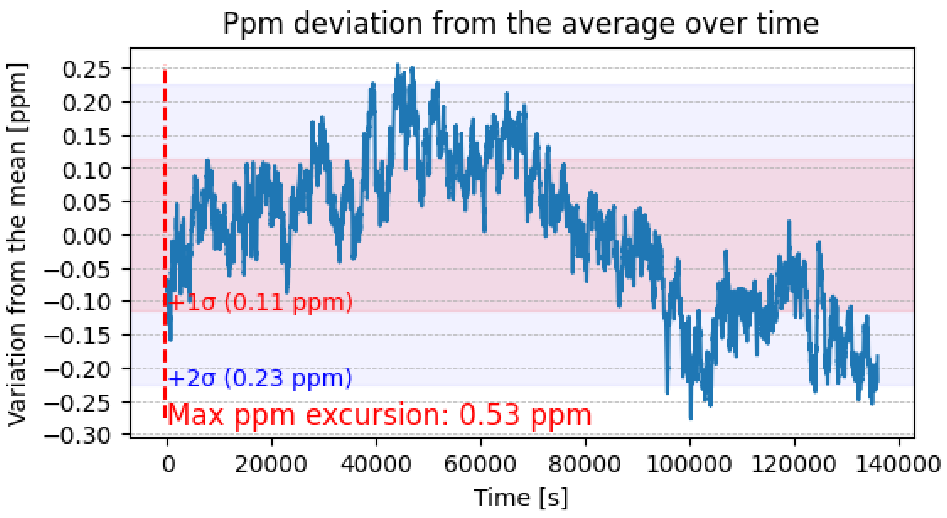


Figure 7.3: Test results for the current reversal test

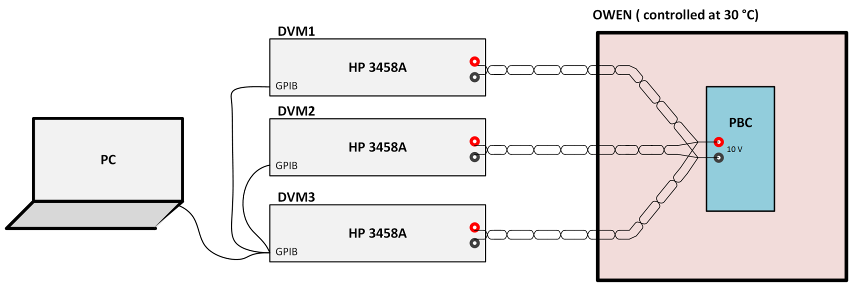


Figure 7.4: HP 3458A baseline test system

The test is run over 24 hours and the results are shown in Figure 7.5. The results show a spread of worst-case stabilities of over 1 ppm for the three DVMs used in the measurement. This is in line with the specifications of the DVMs, which specify a 24-hour accuracy of ±0.55 ppm. This finally shows that the DVMs are the limiting factor in the measurement. It can also be observed from Figure 7.5 how one of the DVMs presents more noise than the other two. This may indicate that the instrument is faulty or, in general, worse than the other two.

7.1.3. Current reversal with reference test

To assess the performance of the system with the DVMs being the limiting factor, it is proposed to use a voltage reference to be measured in between every measurement. To do the switching, a metrology multiplexer could be used to switch between the reference and the shunt resistor. Since such an instrument is not available, and since the functionality is readily available in the Fluke 8588A [22], it is decided to use the Fluke 8588A to switch

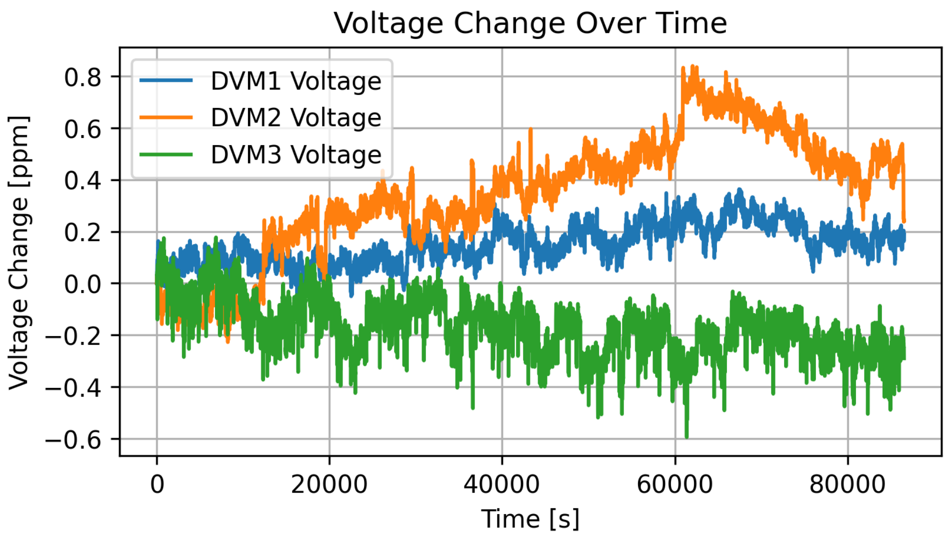


Figure 7.5: Test results for the HP 3458A baseline test

between the reference and the shunt resistor. Moreover, the Fluke 8588A outperforms the low-frequency characteristic of the HP 3458A [31]. The configuration is shown in figure 7.6. Since the Fluke 8588A acquires one measurement per trigger, switching between front and back terminals at subsequent triggers, the current reversal board is modified to hold the same polarity for two cycles, while still producing two separate trigger pulses.

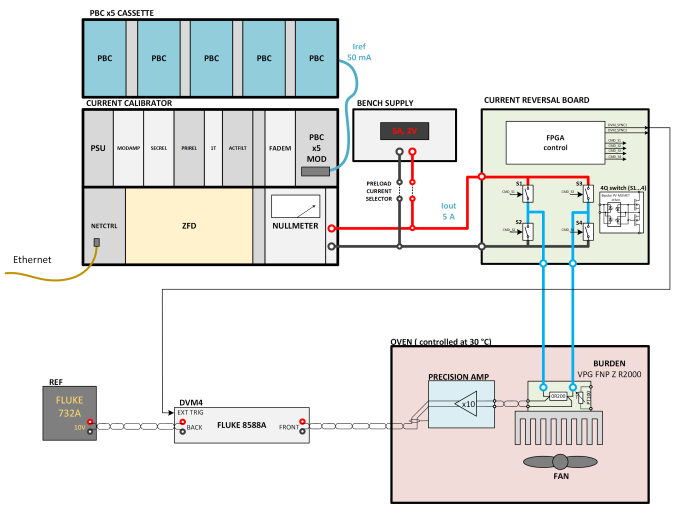


Figure 7.6: Current reversal with reference test system

The reference in use is a Fluke 732B, which is one of four units that make up the main voltage standard for the metrology laboratory in SY-EPC-HPM. The unit is known to be the most stable available, and it has been calibrated traceably in the Swiss Federal Institute of Metrology (METAS) for several years in a row. The specifications for the

Fluke 732B foresee a 0.3 ppm change per 30 days [20], but historical data shows that the unit exhibits a stability of 0.1 ppm over one year, the drift is therefore neglected for this test. The manufacturer of the equipment confirms that the stable drift is a characteristic of the design of the voltage reference [21], so the same drift that always characterized the unit can be expected for this time window.

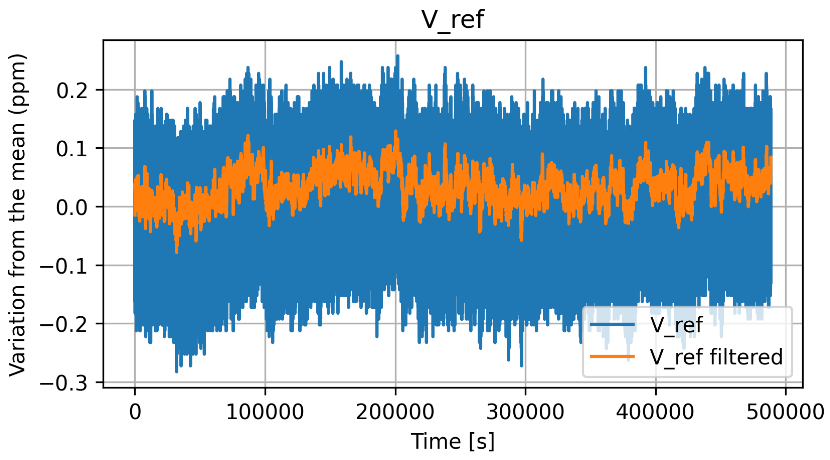


Figure 7.7: Reference measured in the current reversal with reference setup

The results shown in Figure 7.7 show a significant noise on the measurement, which indicates that some noise rejection is required for the measurement to be significant. The specifications for the Fluke 8588A in 24 hours indicate a 0.55 ppm accuracy [22] (at 2σ). The results show a spread of 0.18 ppm over 24 hours (2σ), which is largely within the specifications. If a moving average is applied to the results, on a window of 100 samples, the spread is reduced to 0.058 ppm at 2σ. Considering the average, the peak-to-peak variation is within the required 0.2 ppm. This is enough to provide a baseline for the measurement. The voltage reference is used to establish a calibration for the measuring system at each sample. Considering the voltage reference as the absolute reference for this measurement, every reading of the DVMs can be divided by the reading of the reference. This way, the measurement at the reference is used to calibrate the measurement at the shunt resistor. The specification sheet for the Fluke 8588A quote, for this scenario, has a worst-case 2σ tolerance of 0.10 ppm.

The measurement is presented in Figure 7.8. An initial settling can be observed in the first hours. The reason for this is examined in subsubsection 7.1.3. The measurement is then stable within 0.2 ppm for the rest of the record. This run alone, although not conclusive, is enough to indicate that the stability of the system is on the order of the stability of the reference used plus the stability of the DVM. This is a good result, but a better measurement can be obtained.

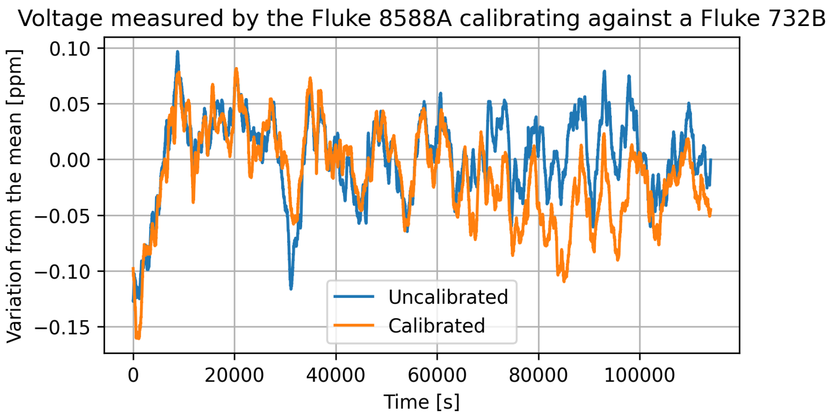


Figure 7.8: Measurement of the current in the setup, calibrated by the reference

Temperature coefficient of the shunt resistor

Every resistor is characterized by a certain temperature coefficient, which indicates how much the resistance of the resistor changes with temperature. The temperature coefficient of the resistor used in this work is specified by the manufacturer to be 0 ± 1 ppm/°C. This means that, for a temperature change of 1 °C, the resistance of the resistor changes by no more than 1 ppm. The setup put in place for the measurement of the current can be used to measure and confirm the temperature coefficient of the resistor. With the same setup as described before, but with the amplifier removed, the temperature in the oven is stepped from 30.0 °C to 35.0 °C. The temperature is then kept constant for some hours. The results for this unit are shown in Figure 7.9. The results show a change of 2.8 ppm/°C, which is considerably worse than what the manufacturer quotes. The measurement also highlights a slow settling process after the temperature step and some hysteresis in the value with temperature. This result, combined with the data on the temperature of the resistor, yields more information on uncertainty of the measurement. The temperature of the resistor is measured with a PT100 sensor embedded in the FNP resistor package, and read by an HP 3458A. A 4-wire ohm measurement is set up for this. The HP 3458A is specified to have a 24-hour accuracy of ±2.2 ppm [27] for a 4-wire 1000 Ω measurement. This is not a concern, as the temperature of the resistor has effect on the measurement on the order of some ppm/°C, so the effect of this uncertainty is negligible.

During the test, the temperature of the resistor is measured, and with the calculated Tc, the impact of the temperature change can be estimated. Figure 7.10 estimates, from the temperature read at the burden resistor, the variation introduced by the temperature change. Without considering the initial settling, the change caused by the resistor Tc is quoted around 0.0355 ppm (2σ) which means 0.11 ppm peak-to-peak in a 99% confidence

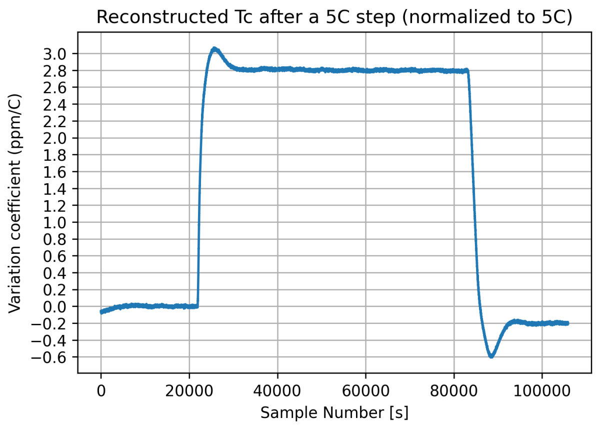


Figure 7.9: Temperature coefficient measurement of the shunt resistor

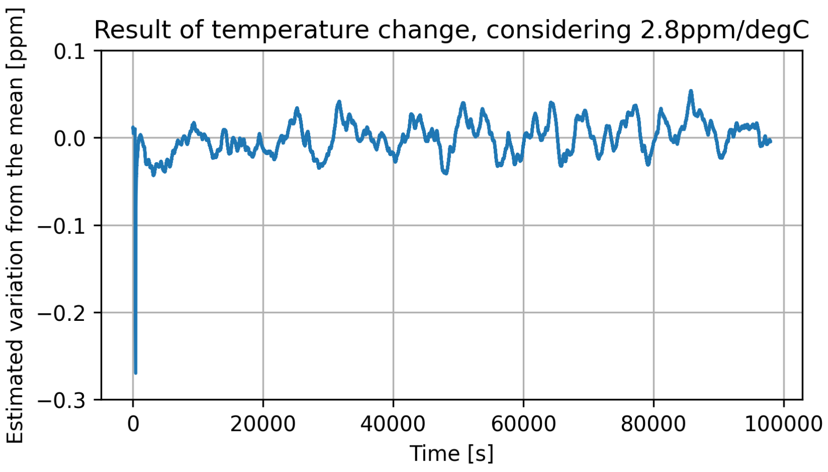


Figure 7.10: Estimated effect of the temperature change on the measurement

interval. We can therefore state that this is the limit of any measurement that uses this resistor. It is also evident how the initial settling is correlated with the temperature change. This is due to the fact that, while the resistor is "preheated" by a 5 A current source, that current source is not as stable nor as accurate as the CDC, and the current change causes a temperature change that results in a resistance change, which in turn causes a change in the voltage drop across the resistor. Moreover, the switch is performed manually and during the switch time no current is put through the resistor, causing cooling. This effect is not negligible, as it is in the order of 0.1 ppm.

What is shown in Figure 7.10 can explain the settling and some of the variations observed in Figure 7.8. This limit can only be overcome by using a resistor with a better temperature coefficient, that can be selected between the available ones. The temperature coefficient of the resistor is therefore finally the limiting factor in the measurement.

Front vs back terminals

The specification for the difference and variation of the front and back terminals of the Fluke 8588A is not quoted in the datasheet. Using the Fluke 732B we can measure a known reference from front and back terminals and compare the results. The results are shown in Figure 7.11. This shows that the difference between the two terminals has a fixed offset of 0.08 ppm and varies in the range of 0.04 ppm peak-to-peak over 24 hours. This is not a concern, as the difference is negligible compared to the stability of the DVMs.

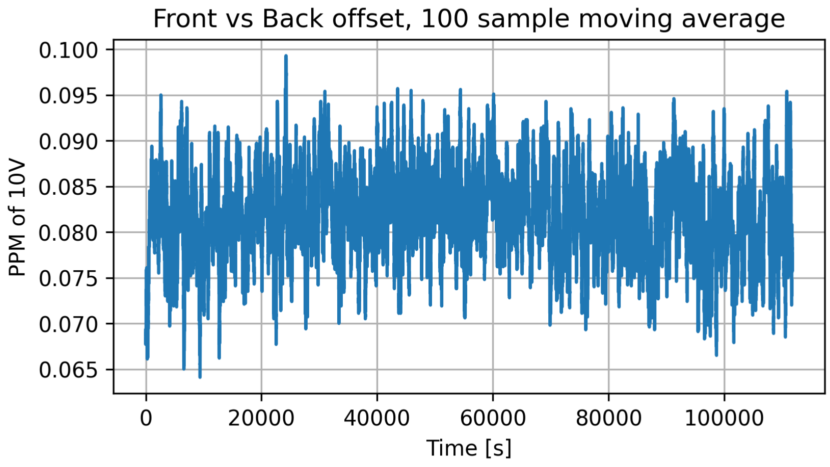


Figure 7.11: Front vs back terminals offset

7.1.4. HPM7177 test

A last test was carried out using the HPM7177 [2]. The HPM7177 is a high-precision digitizer, designed by SY-EPC-HPM. It features an ADR1000 as a voltage reference, which is a very stable reference buried zener diode [14]. The HPM7177 is designed to be used as a high-precision digitizer for the CERN power converters, in the most demanding accuracy class for Hi-Lumi. Its 12-hour was shown to be better than 0.2 ppm by [2]. The HPM7177 is used to measure the voltage drop across the shunt resistor, amplified by the low noise amplifier, both placed in the temperature-controlled chamber.

Six units are used in the measurement, as shown in Figure 7.12. The six units sample at a fixed rate of 10kSample/s, so the data must be decimated for storage. A first averaging brings the data-rate to 100 Samples/s, which are stored for later analysis. The data are read from the digitizers via optical fiber to a National Instruments PXI system, which synchronizes the digitizers and sends the data to a computer for storage.

Five of the channels are used for the measurement, while the sixth channel is used to measure the trigger signal coming from the current reversal board. The reversal and

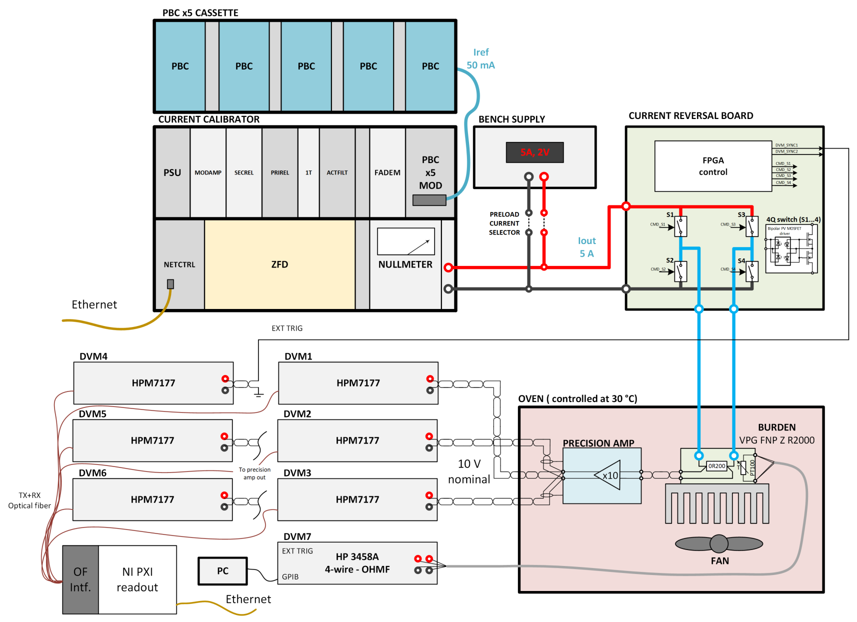


Figure 7.12: HPM7177 test setup

trigger pattern remains the same as described in subsection 7.1.3, with a single sample generated by averaging all the samples in two subsequent windows. An example of the samples, trigger and triggering windows is shown in Figure 7.13.

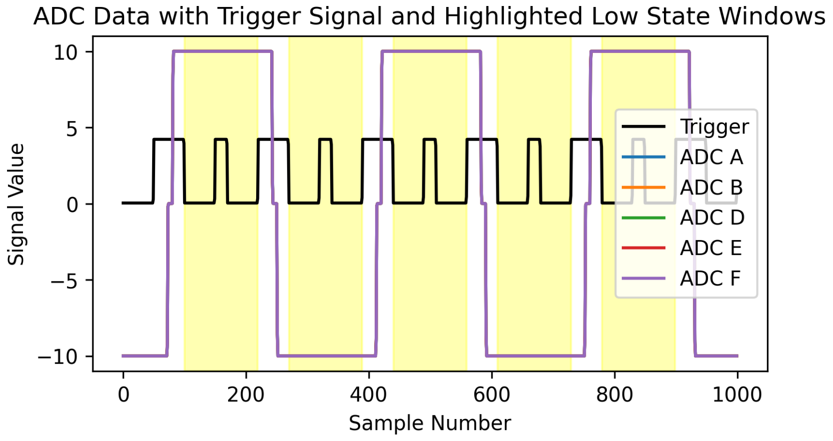


Figure 7.13: HPM7177 samples, trigger and triggering windows

The five digitizer data streams are then averaged to be combined in a single signal, where the noise contribution of the digitizers is reduced by a factor of $\sqrt{5}$. The digitizers are calibrated separately and are expected to maintain calibration for the duration of the test. The standard deviation between the five digitizers stays contained within 0.15 ppm, as shown in Figure 7.14, which shows that the digitizers agree with each other to a sufficient degree.

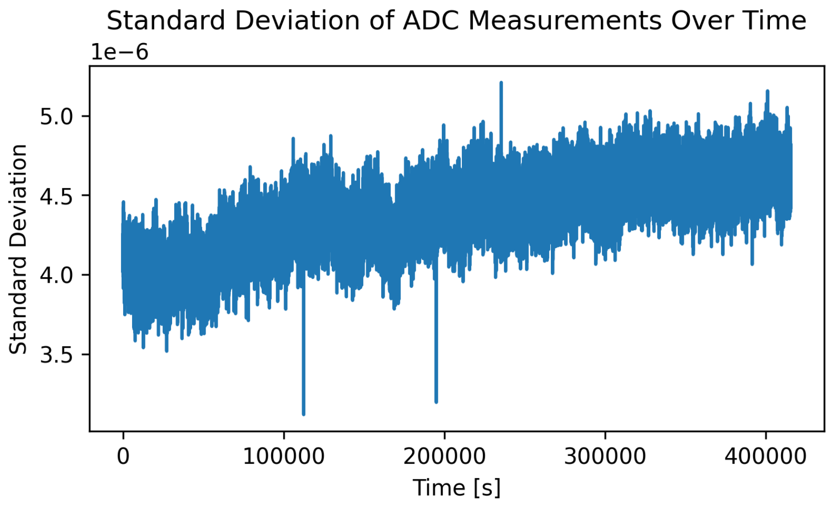


Figure 7.14: HPM7177 standard deviation between the five digitizers

The test has been run in parallel with the one described in subsection 7.1.1, for the same amount of time. The results are presented in Figure 7.15. The result is quoted as having a σ of 0.048 ppm for the whole run and 0.028 ppm after the initial settling, with a total peak-to-peak variation of 0.14 ppm. This remarkable result shows that the system complies with the 12-hour stability specification, which was one of the parameters that were improved the most during the upgrade of the CDC. It is worth noting that the shunt resistor variation alone due to thermal variations is estimated to be around 0.12 ppm in subsubsection 7.1.3, showing once again that the resistor is the limiting factor in the measurement.

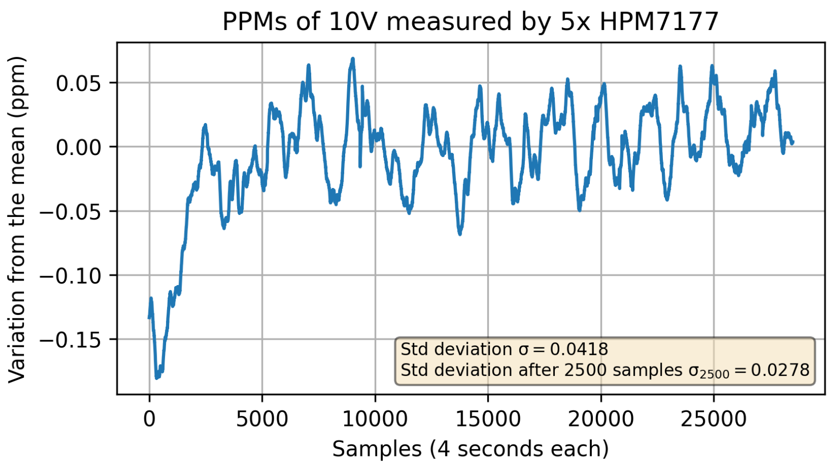


Figure 7.15: HPM7177 test results

8 | Conclusions and Future Developments

In this thesis, we have presented a comprehensive analysis of the design and implementation of the controller for the CERN DCCT Calibrator (CDC). The CDC is a high-precision instrument that will be used to calibrate the current transducers that will be used to measure the current in the power converters. The new controller of the instrument will be key to ensuring the correct calibration of the power converters that will power the upgraded magnets in the HiLumi project.

In the following years, the maintenance of the control system will be minimal, and the system is already designed to accept new features and improvements. The system is also designed to be easily replicated and adapted to other applications compatible with the FGC NCRP protocol. Moreover, at the end of the life of the Zynq-7000 system, the system can be easily ported to a new platform such as the Zynq UltraScale+, as the software is designed to be platform independent and only little changes will be required to the Petalinux configuration.

The CI/CD paradigm will ensure that every change that will be introduced in the future will not only be tested but also documented and reproducible. The CI/CD pipeline will also ensure that the system will be continuously built and that the documentation will be updated accordingly. The continuous build system will be left in place so that it can be kept working with no need to reconstruct it from scratch when changes will be required.

The functionality and performance of the system have been thoroughly tested in lab conditions, and the tests show unprecedented stability from the final system. For the first time after the full upgrade, including the controller, the stability of the output current over 12 hours is documented to be within 0.15 ppm after the initial settling.

Moving forward, several areas could build upon the work presented in this thesis for future research and development. For example, the limiting factor in the measurement turned out to be the temperature coefficient of the resistor. A better measurement of the current stability could be achieved by using a resistor with a lower temperature coefficient. The temperature coefficient of the resistor could also be measured compensated for in software. On the controller side, a USB port on the front could be added and made compatible with the FGC Serial Command Response Protocol (SCRP) for compatibility in scenarios where there might not be access to a network.

A graphical user interface could also be developed to allow the user to interact with the system without the need for a terminal. The user interface could be developed in a web-based framework such as React or Vue.js and could be served by the embedded web server. This would allow for a more practical and immediate control of the instrument.

Bibliography

[1] H. C. Appelo, M. Groenenboom, and J. Lisser. The zero-flux dc current transformer a high precision bipolar wide-band measuring device. *IEEE Transactions on Nuclear Science*, 24(3):1810–1811, 1977. doi: 10.1109/TNS.1977.4329095.

[2] N. Beev, M. Cerqueira Bastos, M. Martino, D. Valuch, L. Palafox, and R. Behr. Design and metrological characterization of a digitizer for the highest precision magnet powering in the high luminosity large hadron collider. *IEEE Transactions on Instrumentation and Measurement (in process)*, Due 2023. Accessed: 2023.

[3] D. Calcoen, Q. King, and P. Semanaz. Evolution of the CERN Power Converter Function Generator/Controller for Operation in Fast Cycling Accelerators. *ICALEPCS2011 Conference Proceedings*, C111010:WEPMN026, 2011. URL `https://cds.cern.ch/record/1563842`.

[4] M. Cejp. Fgcd2 command parser specification, 2022. URL `https://gitlab.cern.ch/mcejp/fgcd2-command-parser/-/blob/master/SPEC.md#ncrp-flavor`.

[5] CERN. Fgc reference pages. URL `https://accwww.cern.ch/proj-fgc/gendoc/def/SiteMap.htm`. Accessed: 2023.

[6] M. Cerqueira Bastos. High Precision Current Measurement for Power Converters. *Proceedings of the CAS-CERN Accelerator School: Power Converters*, 2015. doi: 10.5170/CERN-2015-003.353. URL `https://cds.cern.ch/record/2038670`.

[7] M. Cerqueira Bastos. Cern current calibrator upgrade proposal. Technical Report EDMS 2186692 v2, CERN, 2018.

[8] M. Cerqueira Bastos. New cdc controller (ncc) software requirements. Technical Report EDMS 2894001 v3, CERN, 2023.

[9] M. Cerqueira Bastos, G. Fernqvist, G. Hudson, J. Pett, A. Cantone, F. Power, A. Saab, B. Halvarsson, and J. Pickering. High accuracy current measurement in the main power converters of the large hadron collider: tutorial 53. *IEEE Instrumentation I& Measurement Magazine*, 17(1):66–73, 2014. doi: 10.1109/MIM.2014.6783001.

[10] P. Developers. Pytest documentation, 2022. URL `https://docs.pytest.org/en/7.4.x/`. Accessed: 2023.

[11] S. Developers. Sphinx documentation, 2022. URL `https://www.sphinx-doc.org/en/master/`. Accessed: 2023.

[12] A. Devices. Ad7689 datasheet, 2018. URL `https://www.analog.com/media/en/technical-documentation/data-sheets/ad7682_7689.pdf`. Accessed: 2023.

[13] A. Devices. Ad8655/ad8656 datasheet, 2018. URL `https://www.analog.com/media/en/technical-documentation/data-sheets/ad8655_8656.pdf`. Accessed: 2023.

[14] A. Devices. Adr1000 datasheet, 2022. URL `https://www.analog.com/media/en/technical-documentation/data-sheets/adr1000.pdf`. Accessed: Rev. B 2023.

[15] M. Di Cosmo and B. Todd. The New Modular Control System for Power Converters at CERN. *ICALEPCS2015 Conference Proceedings*, page WEPGF005, 2015. doi: 10.18429/JACoW-ICALEPCS2015-WEPGF005. URL `https://cds.cern.ch/record/2213493`.

[16] Digilent. Cora z7 schematic, 2018. URL `https://s3-us-west-2.amazonaws.com/digilent/resources/programmable-logic/cora-z7/Cora+Z7_sch-public.pdf`. Accessed: 2023.

[17] A. Electronics. Fnp series high power precision shunt resistor, 2021. URL `https://www.rhopointcomponents.com/wp-content/uploads/2021/03/alpha_fnp.pdf`. Accessed: 2023.

[18] G. Fernqvist, B. Halvarsson, and J. Pett. The cern current calibrator-a new type of instrument. In *Conference Digest Conference on Precision Electromagnetic Measurements*, pages 410–411, 2002. doi: 10.1109/CPEM.2002.1034894.

[19] G. Fernqvist, G. Hudson, J. Pickering, and F. Power. Design and evaluation of a 10-mA DC current reference standard. *IEEE Trans. Instrum. Meas.*, 52(2):440–4, 2003. doi: 10.1109/TIM.2003.809915. URL `https://cds.cern.ch/record/643294`.

[20] Fluke. 732b/734a dc reference standard, 1992. URL `https://download.flukecal.com/pub/literature/732b734aimeng_Rebrand0300.pdf`. Accessed: 2023.

[21] Fluke. Fractional ppm traceability using your fluke 734a/732b, 2011. URL `https://download.flukecal.com/pub/literature/1260304D_734A_732B_Fract_Trace_AN_w.pdf`. Accessed: 2023.

[22] Fluke. 8588a reference multimeter specification, 2019. URL https://download.flukecal.com/pub/literature/8588A___pseng0700.pdf. Accessed: Rev G, 2023.

[23] T. Instruments. *LVDS Application and Data Handbook*, 2002. URL https://www.ti.com/lit/ug/slld009/slld009.pdf. Accessed: 2023.

[24] T. Instruments. Sn74lvc8t245 datasheet, 2005. URL https://www.ti.com/lit/ds/symlink/sn74lvc8t245.pdf. Accessed: 2023.

[25] T. Instruments. Sn65lvdxx high-speed differential line drivers and receivers, 2014. URL https://www.ti.com/lit/ds/symlink/sn65lvds2.pdf. Accessed: 2023.

[26] T. Instruments. Opa189 datasheet, 2021. URL https://www.ti.com/lit/ds/symlink/opa189.pdf. Accessed: 2023.

[27] Keysight. 3458a multimeter specification, 2019. URL https://www.keysight.com/us/en/assets/7018-06796/data-sheets/5965-4971.pdf. Accessed: 2023.

[28] M. L. Kidd. Watch out for those thermoelectric voltages!, 2012. URL https://www.cal-labmagazine.com/2012/04/01/watch-out-for-those-thermoelectric-voltages/. Accessed: 2023.

[29] Q. King, N. Laurentino Mendes, and M. Cejp. Fgc command/response protocol, 2018. URL https://wikis.cern.ch/pages/viewpage.action?pageId=70682032#FGCCommand/ResponseProtocol-NetworkCommandResponseProtocol(NCRP).

[30] M. Kovačić. The upgraded cern current calibrator techical seminar, 2022. URL https://indico.cern.ch/event/1159341/attachments/2463036/4223103/EPC_seminar_MK_fin.pdf.

[31] R. Lapuh, J. Kucera, J. Kovac, and B. Voljc. Fluke 8588a and keysight 3458a dmm sampling performance. 2022. URL https://arxiv.org/abs/2205.11321. Accessed: 2023.

[32] E. Lopienska. The cern accelerator complex, 2022. URL https://cds.cern.ch/images/CERN-GRAPHICS-2022-001-1.

[33] M. D. Ltd. Iref2 user's manual, 2016.

[34] M. D. Ltd. Iref3 user's and service manual v3 2, 2023.

[35] V. Nappi. Cdc control application software repository, 2023. URL https://gitlab.cern.ch/cdc-control/cdc-control-application. Accessed: 2023.

[36] P. Odier, M. Ludwig, and S. Thoulet. The DCCT for the LHC Beam Intensity

Measurement. Technical report, CERN, Geneva, 2009. URL `https://cds.cern.ch/record/1183400`.

[37] Toradex. Long term availability, 2022. URL `https://developer.toradex.com/hardware/hardware-resources/long-term-availability/`. Accessed: 2023.

[38] Xilinx. *7 Series FPGAs SelectIO Resources User Guide*, 2017. URL `https://www.xilinx.com/support/documentation/user_guides/ug471_7Series_SelectIO.pdf`. Accessed: 2023.

[39] Xilinx. *Zynq-7000 SoC Technical Reference Manual*, 2017. URL `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`. Accessed: 2023.

[40] Xilinx. *UltraFast Design Methodology Guide for the Vivado Design Suite*, 2018. URL `https://www.xilinx.com/support/documents/sw_manuals/ug1046-ultrafast-design-methodology-guide.pdf`. Accessed: 2023.

[41] Xilinx. *AXI Quad SPI v3.2 LogiCORE IP Product Guide*, 2020. URL `https://www.xilinx.com/support/documentation/ip_documentation/axi_quad_spi/v3_2/pg153-axi-quad-spi.pdf`. Accessed: 2023.

[42] Xilinx. Amd extends product lifecycle for all xilinx 7 series devices through at least 2035, 2022. URL `https://community.amd.com/t5/adaptive-computing/amd-extends-product-lifecycle-for-all-xilinx-7-series-devices/ba-p/563507`. Accessed: 2023.

A | CDC Properties

Here all the FGC properties are documented:

A.1. CALSYS.CDC.ADC.GAIN_1A

EXPERT PROPERTY Setting this expert property with no parameter and the CDC output set to 1A launches a gain calibration for the CDC output current (ADC channel 2). When read, it retrieves the value of the calibrated gain.

Persistent

Type: FLOAT

Range: [0.40000 .. 1.60000]

Default: 1.00000

A.2. CALSYS.CDC.ADC.I_MEAS

Accessing this property triggers a reading of the ADC channel which measures the output current of the CDC and returns a value in Amp.

SET not permitted

Type: FLOAT

A.3. CALSYS.CDC.ADC.I_NULL

Accessing this property triggers a reading of the ADC channel which measures the offset current of the CDC and returns a value in µAmp.

SET not permitted

Type: FLOAT

A.4. CALSYS.CDC.ADC.PBC_TEMP

Accessing this property triggers a reading of the ADC channel which measures the PBC temperature of the CDC and returns a value in ºC.

SET not permitted

Type: FLOAT

A.5. CALSYS.CDC.ADC.RAW_ZERO

EXPERT PROPERTY This property contains the raw value resulting from the zero calibration of the ADC.

Persistent.

SET not permitted.

Type: INT16S

Default: 0

A.6. CALSYS.CDC.ADC.V_MEAS

Accessing this property triggers a reading of the ADC channel which measures the output voltage of the CDC and returns a value in Volt. This measurement includes the voltage across the secondary windings of the CDC and across the current sense resistor.

SET not permitted

Type: FLOAT

A.7. CALSYS.CDC.DAC1TW_CAL

EXPERT PROPERTY This property contains the calibration state of the DAC 1TW system. Setting this property with no parameters will trigger a calibration of channel 0 of the DAC. The DAC calibration is not permitted in VB mode and it requires the CDC to be ON and at zero current with no offset present. The calibration of the zero is done first followed by the calibration of the values corresponding to 10mA which is measured by doing a back to back between the DAC winding with the DAC at full scale and the “calibration” winding with the 10mA reference.

Persistent

Range: `CALIBRATED`, `UNCALIBRATED`

A.8. CALSYS.CDC.DAC1TW.ZERO

EXPERT PROPERTY This property contains the value, in volt, of the DAC output that generates a zero current in the DAC winding.

Persistent

SET not permitted

Type: FLOAT

Default: 0.00000

A.9. CALSYS.CDC.DAC1TW.FS

EXPERT PROPERTY This property contains the value, in volt, of the DAC output that generates a 10mA current in the DAC winding.

Persistent

SET not permitted

Type: FLOAT Default: 10.00000

A.10. CALSYS.CDC.DAC.RAW

EXPERT PROPERTY This property returns the raw value corresponding to the preset output of the specified DAC channel.

SET not permitted

Type: INT16S

A.11. CALSYS.CDC.DAC.VOLT

EXPERT PROPERTY This property sets the specified DAC channel output to the indicated value in Volt. When read, it returns the last value sent to the specified DAC channel.

Type: FLOAT

Range: [10.00000 .. 10.00000]

A.12. CALSYS.CDC.I_REF

Setting this property sets the CDC to the requested I_REF. Reading this property returns the present value of I_REF. This operation may take up to 5s.

Type: FLOAT Range: depends on the `CALSYS.CDC.OUTPUT` property

`CALSYS.CDC.OUTPUT = RANGE_1A:` [-1.0000000 .. 1.0000000]

`CALSYS.CDC.OUTPUT = RANGE_2A5:` [-2.5000000 .. 2.5000000]

`CALSYS.CDC.OUTPUT = RANGE_5A:` [-5.0000000 .. 5.0000000]

`CALSYS.CDC.OUTPUT = RANGE_10A:` [-10.0000000 .. 10.0000000]

A.13. CALSYS.CDC.NULL.AUTO

Setting this property will trigger a CDC offset nulling action. The CDC offset nulling can only be done with the CDC on at zero current and when not in VB mode. This operation may TAKE up to 5s.

GET not permitted

A.14. CALSYS.CDC.NULL.KI

EXPERT PROPERTY This expert property contains the value of the proportional gain on the PI controller used for the offset nulling.

Persistent

Type: FLOAT

A.15. CALSYS.CDC.NULL.KP

EXPERT PROPERTY This expert property contains the value of the proportional gain on the PI controller used for the offset nulling.

Persistent

Type: FLOAT

A.16. CALSYS.CDC.NULL.NR

EXPERT PROPERTY This expert property contains the value of the maximum number of offset nulling attempts (cycles ADC reading/ DAC setting).

Type: INT16U

A.17. CALSYS.CDC.OUTPUT

Setting this property switches the current calibrator OFF or ON in the indicated RANGE. If this property is set to OFF and there is current present at the CDC output, the current is set to zero before turning off the CDC. If this property is set to a valid current range (see below) the CDC status is verified before switching the CDC on to that range. If there are any alarms active, the action is not performed and an error message is returned. Once an ON command is accepted and the relay configuration has been set, the CDC offset current is nulled. This operation may take up to 5s.

SET not permitted

Type: Range: `OFF`, `RANGE_1A`, `RANGE_2A5`, `RANGE_5A`, `RANGE_10A`

A.18. CALSYS.CDC.STATUS.RELAYS

EXPERT PROPERTY This bitmask symbol list is used to identify all the presently active relays in the current calibrator zero flux detector. Accessing this property will trigger a reading of the relay status.

SET not permitted

Range: `INV2048T`, `ON2048T`, `INV1024T`, `ON1024T`, `INV512T`, `ON512T`, `INV256T`, `ON256T`, `INV128T`, `ON128T`, `INV64T`, `ON64T`, `INV32T`, `ON32T`, `INV16T`, `ON16T`, `INV8T`, `ON8T`, `INV4T`, `ON4T`, `INV2T`, `ON2T`, `INV1T`, `ON1T`, `INV1TCAL`, `ON1TCAL`, `INV1TDAC`, `ON1TDAC`, `ONS4T`, `ONS8T`, `ONS16T`, `ONS40T`

A.19. CALSYS.CDC.VB_MODE

Getting this property will tell you if a VB is connected to the CDC (VB_MODE =1) or not (VB_MODE =0).

SET not permitted

Type: INT18U

A.20. CALSYS.DCM.ADC.CAL

EXPERT PROPERTY

Setting this property with no parameters will trigger an ADC calibration. The ADC calibration uses the value in DCM.ADC.VREF to correct for the full scale calibration. Since the ADC channels are multiplexed, the calibration is valid for all channels.

Range:

- UNCALIBRATED
- CALIBRATED

A.21. CALSYS.DCM.ADC_RAW

EXPERT PROPERTY Accessing this property triggers an ADC reading for the specified channel and returns the value in raw reading. If in CDC mode the ADC channels are differential and only 8 channels are available. If in SM mode the ADC channels are single ended and 16 channels are available.

SET not permitted

Type: INT16

A.22. CALSYS.DCM.ADC.VOLT

EXPERT PROPERTY Accessing this property triggers an ADC reading, for the specified channel and returns the value in Volt. If in CDC mode the ADC channels are differential and only 8 channels are available. If in SM mode the ADC channels are single ended and 16 channels are available.

SET not permitted

Type: FLOAT

A.23. CALSYS.DCM.PS05.CAL

EXPERT PROPERTY This property accepts a value for calibration of the 5V power supply voltage ADC reading. To calibrate, the 5V voltage must be read with a calibrated DVM and the PS05.CAL property should be set to the measured value. A gain error is calculated automatically.

Type: FLOAT

Range: [4.00000 .. 6.00000]

A.24. CALSYS.DCM.PS05.MEAS

Accessing this property triggers a reading of the 5V power supply voltage and returns the value in Volt.

SET not permitted

Type: FLOAT

A.25. CALSYS.DCM.PS12.CAL

EXPERT PROPERTY This property accepts a value for calibration of the 12V power supply voltage ADC reading. To calibrate, the 12V voltage must be read with a calibrated DVM and the PS12.CAL property should be set to the measured value. A gain error is calculated automatically.

Type: FLOAT

Range: [11.0000 .. 13.0000]

A.26. CALSYS.DCM.PS12.MEAS

Accessing this property triggers a reading of the 12V power supply voltage and returns the value in Volt.

SET not permitted

Type: FLOAT

A.27. CALSYS.DCM.PS15.CALNEG

EXPERT PROPERTY This property accepts a value for calibration of the -15V power supply voltage ADC reading. To calibrate, the -15V voltage must be read with a calibrated DVM and the PS15.CALNEG property should be set to the measured value. A gain error is calculated automatically.

Type: FLOAT

Range: [-16.0000 .. -14.0000]

A.28. CALSYS.DCM.PS15.CALPOS

EXPERT PROPERTY This property accepts a value for calibration of the +15V power supply voltage ADC reading. To calibrate, the +15V voltage must be read with a calibrated DVM and the PS15.CALPOS property should be set to the measured value. A gain error is calculated automatically.

Type: FLOAT

Range: [14.0000 .. 16.0000]

A.29. CALSYS.DCM.PS15.MEASNEG

Accessing this property triggers a reading of the -15V power supply voltage and returns the value in Volt.

A.30. CALSYS.DCM.PS15.MEASPOS

Accessing this property triggers a reading of the +15V power supply voltage and returns the value in Volt.

A.31. DEVICE.VERSION.MAIN_PROG

This property indicates the current version of the software.

SET not permitted

Type: INT16U Range: INT16U

A.32. STATUS.FAULTS

This bitmask symbol list is used to identify all the faults that may be reported by class 150.

SET not permitted

Range per class: Class 150

- PBC_TMP: PBC temperature fault
- PBC_CAL: PBC calibration fault
- PBC_CMPL: PBC compliance fault
- PBC_CHR: PBC charge fault
- VB_PWR: voltage booster unpowered
- VB_TMP: voltage booster over temperature
- PA_TMP: power amplifier over temperature
- MOD_TMP: modulator amplifier over temperature

A.33. STATUS.WARNINGS

This bitmask symbol list is used to identify all the warnings that may be reported by classes 150 and 151.

SET not permitted

Range per class: Class 150

- PBC_TMP: PBC temperature warning
- CLP_ON: clamp on when it should be off
- CLP_OFF: clamp off when it should be on

List of Figures

List of Tables

Acronyms

1-Wire	1-Wire. 21, 23, 24, 26, 28, 30, 31, 34, 37, 38, 44, 57, 70, 72
1TDAC	One-Turn DAC Driver. 14
ACTFILT	Active Filter. 14
ADC	Analog to Digital Converter. 23, 24, 26, 31, 37, 43, 44, 73
ALICE	A Large Ion Collider Experiment. 2
ATLAS	A Toroidal LHC Apparatus. 2, 3
AXI	Advanced eXtensible Interface. 37–40, 42–44, 53, 71
BE-CEM	Beams department - Controls Electronics and Mechatronics group. 25
BOM	Bill of Materials. 32
BRAM	Block RAM. 44
BSP	Board Support Package. 48
CD	Continuous Deployment. 80
CDC	CERN DCCT Calibrator. 7–15, 17, 18, 25, 28, 33, 34, 52, 57, 62–68, 70–72, 74, 92, 95
CERN	Conseil Européen pour la Recherche Nucléaire. 1–4, 7, 8, 14, 15, 18, 22, 24–26, 28, 30, 38, 79, 93
CI	Continuous Integration. 79, 80
CI/CD	Continuous Integration and Continuous Deployment. 18, 19, 80, 97
CLK	Clock. 30, 34
CMS	Compact Muon Solenoid. 2, 3
COTS	Commercial Off-The-Shelf. 30, 34, 73, 85

CRP Command-Response Protocol. 13

CS Chip Select. 30, 34

DAC Digital to Analog Converter. 9, 66, 67, 76

DCCT Direct-Current Current Transformer. 3, 4, 6–9, 11

DI/OT Distributed I/O Tier. 25, 26, 28

DSP Digital Signal Processor. 13

DTB Device Tree Blob. 51, 53

DTS Device Tree Source. 51, 53

DTSI Device Tree Source Include. 51–53

DVM Digital Voltmeter. 85–88, 90, 93

EDA Electronic Design Automation. 34

EEPROM Electrically Erasable Programmable Read-Only Memory. 55

EMI Electromagnetic Interference. 12

FADEM Feedback and Demodulation Amplifier. 14, 37, 42, 57, 66, 67, 72–74

FGC Function Generator Controller. 4, 14, 17, 44, 57–60, 64, 65

FIFO First In First Out. 42

FPGA Field Programmable Gate Array. 18, 19, 22–24, 28, 30, 37, 38, 42–45, 47, 82, 87

FSBL First Stage Boot Loader. 50, 51, 55

GPIO General Purpose Input Output. 23, 24, 26, 38, 39, 43, 53, 71

HDF Hardware Description File. 47, 51

Hi-Lumi High Luminosity LHC. 3, 11, 93

HTTP Hypertext Transfer Protocol. 57, 71, 72

IC Integrated Circuit. 30, 33, 34, 73

IO Input Output. 26, 42–45, 76

IP Intellectual Property. 30, 37–40, 42–45, 53, 71, 72

LFSR Linear Feedback Shift Register. 44, 45

LHC Large Hadron Collider. 2–4, 8

LHCb Large Hadron Collider beauty. 2

LINAC2 Linear Accelerator 2. 2

LINAC4 Linear Accelerator 4. 2

LVDS Low Voltage Differential Signaling. 21, 26, 29, 30, 34, 37, 40, 42

MAC Media Access Control. 55

METAS Swiss Federal Institute of Metrology. 89

MISO Master In Slave Out. 30, 34

MODAMP Modulation Amplifier. 13

MOSI Master Out Slave In. 30, 34

MutEx Mutual Exclusion. 64, 70

NCRP Network Command-Response Protocol. 13–15, 17, 57–60, 62–64, 72

NETCTRL Network Controller. 14, 64

NTP Network Time Protocol. 72

NULLMETER Nullmeter. 14, 37, 57, 66, 72

PI Proportional Integral. 8

PL Programmable Logic. 37–39, 42–44, 51–53

PRIMREL Primary Relays. 14, 37, 42, 57, 66, 67, 71–73, 76

PS Proton Synchrotron. 2

PS Processing System. 37, 39, 40, 42, 43, 51

PSB Proton Synchrotron Booster. 2

PSU Power Supply Unit. 13

RegFGC Regulating FGC. 4

RGB Red Green Blue. 43

RootFS Root File System. 54

SD Secure Digital. 51

SECREL Secondary Relays. 14

SoC System-on-Chip. 13, 18, 22–26, 28, 29, 31, 33, 42, 43

SoM System on Module. 23, 25–28

SPI Serial Peripheral Interface. 14, 21, 23, 24, 26, 29, 30, 34, 37, 38, 40, 42, 43, 52, 57, 72, 73, 76

SPS Super Proton Synchrotron. 2

SSH Secure Shell. 54

SY-EPC Systems department - Electrical Power Converters group. 24

SY-EPC-CCE SY-EPC - Converter Control Electronics section. 38

SY-EPC-CCS SY-EPC Converter Control Software section. 15

SY-EPC-HPM SY-EPC High Precision Measurements section. 11, 89, 93

TCP Transmission Control Protocol. 13, 64

TFTP Trivial File Transfer Protocol. 51

UART Universal Asynchronous Receiver/Transmitter. 39

USB Universal Serial Bus. 51

VB Voltage Booster. 11

VHDL VHSIC Hardware Description Language. 37

XADC Xilinx Analog to Digital Converter. 38, 43, 53, 57, 70

ZFD Zero-Flux Detector. 14

Acknowledgements

I would like to thank my supervisor, Prof. Federica Villa, for her support and guidance throughout this project. Her supervision and advice were truly helpful in the development of the thesis. I am also grateful to her for the opportunity to work on this project and for the trust she has placed in me.

I would also like to thank my co-supervisor, Miguel Cerqueira Bastos, to whom I owe the opportunity to work on this project, for supporting my work with the instrument, providing clear and deep explanations of the internal workings and of the analog phenomena that govern the instrument, as well as for evaluating my proposals and for discussing with me every doubt I had.

I want to thank also Dr. Slawosz Uznanksi, who has been a great help in setting a clear path for the project. I wish him a brilliant career in his new position as an Astronaut at ESA.

I owe a thanks to Adrian Byszuk, for his constant help and incredible expertise in the FPGA and embedded world, he never missed a point with his answers to every question I had and provided a huge day-to-day support in the development of the project.

I would like to thank my colleagues in the CERN HPM team for their invaluable support in the high-precision measurements and their help in the characterization of the instrument, in particular to Nikolai Beev for dedicating a lot of time to ensure that I could obtain the best possible measurements.

I would also like to thank the rest of my colleagues in the CERN CCE section for their support in the development and for providing the necessary resources to carry out this project. The team welcomed me from the first day, allowing me to integrate immediately into the team and work in a stimulating and collaborative environment.

I would like to express my gratitude to my family and friends for their support throughout my academic journey. Their encouragement, love, and understanding have been instrumental in helping me achieve my goals.

Finally, I would like to thank my girlfriend, Silvia, for her support and for being always by my side, even in the most difficult moments.