UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Sede Amministrativa: Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Scuola di Dottorato di Ricerca in Ingegneria dell'Informazione

Indirizzo: Scienza e Tecnologia dell'informazione

Ciclo XXVII

Study, Design, Development and Implementation of Distributed Control Systems using EPICS for the SPES Project

Direttore della Scuola: Ch.mo Prof. Matteo Bertocco

Coordinatore d'indirizzo: Ch.mo Prof. Carlo Ferrari

Supervisore: Ch.mo Prof. Matteo Bertocco

Ch.mo Dr. Alberto Andrighetto

Ch.mo Dr. Marco Bellato

Dottorando: Jesus Alejandro Vasquez Stanescu

I dedicate this work to my fiancée Mary and my family. I am here today thanks to their constant and unconditional love and support.

Thanks to Dr. Alberto Andrighetto, Dr. Marco Bellato, and Dr. Matteo Bertocco. They welcomed me into their groups, taught me important lessons, and gave me all the support and knowledge that I needed on all times.

A special thanks to all my friends and colleges who helped me to accomplish this important goal.

Contents

Table of Figures

Table of Tables

List of abbreviations

A/D	Analog-To-Digital
ADC	Analog-To-Digital Converter
CA	Channel Access
CB	Charge Breeder
COM	Computer-On-Modules
CPU	Central Processing Unit
CSS	Control System Studio
D/A	Digital-To-Analog
DAC	Digital-To-Analog Converter
DB	Decibel
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
EBIS	Electron Beam Ion Source
ECR	Electron Cyclotron Resonance
ENOB	Effective Number Of Bits
EPICS	Experimental Physics And Industrial Control System
EURISOL	European Isol Facility
FFT	Fast Fourier Transform
FGPA	Field-Programmable Gate Array
GPIO	General-Purpose Input/Output
GUI	Graphical User Interface
HRMS	High-Resolution Mass Separator
I/O	Input/Output
IDE	Integrated Development Environment
INFN	Istituto Nazionale Di Fisica Nucleare
IO	Input/Output
IOC	Input Output Controllers
IP	Internet Protocol
ISOL	Isotope Separation On Line
LINAC	Linear Accelerator
LNL	Laboratori Nazionali Di Legnaro
MCP	Microchannel Plate
MPS	Machine Protection System
NTP	Network Time Protocol
OS	Operating System
OWFS	1-Wire File System
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PID	Proportional-Integral-Derivative

PLC	Programmable Logic Controller
PoE	Power Over Ethernet
PPS	Personal Protection System
PV	Process Variables
RFQ	Radio-Frequency Quadrupole
RIB	Radioactive Ion Beam
RSTP	Rapid Spanning Tree Protocol
SAR	Successive Approximation Register
SATA	Serial ATA
SCADA	Supervisory Control And Data Acquisition
SD	Secure Digital
SIL	Safety Integrity Level
SINAD	Signal To Noise And Distortion Ratio
SoC	System On Chip
SPES	Selective Production Of Exotic Species
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USB	Universal Serial Bus
VGA	Video Graphics Array
VPN	Virtual Private Networks
XML	Extensible Markup Language

Abstract

One of the most important project supported by the INFN (Istituto Nazionale di Fisica Nucleare) it is SPES (Selective Production of Exotic Species), which aims to develop a facility for the production of RIBs (Radioactive Ion Beams) and radioisotopes for nuclear medicine applications. The LNL (Laboratori Nazionali di Legnaro) was chosen as the site for the construction and operation of SPES. This kind of facility brings many technology challenges, which require research and development on the mechanical, structural and material fields, as well as on the control systems for the coordinated deployment of the accelerator facilities. For the past five years, an off-line laboratory has been in operation at the LNL. This laboratory serves as a research and development test bench for engineers of different fields. New devices and techniques developed on this laboratory will be later be implemented on the final SPES facility.

For the SPES project, EPICS (Experimental Physics and Industrial Control System) was chosen as the standard framework for the control systems. The general architecture is a series of distributed IOCs (Input Output Controllers) for controlling all the accelerator instrumentations, using a custom communication protocol knows as Channel Access (CA). For critical application from the safety point of view, it was chosen to use PLC (Programmable Logic Controllers), instead of native EPICS controllers. These applications comprises mainly interlock systems for both Machine Protections Systems (MPS) and Personnel Protection Systems (PPS). Furthermore, for PPS certificated safety PLCs must be used. Although PLCs will directly control these systems, an interface to EPICS must be available for sharing process values with the rest of the facility control systems.

The arrival of the SPES project to the LNL has also demanded a review and upgrade of many of the control systems present in the laboratories, in particularly those that are going to be directly interconnected to SPES, as it is the case of the ALPI superconductive LINAC (Linear Accelerator) control system. These upgrades will allowed an interconnection and interoperability of all the systems using the EPICS CA transport layer. This imply to develop and implement EPICS IOCs, not only for controlling the SPES instrumentation, but also for all the instrumentation present on the LNL accelerator complex. Consequently, a standardization of the hardware and software used to develop all these control systems is fundamental.

In this context, it is evident the need of custom I/O controllers that embed different functionalities to cope with the needs of many subsystems, e.g. tape system, charge breeder, target and ion source, etc. This

need has driven the development of a standard hardware platform for the embedding of the EPICS IOC of all the future control systems. This will help to considerably reduce costs and maintenance efforts.

For the off-line laboratory, a low-cost embedded IOC was designed using the computer board Raspberry Pi (Model B, rev. 2). In order to provide data acquisition capabilities to this board, tailored expansion boards where developed at LNL. They are connected to the GPIO port of the Raspberry PI and interface drivers for EPICS were implemented. Diverse kinds of boards have been developed for specify applications. Four of these IOC were implemented on the off-line laboratory for: the beam diagnostic data acquisition system, the mass separator control system, the vacuum measurement data acquisition system and the PLC to EPICS interface. The performance results demonstrate that this IOC is capable of manage this instrumentation under the conditions requested by the operators.

While a low-cost solution is completely valid for the off-line laboratory, SPES and the LNL accelerators require a more robust and standard solution, which can operated under industrial environments and constructed with hardware components that can be available on the marked for the major part of the life cycle of the accelerators. In this sense, a second version of EPICS IOCs were developed using as hardware platform the COM (Computer on Modules) form factor. The peripheral devices needed for developing all the foreseen control systems to be used at LNL will be connected to the COM, such as ADCs, DACs, stepper motors controllers, etc. They will be installed on a custom Carrier Board along with the COM. This new IOC is intended as a basic construction block for all the control systems at LNL. In the next years, all the new control systems will target this hardware device.

Currently at LNL, prototypes of this new EPICS IOC have been developed using an ADLINK Type 6 COM Express module on a generic carrier board along with commercial ADCs and DACs PCI Express boards. They have been install on different application at LNL. The performance test results show that these IOCs are more than capable of manage the accelerator instrumentations as required. The CPU units have more than enough computing power required by the control system algorithms, while consuming low power.

For the application based on PLCs, one of the first system to be upgrade at the LNL accelerator complex, in order to be compatible with SPES, was the personnel access control system. The principal motivations were its obsolescence and the need of its interconnection to EPICS. This is a new fail-tolerant system, based on a redundant CPU PLC, in conjunction with remote IO islands, connected on a redundant optic fiber communication ring. For this new system, it was implemented an EPICS interface, in order to access all the status of the system for sharing it with the rest of the control systems.

Sommario

Uno dei progetti più importanti dell'INFN (Istituto Nazionale di Fisica Nucleare) è SPES (Selective Production of Exotic Species), lo scopo del quale è lo sviluppo di un complesso per la produzione di RIBs (Radioactive Ion Beams) e radioisotopi per applicazione di medicina nucleare. I LNL (Laboratori Nazionali di Legnaro) è stato scelto come luogo per la costruzione ed operazione di SPES. Questo tipo di complesso presenta un gran numero di sfide tecniche che richiedono sviluppi nei campi della meccanica e materiali, cosi come anche nel campo dei sistemi di controllo per l'implementazione coordinata delle funzione dell'acceleratore. Negli ultimi cinque anni, un laboratorio off-line è stato in operazione nei LNL. Questo laboratorio serve come banco di prove ingegneristico per la ricerca e sviluppi in diversi campi. Nuovi dispositivi e tecniche sviluppati in questo laboratorio saranno implementate poi nel complesso finale di SPES.

Per il progetto SPES, è stato scelto EPICS (Experimental Physics and Industrial Control System) come quadro generale per lo sviluppo dei sistemi di controllo. L'architettura generale del sistema è basata su una serie di IOC (Input/Output Controllers) distribuiti, per il controllo della strumentazione del acceleratore, usando un protocollo di comunicazione proprio chiamato Channel Access (CA). Dall'altra parte, per applicazione critiche dal punto di vista della sicurezza, è stato scelto l'utilizzo di PLC (Programmable Logic Controllers), invece di controlli EPICS nativi. Queste applicazioni comprendono fondamentalmente sistemi di interlocks per sistemi di protezione sia per le persone (PPS) sia per le machine (MPS). Inoltre, per i sistemi di protezione alle persone, devono essere usati PLC di sicurezza certificati. Mentre i PLC controlleranno direttamente ogni sistema, deve esserci disponibile una interfaccia di comunicazione verso EPICS in modo di condividere i valori di processo col resto dei sistemi del complesso.

L'arrivo del progetto SPES ai LNL ha richiesto una revisione ed aggiornamento di molti sistemi di controllo presenti nei laboratori, in particolare quelli che saranno interconnessi con SPES, come nel caso del sistema di controllo del LINAC (Linear Accelerator) superconduttivo ALPI. Questi rinnovamenti permetteranno la interconnessioni ed interoperabilità di tutti i sistemi, usando il CA di EPICS come protocollo di comunicazione generale. Questo implica che devono essere sviluppati IOC EPICS, non solo per la strumentazione di SPES, ma anche per tutta la strumentazione presente del complesso di acceleratori dei LNL. Conseguentemente, è fondamentale una standardizzazione del hardware e software usato per lo sviluppo dei sistemi di controllo.

In questo contesto, è evidente la necessità di controllori fatti a misura con diverse funzionalità integrate per coprire le necessità di molti sottosistemi, come ad esempio, tape systems, charge breeder, target e ion source, ecc. Questa necessità ha promosso lo sviluppo di piattaforme hardware standard per i IOC EPICS che saranno usati per tutti i sistemi di controllo futuri. Questo aiuterà a ridurre considerevolmente i costi ed impegni di manutenzione.

Per il laboratorio off-line, è stato disegnato un sistema IOC a basso costo usando il computer a scheda singola Raspberry Pi (Model B, rev. 2). Per fornire a questo dispositivo capacità di acquisizione dati, sono state sviluppate, ai LNL, schede di espansione su misura. Queste schede sono state connesse alla porta GPIO del Raspberry Pi e drivers di interfaccia per EPICS sono stati implementati. Diverse tipi di schede sono stati sviluppati per applicazioni specifiche. Quattro di questi IOC sono stati implementati nel laboratorio off-line per: il sistema di acquisizione dati della diagnostica di fascio, il sistema di controllo del separatore di massa, il sistema di acquisizione dati dei misuratori di vuoto e l'interfaccia di comunicazione tra PLC ed EPICS. I resultati delle prove di prestazione dimostrano che questo IOC è capace di gestire la strumentazione, sodisfacendo i requisiti imposti dagli operatori.

Mentre una soluzione a basso costo è totalmente valida per il laboratorio off-line, SPES ed l'acceleratori ai LNL richiedono una soluzione più robusta e standard, la quale possa operare sotto condizioni industriali cosi come essere costruita con componenti hardware che abbiamo una diponibilità in mercato per la maggiore parte del ciclo di vita utile dei acceleratori. Per questo motivo, una seconda versione di EPICS IOC è stata sviluppata usando come piattaforma hardware il form factor COM (Computer-on-Module). I dispositivi periferici necessari per lo sviluppo di tutti i sistemi di controllo previsti ai LNL saranno connessi al modulo COM, come per esempio ADCs, DACs, motori a passo, ecc. Tutti questi dispositivi periferici saranno istallati su una Carrier Board insieme al modulo COM. Questo nuovo IOC è intesso come un blocco standard di costruzione per tutti i sistemi di controllo ai LNL. Nei prossimi anni, tutti i nuovi sistemi di controllo punteranno a questo dispositivo hardware.

Attualmente ai LNL, sono stati sviluppati prototipi di questi IOC usando moduli COM Express Type 6 dell'ADLINK, istallati su carrier boards generiche, insieme a schede commerciali, di tipo PCI express, con ADCs e DACs. Questi prototipi sono stati istallati in diverse applicazioni ai LNL. I resultati delle prove di prestazioni mostrano che questi IOC sono più che capaci di gestire la strumentazione dell'acceleratori, soddisfacendo le condizioni richieste. L'unità CPU possiedono potere di computo più che sufficiente per implementare l'algoritmi di controllo, mantenendo un consumo di potenza basso.

Per il caso dell'applicazione basate su PLCs, uno dei primi sistemi del complesso di acceleratori ai LNL scelto per essere aggiornato, in modo di essere compatibile con SPES, è il sistema di controllo degli accessi. Le motivazioni principali sono stati l'obsolescenza del sistema precedente cosi come la necessità della sua futura interconnessione alla rete EPICS. Il nuovo è un sistema di tipo fail-tolerant, il quale usa un PLC con CPU ridondata, congiuntamente con delle isole di IO remote collegate mediante una rete di comunicazione in fibra ottica con topologia ridondata in anello. Per questo nuovo sistema, è stata implementata una interfaccia di comunicazione verso EPICS, mediante la quale è possibile accedere allo stato di tutto il sistema, per la sua condivisione col resto dei sistemi di controllo.

Introduction

One of the most important project supported by the INFN (Istituto Nazionale di Fisica Nucleare) is SPES (Selective Production of Exotic Species), which aims to develop a facility for the production of RIBs (Radioactive Ion Beams) and radioisotopes for nuclear medicine applications. The LNL (Laboratori Nazionali di Legnaro) was chosen as the site for the construction and operation of SPES. This new kind of facility brings many technology challenges, which require research and development on the mechanical, structural and material fields, as well as on the control systems for the coordinated deployment of the accelerator facilities. For the past five years, an off-line laboratory has been in operation at the LNL. This laboratory serves as a research and development test bench for engineers of different fields. New devices and techniques developed on this laboratory will be later be implemented on the final SPES facility.

For the SPES project, EPICS (Experimental Physics and Industrial Control System) was chosen as the standard framework for the control systems. The general architecture is a series of distributed IOCs (Input Output Controllers) for controlling all the accelerator instrumentations, using a custom communication protocol knows as Channel Access (CA). For critical application from the safety point of view, it was chosen to use PLC (Programmable Logic Controllers), instead of native EPICS controllers. These applications comprises mainly interlock systems for both Machine Protections Systems (MPS) and Personnel Protection Systems (PPS). Furthermore, for PPS certificated safety PLCs must be used. Although PLCs will directly control these systems, an interface to EPICS must be available for sharing process values with the rest of the facility control systems.

The arrival of the SPES project to the LNL has also demanded a review and upgrade of many of the control systems present in the laboratories, in particularly those that are going to be directly interconnected to SPES, as it is the case of the ALPI superconductive LINAC (Linear Accelerator) control system. These upgrades will allowed an interconnection and interoperability of all the systems using the EPICS CA transport layer. This imply to develop and implement EPICS IOCs, not only for controlling the SPES instrumentation, but also for all the instrumentation present on the LNL accelerator complex. Consequently, a standardization of the hardware and software used to develop all these control systems is fundamental.

In this context, it is evident the need of custom I/O controllers that embed different functionalities to cope with the needs of many subsystems, e.g. tape system, charge breeder, target and ion source, etc. This

need has driven the development of a standard hardware platform for the embedding of the EPICS IOC of all the future control systems. This will help to considerably reduce costs and maintenance efforts.

In this thesis, there will presented contributions to the overall control system architecture of the SPES RIB accelerator. It will be shown the design, development and deployment of the hardware and software for the control of the Target and Ion Source and the Electron Cyclotron Resonance (ECR) ion source subsystem and how they harmonize with the rest of the control systems. It will be detailed how the original design of the IO controller has evolved into a custom version that serves the needs of typical accelerator instrumentation. It will also be shown how the design and implementation of a typical safety-related subsystem (an access control syste) co-exist and blends into the overall architecture of the controls for the LNL accelerator complex.

Chapter 1 gives a general overview of the SPES project at the LNL accelerator complex. On chapter 2 the control system for SPES and its impact on the LNL control system will be described. Chapter 3 and 4 describe in details two kind of I/O controller developed for the EPICS control system of SPES, presenting the result from performance test and how they have been used for the deployment of control systems at the LNL. Finally, on chapter 5, it is presented a safety-related subsystem developed for the LNL accelerator complex and its integration to the overall control system architecture, as part of the SPES project.

Chapter 1

The SPES project at the Legnaro National Laboratories

1.1 Introduction

Nuclear physics studies the properties and stability of the atomic nucleus, where most of the mass of the atom concentrates. The atomic nucleus is positioned in the centre of the atom and it is composed of protons (positively charged particles) and neutrons (neutral particles). Protons and neutrons are both called nucleons and their mass ($\sim 1.66\times10^{-27}$ kg) is approximately 1800 times bigger than electron's one ($\sim 9.11\times10^{-31}$ kg). The stability of the atomic nucleus is guaranteed by the strong interaction, one of the four fundamental interactions of nature (along with electromagnetic force, weak interaction and gravitation) permitting to bind protons and neutrons together contrasting the electrostatic repulsions between positively charged protons. The strong interaction is a very intense force and because of it, a lot of energy is requested to break the atomic nucleus.

The effect of the ratio between number of protons (Z) and neutrons (N) in determining the stability of one particular nucleus is well represented in the nuclide chart shown in Figure 1.1 [1], in which the black squares indicate the stable or extremely long-lived nuclei, forming the so-called "valley of stability". For low Z, stable nuclei are those with N = Z, since in this case the interaction between protons and neutrons it is slightly stronger than the proton-proton and neutron-neutron ones. When the mass number $A \geq 40$ (where A = Z + N), the stability curve diverges from the ideal N = Z line towards the nuclei with N > Z (neutron-rich nuclei), because the electrostatic repulsion between protons tend to prevail over the other stabilizing forces, so that more neutrons – which are non-charged particles not contributing to electrostatic repulsion – are needed to stabilize the nucleus. For extremely high values of A, and consequently very big atomic radius, strong interaction tend to lose its efficacy with respect to electrostatic repulsion, thus drastically limiting the stability of the so-called Super Heavy Elements (SHE).

Nuclei with excess or shortage of neutrons, and consequently far from the "valley of stability", are unstable, radioactive and decay emitting particles (α and β) and γ rays. Unstable nuclei are commonly called "exotic nuclei" and at present approximately 3600 of them can be produced in dedicated facilities

distributed all over the world. However, theoretical calculations predict the existence of more than 6000 unstable nuclei and a great amount of them is still unknown, represented on Figure 1.1 within the borderlines defined by the proton and neutron driplines.

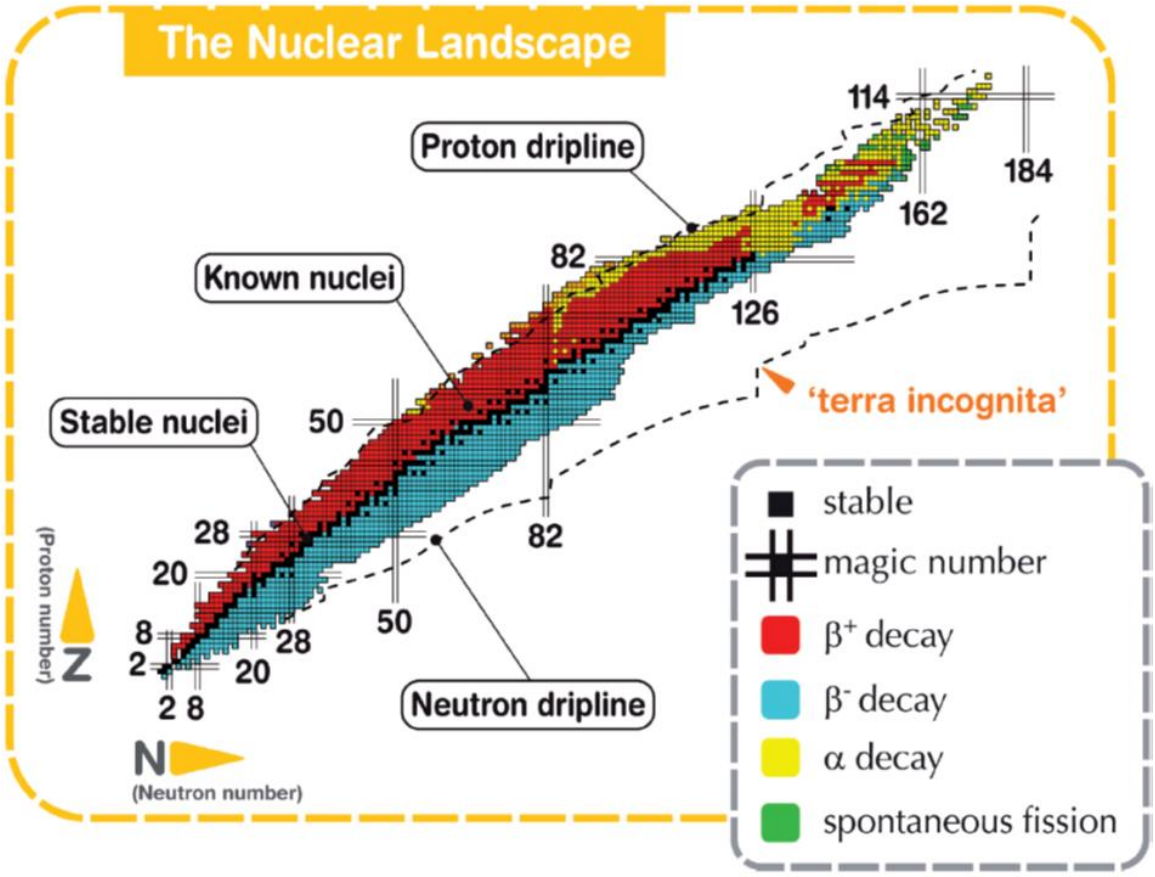


Figure 1.1. Chart of nuclides. [1].

The already known and studied nuclei are indicated on Figure 1.1 in red, blue, yellow, green and most of the unknown nuclei lie in the neutron rich side, on the white area. Theoretical calculations demonstrated that beyond the driplines nuclei have so a high level of instability that start to emit nucleons very quickly in order to reach a sufficient level of stability, entering within the aforementioned borderlines.

Among all the techniques used to investigate the properties of nuclei far from stability, the use of RIBs have been gaining a significant scientific interest due to the large number of species which can be produced and the possibility to use them for studies in different fields of science.

1.2 Radioactive Ion Beams Productions

Nowadays mature techniques for the production of radioactive ion beams make it possible to study nuclei and their properties in an unprecedented way, allowing important improvements in fundamental nuclear physics research and in many applications in various fields of science. Facilities for the production of RIBs allow the detailed study of exotic nuclei, from a nuclear physics point of view but also opening the way to their application in different fields of science.

In a general view, the production of RIBs consists of two very different phases: the generation of the exotic species, obtained usually by means of a nuclear reaction occurring in a well-confined area, and the transport of the produced species to a dedicated experimental area, during which various operations of identification and purification of the desired beam are performed.

With the technical advances and improvements in the layout of modern RIB facilities, the quality of the produced beams is becoming more and more a strategic parameter, especially in the case of very short-lived isotope.

There are two complementary ways to make good quality beams of exotic nuclei: the “in-flight separation technique” and the “isotope separation on line (ISOL) technique”. Both methods transport the nuclei of interest away from the place where they are produced (characterized by a large background due to nuclear reactions) to a well-shielded experimental set-up, where the nuclear properties can be explored. Nuclei transport serves not only to create low background conditions but also to purify the beam and to prepare it in the necessary conditions with respect to energy, time and ion optical properties for the experiments.

1.2.1 The In-Flight Separation technique

The In-Flight Separation technique is based on the nuclear reaction between a primary beam and a lighter production target, in order to produce a secondary beam of radioactive ions, as indicated in Figure 1.2. Usually, the primary beam has energies of several tens of MeV/u (u = mass of a nucleon ~ $1.66 \cdot 10^{-27}$ Kg) so that its interaction with the primary target results in a fragmentation reaction. The produced fragments are then selected using electro-magnetic devices and sent to the experimental areas. In this kind of facility the beam transport system represent a fundamental element: it has to suppress the intense primary beam

transmitted through the primary target and in the same time to guarantee an high quality secondary beam transportation.

This method provides isotopes not very far from stability without limitations due to lifetimes or chemical properties; it allows also for an easy variation of the energy of the reaction products within a certain range and can be implemented in existing heavy-ion accelerators. On the other side the main disadvantages are the low beam quality, due to contamination coming from the primary beam, and the low number of ion species that can be produced.

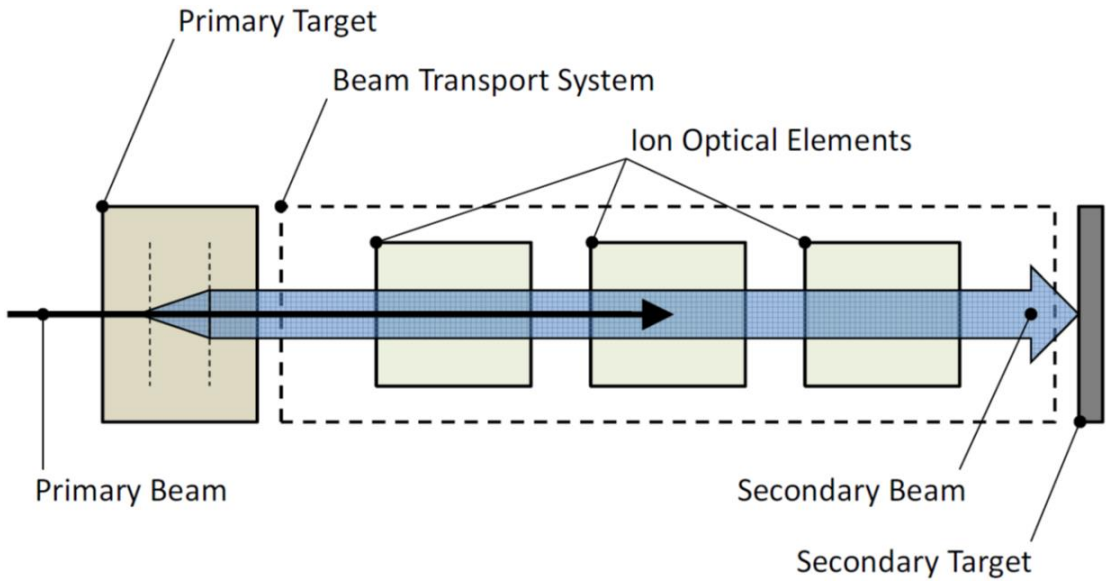


Figure 1.2. Schematic representation of the In-Flight Separation technique. [2].

1.2.2 The Isotope Separation On-Line technique

In an ISOL facility, the radioactive isotopes are produced in a heated target and then transported from the target to an ion source. Once ionized they can be extracted, selected using a dipole magnet, and subsequently accelerated to the required energy. The production sequence is designed to be efficient, fast, selective and highly productive.

In Figure 1.3 a schematic representation of an ISOL facility is reported. It is characterized by the fundamental and characteristic steps of the ISOL technique: production, thermalization, ionization, extraction, mass separation, cooling, charge-state breeding and acceleration. The section of the facility comprehending target, catcher, transfer line and ion source is commonly referred to as “target-ion source system”. Both physical (production cross section, decay – half life, ionization potential, etc.) and chemical

(molecular formation probability, volatility, etc.) properties of the nuclei of interest and of the target materials adopted are important to determine the success of each step in reducing the delay time, which is the average time the radioactive atoms spend from the moment of production to the moment of arrival at the experimental area. In some cases, two or more steps are embedded in a single phase, like for example in the case of production and thermalization. Each single step is described as follows.

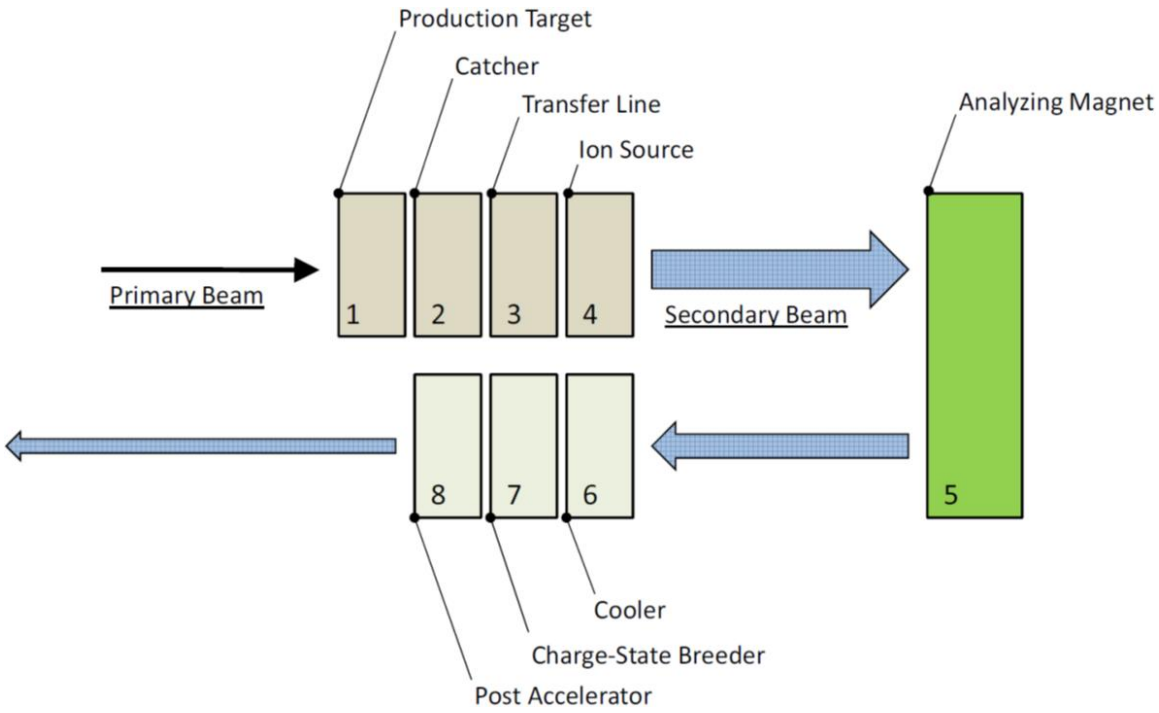


Figure 1.3. Schematic representation of the ISOL technique. [2].

- Production target: A primary beam of light ions, heavy ions, electrons, or neutrons hits a target to produce the desired radioactive ion beams according to different reaction mechanisms. In some cases, a converter is used to transform a primary beam of protons or electrons into a flux of neutrons or γ rays, respectively. This flux is then sent onto the target inducing the nuclear reactions of interest. When choosing the target, it is fundamental to optimize the beam-target combination with respect to the highest production cross-section and the lowest amount of contaminants, and to use target material that is able to stand the highest possible beam currents without damage.
- Catcher: In some cases, the target and the catcher are the same element. The catcher can be solid or gaseous and it is used to stop (thermalize) the exotic nuclei once they are produced by nuclear reactions. After thermalization, a part of the radioactive nuclei of interest is able to escape from

the catcher and to reach the ion source passing through the transfer line described in the following paragraph.

- Transfer line: Through the transfer line, the radioactive atoms are transported from the target/catcher to the ion source by effusion. The transfer line is kept generally at high temperature to avoid loss of radioactive atoms by sticking on its walls.
- Ion source: Depending on the particular requirements, several different ionization mechanisms are used to transform the radioactive atoms into radioactive ions. In general, singly positive or negative charged ions are produced. To allow the acceleration/extraction of the radioactive ions the ion source is held at a positive (in case of beams of positively charged ions) or negative (for negatively charged ions) high voltage.
- Analyzing magnet: Once extracted from the ion source the low energy ion beam is mass separated by an analyzing magnet. Then, it is transported to the focal plan where it can be opportunely focalized. An important property that express the quality of the system is the mass resolving power, defined as $R = M/\Delta M$, where ΔM is the FWHM (full width at half maximum) of a beam of ions with mass M in the focal plane of the separator, which has to be maximized in order to better select the desired isotope.
- Beam cooler: A "cooler" is used to improve the optical properties of the ion beam and to bunch it, in order to increase the peak to background ratio of certain applications like laser spectroscopy or to inject the beam into a charge-state breeder. Two kind of devices are used as coolers: "Penning traps" and "Radio Frequency (RF) coolers". The former is based on the storage of ions using a combination of magnetic and electrical fields; the latter using electrical DC (Direct Current) and RF (Radio Frequency) fields.
- Charge-state breeder: In order to have a simpler and more efficient post-acceleration it is useful to produce a multiple charge state ion beam before the injection into the post-accelerator. A charge-state breeder transforms a singly charged ion beam into a multiple charged one. Two types of charge-state breeding ion sources are used: the Electron Beam Ion Source (EBIS) and the Electron Cyclotron Resonance (ECR) ion source. Both are based on an intense bombardment of the beam ions with energetic electrons, with electron impact ionization [3] yielding ions in higher charge states. The plasma of ions and electrons is confined through electrical and strong magnetic fields

- Post accelerator: The highly charged ion beam from the charge-state breeder or the beam of singly charged ions is then injected into the post-accelerator. After the post-acceleration step, the beam is sent to the experimental halls.

1.3 The SPES Project at LNL

The so-called SPES project [4] is intended to develop a radioactive ion beam facility at National Institute of Nuclear Physics (Legnaro, Padua, Italy). In particular, the SPES facility will produce neutron rich nuclei with mass in the range 80-160 [5] [6], to use for forefront research in nuclear physics and for many applications in different fields of science.

The facility proposed for the SPES project has two main goals: to provide a radioactive ion beam accelerator system to perform forefront research in nuclear physics by studying nuclei far from stability, and to develop an accelerator based interdisciplinary research center. The SPES facility will be mainly devoted to the production of neutron-rich radioactive nuclei with mass number in the range 80-160 by the 238U fission at a rate of 10^{13} fission/s. The production of few selected proton-rich isotopes will be carried out as well. An overview of the SPES facility is presented in Figure 1.4. This facility is currently been constructed at LNL, as it can be seen on Figure 1.5.

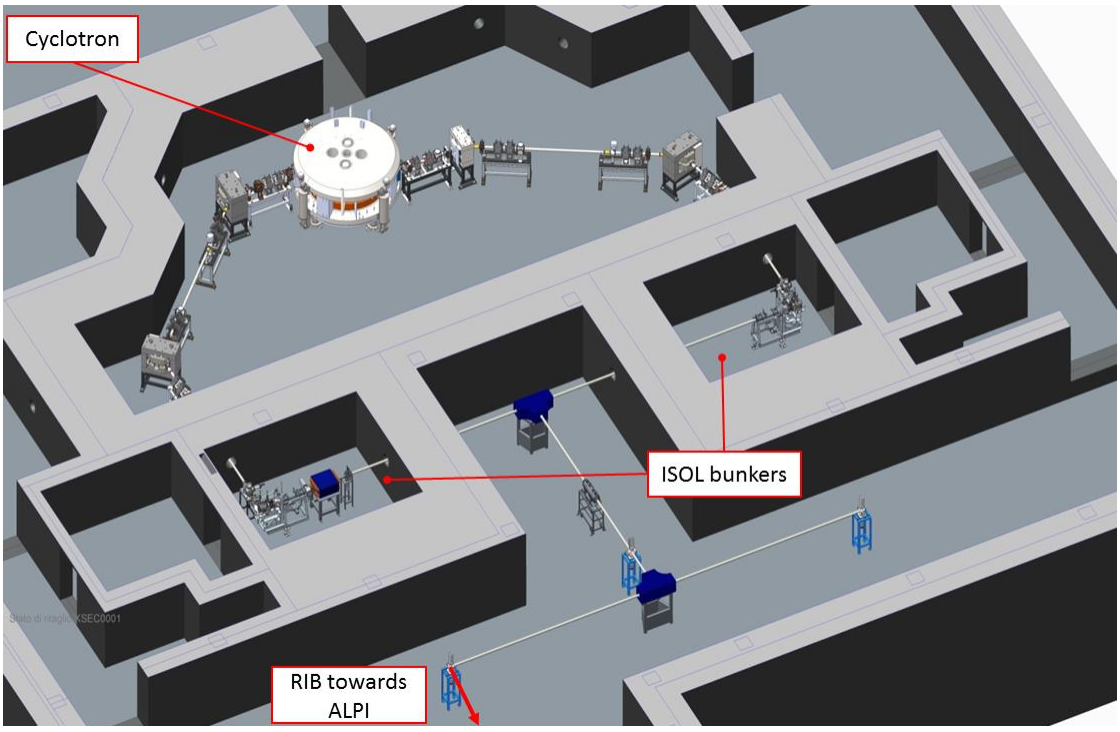


Figure 1.4. General overview of the SPES facility.

Figure 1.5. Status of the construction of the SPES facility at LNL.

Neutron-rich ion beams will be produced according to the ISOL technique, using the proton induced fission on a direct target of uranium carbide, while other types of targets and nuclear reactions will be exploited to produce proton-rich species. The primary beam (proton beam) is furnished by a Cyclotron with variable energy (15-70 MeV) and a maximum current of 0.750 mA upgradeable to 1.5 mA and splitted on two exit ports.

The SPES facility is designed to supply a second generation of exotic beams able to perform a step toward EURISOL (European ISOL Facility) [1] and to offer a powerful accelerator based system for research in applied physics, nuclear astrophysics, solid-state physics and nuclear medicine.

The most critical element of the SPES project is the production target; it represents an innovation in terms of capability to sustain the primary beam power. The design is carefully oriented to optimize cooling by thermal radiation taking advantage of the high operating temperature of approximately 2000°C. An extensive simulation activity has been performed to study the target thermal behaviour [7] and its release properties. The production target and all the experimental apparatus needed to guarantee its functioning

will be designed following the ISOLDE (CERN, Switzerland), the HRIBF (ORNL, USA) and the EXCYT (INFN, LNS, Italy) projects and special care will be devoted to the safety and radioprotection of the system. According to the estimated level of activation in the production area a special infrastructure will be designed; the use of up-to-date techniques of nuclear engineering will result in a high safety level of the installation.

The isotopes will be extracted and ionized at +1 charge state with an ion source connected with the production target by means of an opportunely designed transfer line. Different kinds of ion sources will be used to obtain all the radioactive ion beams of interest.

The transport and the selection of the exotic beam at low energy and low intensity is a challenging task. A cooler and a high resolution analyzing magnet will be placed in series along the beam line after the extraction of the ions from the ion source to improve the optical quality of the beam and to perform an accurate mass selection, respectively.

To optimize the post acceleration, a charge breeder will be introduced: it will increase the charge state to +N before injecting the exotic beam in the superconductive RFQ which represents the first reacceleration stage before the injection into the linear accelerator ALPI. The proper velocity matching needed to enter RFQ will be furnished by High Voltage platforms operating approximately at 250 kV. The expected beam on experimental targets will have a rate on the order of 10^8-10^9 pps (particles per second) for ^{132}Sn, ^{90}Kr, ^{94}Kr and 10^7-10^8 pps for ^{134}Sn, ^{95}Kr with energies of 9-13 MeV/u.

Figure 1.6 shows all the components of the SPES project and their integration into the LNL accelerator complex.

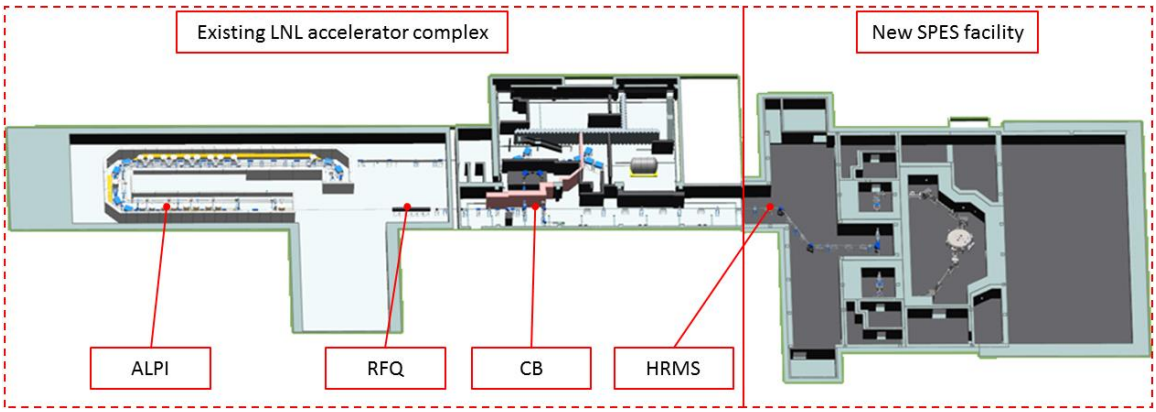


Figure 1.6. The SPES project integration with the LNL accelerator complex.

1.4 The Off-Line SPES Laboratory at LNL

The apparatus known as the SPES Front-End, it is one of the main elements of the project. It is the core of the production area, holding the target into the primary and secondary beam lines. It can be seem as a converter, taking in the primary proton beam from the cyclotron, and delivering the radioactive ion beam.

An off-line (i.e. without the presence of the primary proton beam) version of the Front-End apparatus have been constructed at the LNL. It has been in operation for the past five years, and it serves as a research and development test bench for engineers of different fields. New devices and techniques developed on this laboratory will be later be implemented on the final SPES facility.

With this off-line Front-End it is possible to accelerate stable +1 ion up to 30keV. It is divided into four functional subsystems, as showed on Figure 1.7. The first one is the target and ion source complex where the ion beam is produced and extracted. The second one is the beam optics subsystem, where the direction and focalization properties of the beam are manipulated using a four electrostatic steerers and one electrostatic quadrupole triplets. The third subsystem is the mass separator, where the produced ion beam is filtered into a certain ion mass region. Finally, the last subsystem is the beam diagnostic, used to measure the relevant beam information like the beam shape and intensity, using beam profile monitors, faray cups and an emittance meter.

1.5 Conclusions

In this chapter, a general overview of the SPES project was presented. RIB production techniques were reported. A detailed description of the main components constituting an ISOL facility was furnished and general presentation of the main components of the SPES facility were reported. The SPES off-line laboratory in operation at the LNL was described.

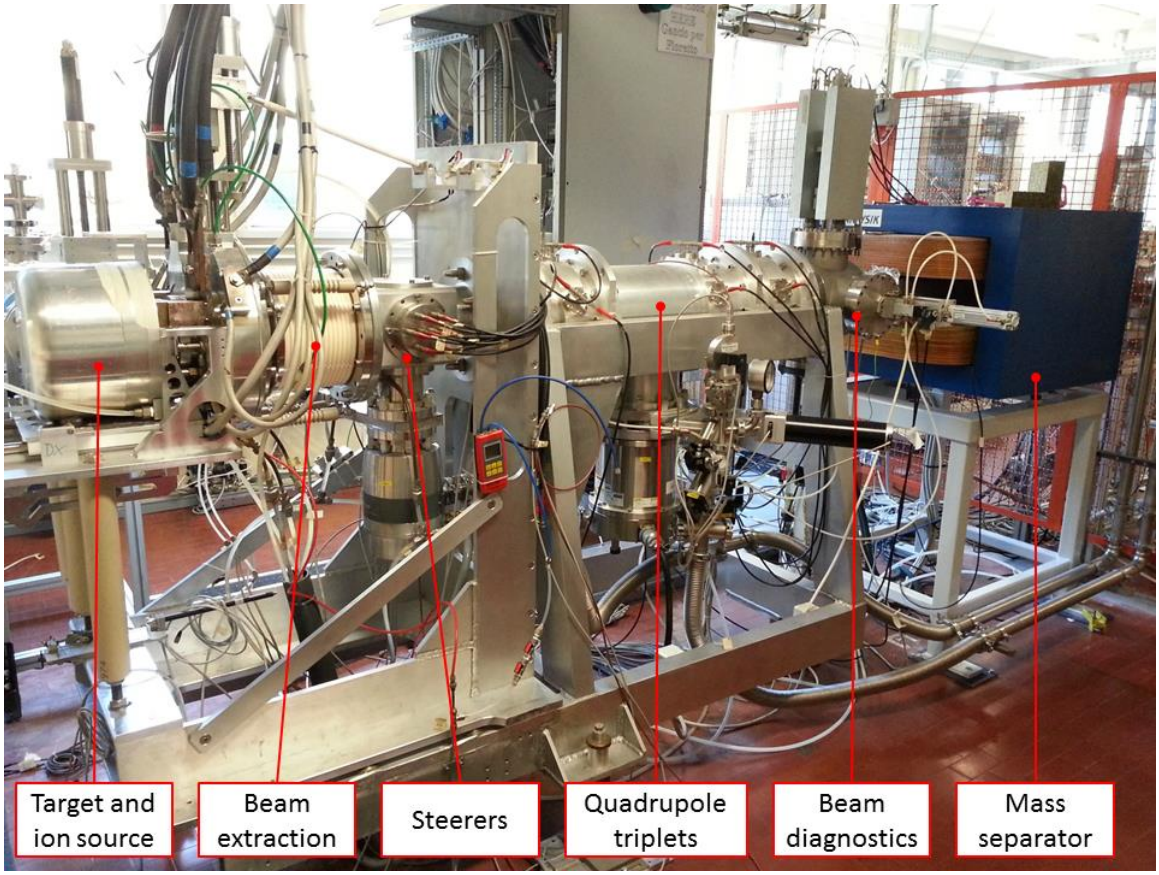


Figure 1.7. The SPES off-line front end apparatus installed at LNL.

Chapter 2

The Control System for the SPES Project

2.1 Introduction

Designing the control system for a project as SPES is a very complex, as well as time and resource consuming mission. A large number of subsystem must interoperate in order to be able to manage a facility of this magnitude. Moreover, the subsystem are very heterogeneous, going from beam diagnostics devices until safety personal protection systems.

EPICS was chosen as the main framework for the control system. It is use on many physics facilities around the world, and it supported by a big community. On the general architecture, some subsystem will be controlled using native EPICS controllers, while others will be controlled using PLCs. EPICS will also serve as a main supervisor network of all the control system.

On the next chapter, an introduction to EPICS will be presented. Then, the control system for SPES will be described.

2.2 Experimental Physics and Industrial Control System (EPICS)

EPICS is a set of software tools and applications which provide a software infrastructure for building distributed control systems, usually used on system such as particle accelerators, large experiments and major telescopes. Such distributed control systems typically comprise tens or even hundreds of computers, networked together to allow communication between them and to provide control and feedback of the various parts of the device from a central control room, or even remotely over the internet.

It uses Client/Server and Publish/Subscribe techniques for communication between the various computers. Most servers (called Input/Output Controllers or IOCs) perform real-world I/O and local control tasks, and publish this information to clients using the Channel Access (CA) network protocol. CA is specially designed for the kind of high bandwidth, soft real-time networking applications that EPICS is

used for, and is one of the reasons that allows it to be used for building a control system comprising hundreds of computers.

Figure 2.1 shows the generic architecture of EPICS control system. The CA clients are programs that required access to process values (PVs) to carry out certain purpose. The service that a CA server provides is access to a PV. The mean of communication is the CA protocol over an Ethernet network.

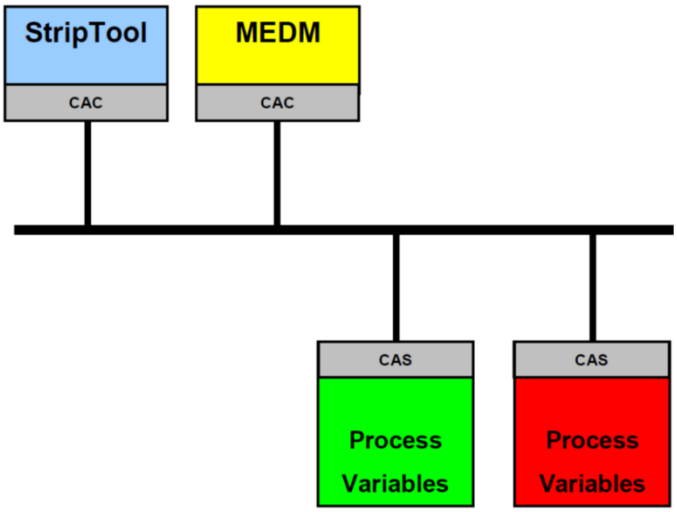


Figure 2.1. The EPICS control system architecture.

The procedure for searching variables starts by the client, broadcasting a UDP request package with the name of the PV, looking for the server in which they exist. Then, the server respond and a TCP connection is stablished for sending the requested information. Figure 2.2 illustrates this procedure. In addition, the clients can set monitors on PVs and it will then be notified when values changes.

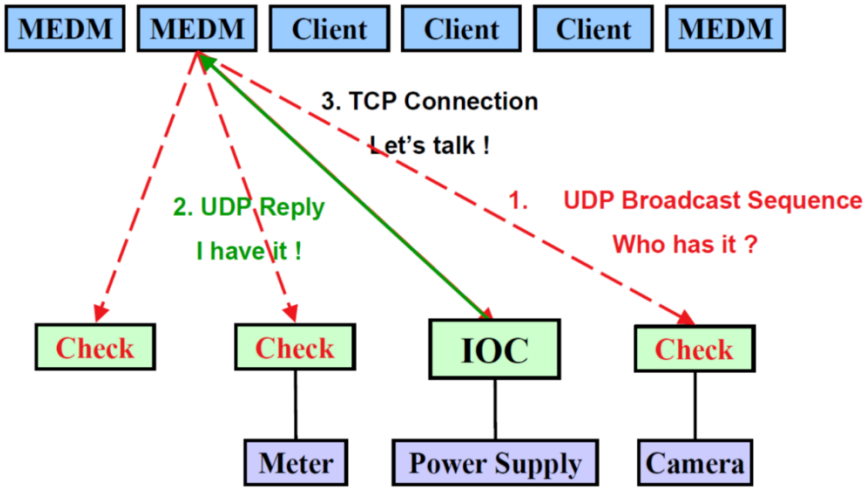


Figure 2.2. EPICS search and connect procedure.

In EPICS, a PV is a named piece of data associated with the machine (e.g. status, setpoint, parameter), which has a set of attributes (alarm status and severity, timestamp, normal operation range, control limits, engineering unit designators).

An EPICS record (see Figure 2.3 for a graphical representation) is an object with a unique name, a behavior that it is defined by its record type, controllable properties, optional association with hardware I/O, and links to other records. Its fields define when to process, where to get/put the data from/to, how to turn raw data into numeric engineering value, limits indicating when to report an alarm, a process algorithm, etc. The field are also used for holding the input or output values, the alarm status, timestamp, etc. Records have some functionality associated with them, as for example scaling, filtering, alarm detection, calculation, etc. The data within a record is accessible via PVs. A collection of record is called a database, and it is load by the IOC.

An EPICS IOC have two primary application specific components: the real-time database of records, and state notation language programs used to implement state oriented programs. The machine status, information and control parameters are defined as records in the application specific database.

An IOC runs a set of EPICS routines called iocCore used to define PVs and implement real-time control algorithms. It uses database records to define PV and their behavior.

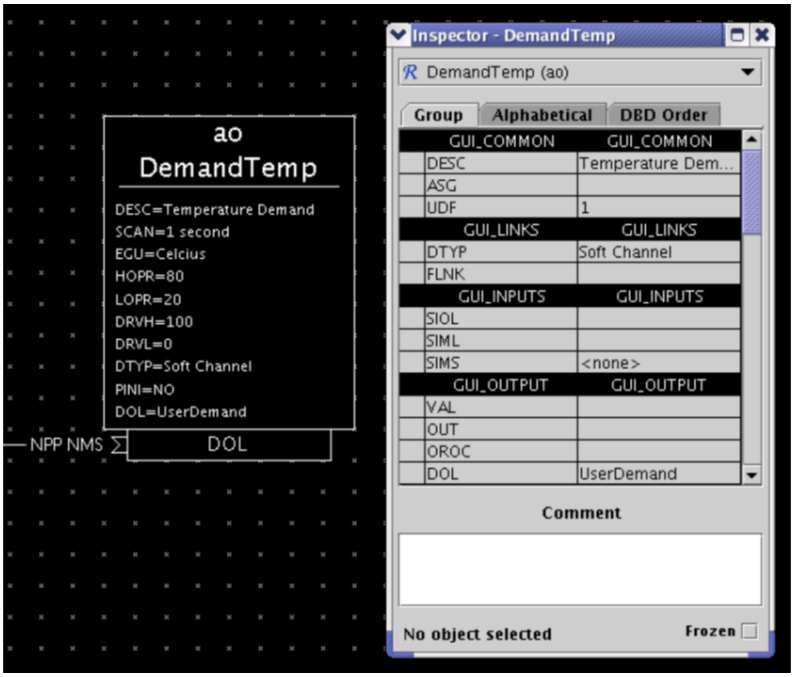


Figure 2.3. Graphci view of an EPICS record.

As its name implies, an IOC often performs input/output operations to attached hardware devices. The IOC associates the values of EPICS PVs with the results of these I/O operations. It can perform sequencing, closed loop control and other computations. A set of routines known as “devices support” are required in order to perform this I/O operation with the instruments, interfacing the hardware and the EPICS record.

EPICS tools are available to accomplish almost any typical distributed control system functionality, such as remote control, data conversion and filtering, closed loop control, alarm detecting and logging, data trending and archiving, automatic sequencing, configuration control, data acquisition, data analysis, among many others.

Figure 2.4 presents an overview of the software components and layer on a EPICS IOC.

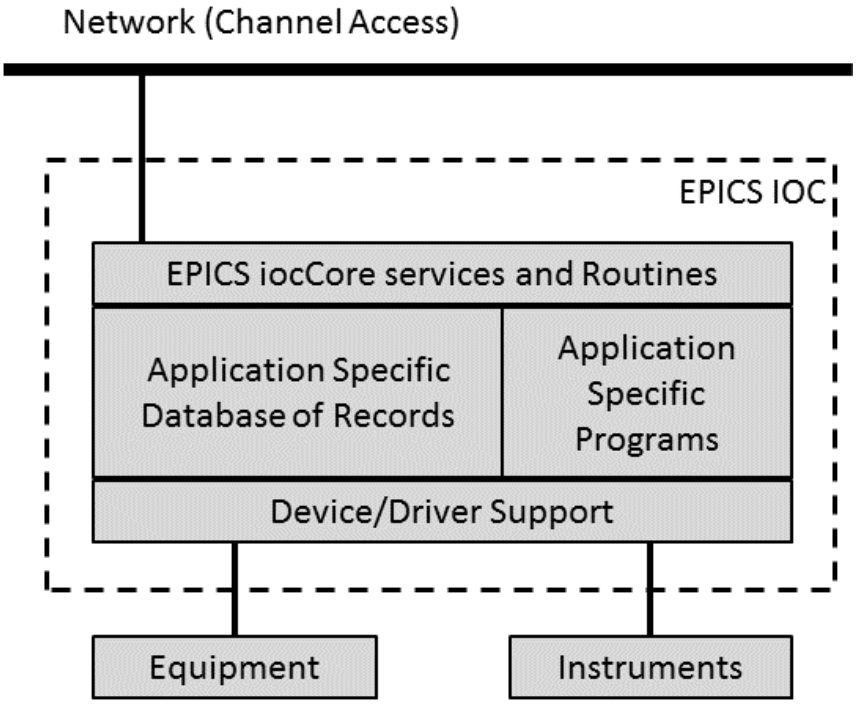


Figure 2.4. EPICS IOC software components.

From the hardware point of view, IOCs can be an embedded microntroller, a rack-mounted server, a laptop or desktop PC or MAC, or a single board computer. Usually, this system has installed a variety of modules (GPIB, RS-232, DIO or AIO cards, etc.) which interface to control system instruments (oscilloscopes, network analyzers, power supplies, etc.) and devices (motors, thermocouples, switches, etc.). It may be running on GNU/Linux, MS Windows, MacOS, Solaris, Darwin, RTEMS, HP-UX or vxWorks.

Among the EPICS CA clients, one of the most widely used is Control System Studio (CSS) [8]. It is an Eclipse-based collection of tools to monitor and operate large scale control systems, such as the ones in the accelerator community. Figure 2.5 shows an example of an Graphical User Interface (GUI) developed using CSS for an EPICS control system.

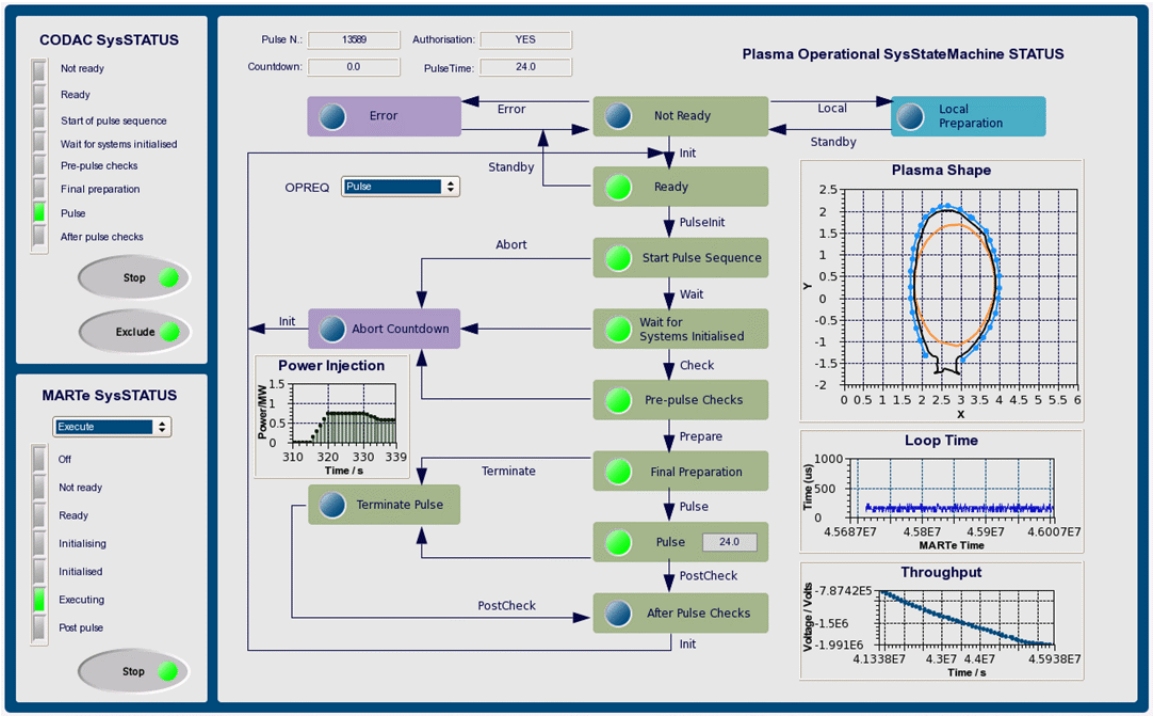


Figure 2.5. An example of an EPICS GUI developed using CSS.

2.3 The SPES Control System

For the SPES project, EPICS was chosen as framework for the development of the control system. The system will be divided into two main areas: control system based on native EPICS IOCs, and control system based on PLCs. A description will be presented on the following chapters.

The EPICS CA will be used as the main control network, where all control systems will be connected, for sharing information allowing their interoperability. The main control network will be subdivided into Virtual Private Networks (VPNs), one for each subsystem, in order to separate the broadcast domains of each subsystem sub-network. An EPICS CA gateway will allow the communication between different VPNs.

On the main control network, basic EPICS services will be deployed, such as data archiving, graphical user interfaces, autotuning applications, etc.; as well as basic network services as NTP, DNS, DHCP, firewalls, among others.

Figure 2.6 presents the network architecture of the SPES EPICS based control network.

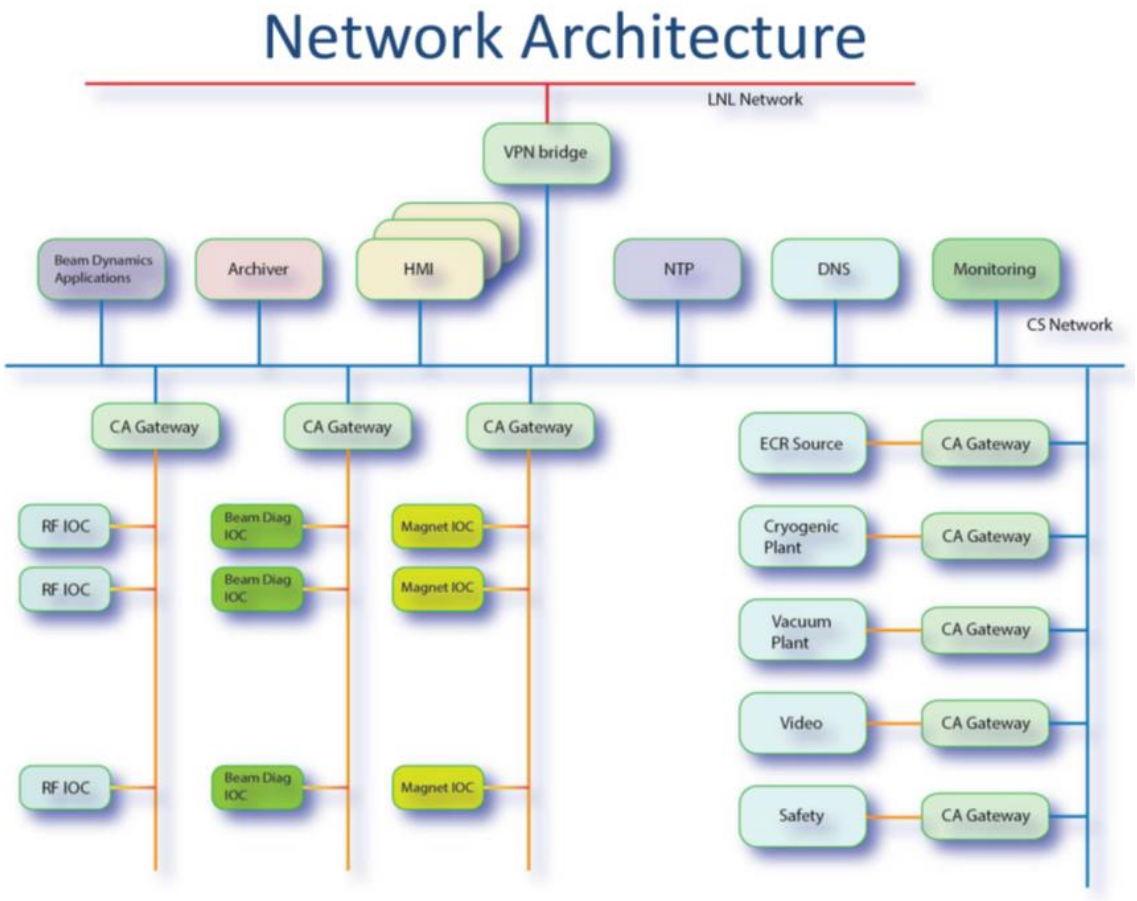


Figure 2.6. The SPES control network architecture.

2.3.1 EPICS based control systems

The general architecture for the EPICS based control system consists on a series of distributed IOCs for controlling all the accelerator instrumentations.

The subsystem that will be controlled using native EPICS IOCs are the Radio-Frequency (RF) controllers, the beam diagnostic, the beam transport devices, the target and ion source, ECR source, the charge breeder, among others.

2.3.2 PLC based control systems

For the subsystems where a high reliability is necessary, commercial PLC will be used for developing the control system. Some examples of these subsystems are machine and personal protection system, vacuum, ventilation, water cooling, cryogenic plant, among other control systems.

Although PLCs will directly control these systems, they will also be interconnected to the EPICS CA network, mainly with read-only privileges. For this, interface devices will be used which will read the information for the PLCs and make them available into the EPICS network, through PVs.

2.3.3 Impact on the LNL accelerator complex control system

The arrival of the SPES project to the LNL has also demanded a review and upgrade of many of the control system present in the laboratories, in particularly those that are going to be directly interconnected to SPES, as it is the case of the ALPI superconductive LINAC. These upgrades is necessary in order to allow the interconnection and interoperability of all the systems using the EPICS CA transport layer. This implies to used EPICS IOCs, not only for the SPES instrumentation, but also for all the instrumentation present on the LNL accelerator complex. Consequently, a standardization of the hardware and software used to develop all these control systems is required.

In this context, it is evident the need of custom I/O controllers that embed different functionalities to cope with the needs of many subsystems, e.g. tape system, charge breeder, target and ion source, etc. This need has driven the development of a standard hardware platform for the embedding of the EPICS IOC of all the future control systems. This will help to considerably reduce costs and maintenance efforts.

2.4 Conclusions

In this chapter, a general description of the basic concept of EPICS was presented. An overview of the SPES control system was described. It was reported its division into EPICS-based and PLC-based, and a description of each subsystem was presented. Finally, it was also illustrate the effect that SPES have had into the LNL control systems and its implication in the future development of both systems.

Chapter 3

EPICS IOCs for the off-line laboratory

3.1 Introduction

At LNL, for the past four years it has been in operation an off-line laboratory for the SPES project. In this laboratory, the SPES target front-end apparatus have been tested. The main purpose of this laboratory is to serve as a test bench for new instrumentation, detectors, and control systems.

The control system of this laboratory is in charge of managing the front-end apparatus. It takes care of managing the beam production, beam transport and beam diagnostic. Each one of these subsystems, comprises a series of instrument devices as, for example, power supplies, detectors, stepper motors, among others.

From early stages of the SPES project, EPICS [9] was chosen as the standard framework for developing the control system. Consequently, a variety of commercial EPICS IOCs (Input/Output Controllers) have been used on the control system of the off-line front-end [10] [11].

For this purpose, a new kind of IOC was developed at LNL and tested on the off-line laboratory. It is based on the computer board Raspberry Pi [12] with custom-made expansion boards. These IOCs represent a flexible, easy to adapt, low cost and open solution for the control system of this laboratory.

On this chapter, this IOC is presented. Both hardware and software developments will be described in details, together with performance results. There will be described also the control systems implemented using this IOC on the SPES off-line laboratory

3.2 IOC Description

The IOC is intended as a low cost and adaptable solution for developing the control system of the SPES off-line laboratory using EPICS. The core of the device is the computer board Raspberry Pi, which has the most common components of a PC, such as core CPU, RAM memory and standard IO interfaces (USB, Ethernet, Audio, Video, GPIO, among others).

In order to provide the necessary additional IO capabilities to the control system, not usually found on standard PCs, custom expansion boards were developed.

The Raspberry Pi runs an operating system (flavor of Linux) called Raspbian [13]. On top of it, interface drivers for communication with the expansion boards, combined with EPICS applications and tools were built.

3.2.1 Hardware Architecture

The IOC was developed using the computer board Raspberry Pi (Model B, rev. 2) as a core (Figure 3.1). This computer board uses the Broadcom BCM2835 System on Chip (SoC), which contains an ARM1176JZFS application processor running at 700 Mhz (although it can be overclocked to 900 Mhz), a Videocore 4 multimedia co-processor, a OpenGL-ES 1.1/2.0 GPU, and 512 MB of RAM. The LAN9512 USB 2.0 HUB and 10/100 Ethernet Controller is connected to the SoC in order to provide two USB 20.0 ports and an Ethernet port to the Raspberry Pi. Furthermore, low-level General Purpose Input/Output (GPIO) and a SD card port as available from the SoC. Figure 3.2 shows a simplify block diagram of the Raspberry Pi.

Figure 3.1. Raspberry Pi computer board. (model B, revision 2).

The system boots from an external SD card, which holds the operation systems and acts as long-term storage. The board is powered with 5VDC and consumes around 3.5W.

In order to provide data acquisition and instrumentation control capabilities to this board, tailored expansion boards where developed at LNL. Four boards were designed: a 1-channel, current-signal

acquisition board; a 40-channel, current-signal acquisition board; a stepper motor controller board, and a general purpose IO board. These boards were connected to the Raspberry Pi through its General Purpose I/O port.

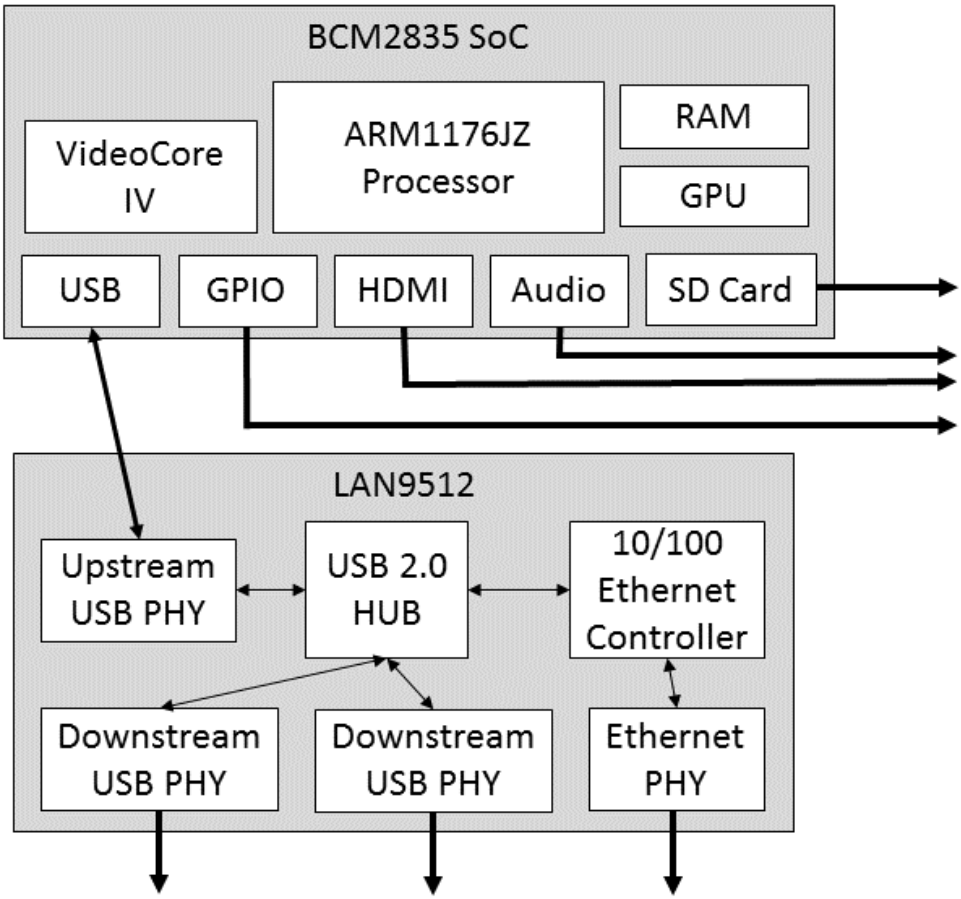


Figure 3.2. Raspberry Pi block diagram. (Model B, revision 2).

For applications where additional standard IO interfaces were needed (for example, serial and Ethernet ports), commercial USB converter were used. These additional ports are automatically managed by the operating system and, therefore, no interface drivers were needed for using them.

On the following chapters, each one of the expansion board will be described.

3.2.1.1 1-Channel, current signal acquisition board

Many of the beam diagnostic detectors at LNL provide current signals to the acquisition systems. Usually, the dynamic range of these current signals is very wide, going from the pA until the mA range. For this reason, having an acquisition board capable of directly accepting this kind of current signals on the control

system is of great advantage, eliminating the need of a pre-amplification, conversion and signal conditioning stage.

The board is formed by a first stage were the current signal is amplified and converted to a voltage signal, followed by an analog to digital conversion stage.

For the current to voltage conversion stage, the LOG112 amplifier [14] was used. This integrated circuit computes the logarithm ratio of an input current relative to a reference current. The output signal is trimmed to 0.5 V per decade of input current. Furthermore, an operational amplifier is available internally on this device, which can be used for scaling the output signal. Figure 3.3 shows a functional diagram of this device.

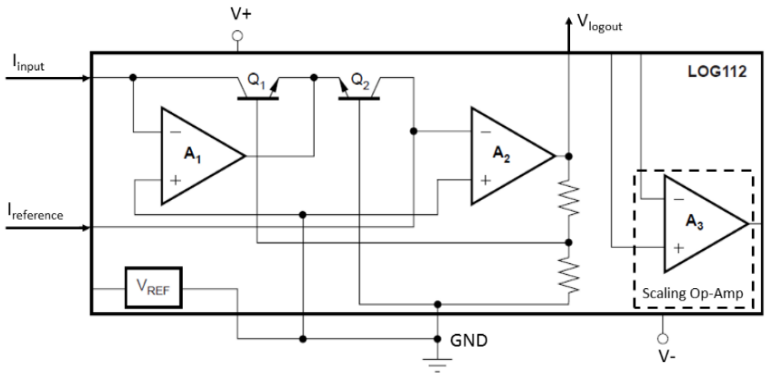


Figure 3.3. LOG112 functional block diagram. [14].

The output signal of this device is expressed by

$$V_{out} = A_{scaling} \cdot 0.5 \cdot log\left(\frac{I_{input}}{I_{reference}}\right) \quad (3.1)$$

Where, I_{input} represents the input current, $I_{reference}$ represents the reference current, and $A_{scaling}$ represents the gain set on the scaling operation amplifier stage.

On the final acquisition board, a 124 nA current signal, generated using a temperature compensated current source, is used as reference. The external input current is directly connected to the current input of the LOG112. The scaling operation amplifier is used with a gain of 4.1. Hence, the output signal of this convertion stage, as defined in (3.1), is expressed by

$$V(I_{input}) = 2.05 \cdot log\left(\frac{I_{input}}{124 \cdot 10^{-9}}\right) \quad (3.2)$$

Where, I_{input} represents the input current.

The LOG112 accepts current from 100pA until 3.5mA. This current range produce an output signal, according to equation (3.2), of approximately -6.3 V and 9.1 V, respectively.

On the software side, the acquired value will be converted back to the corresponding input current value using an exponential function, thus actually inverting equation (3.2).

For the analog to digital conversion, the ADS8509 ADC [15] was used. This converter contains a complete 16-bit, capacitor-based, successive approximation register (SAR) A/D converter with sample-and-hold, reference, clock, and a Serial Peripheral Interface (SPI) compatible serial data interface. The input signal range can be set to different values, using an external resistive network. Figure 3.4 shows a functional block diagram of this device.

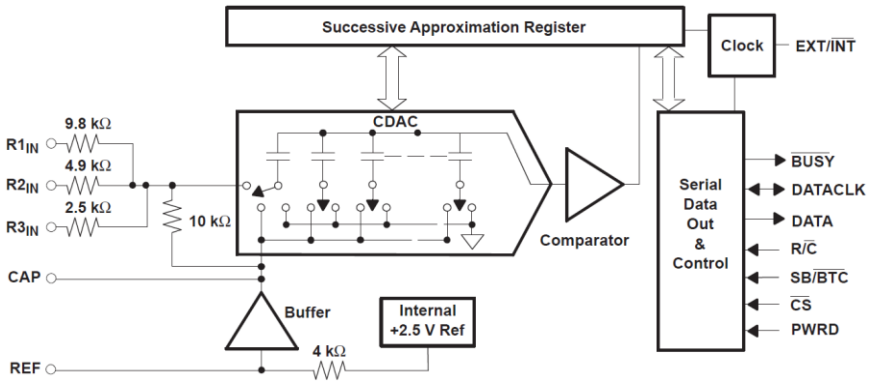


Figure 3.4. ADS8509 functional block diagram. [15].

On the final acquisition board, the range of this ADC is set to +/-10 V. The output of the previous current-to-voltage conversion stage is connected to the input of the ADC. Lastly, the SPI bus is connected to the Raspberry Pi GPIO interface for data acquisition.

Figure 3.5 shows the circuit schematic of the final acquisition board, while Figure 3.6 shows a picture of the final board.

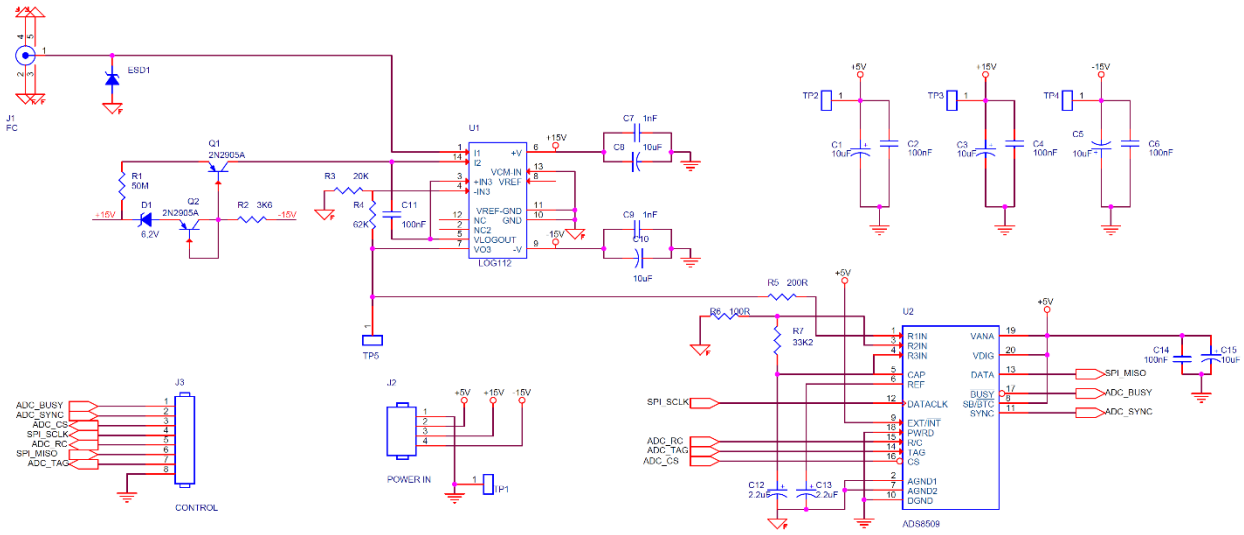


Figure 3.5. Circuit schematics of the 1-channel, current input acquisition board.

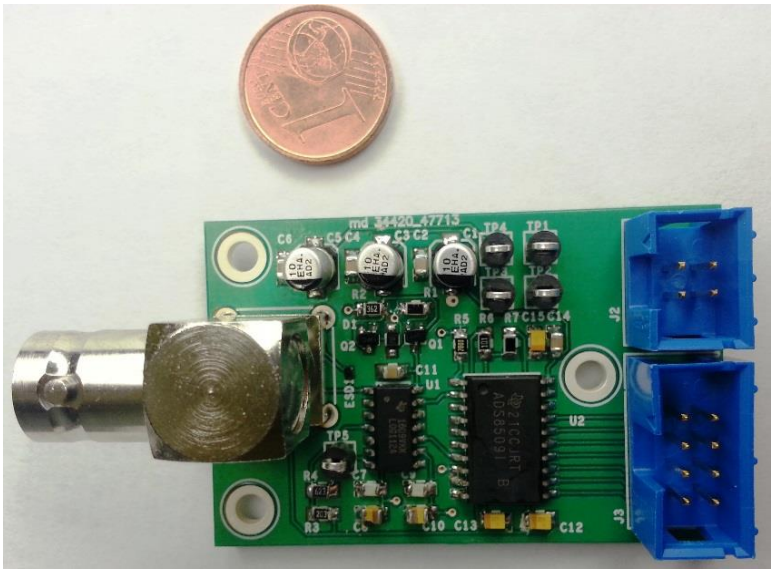


Figure 3.6. 1-channel, current signal acquisition board.

3.2.1.2 40-Channel, current signal acquisition board

As previously explained for the 1-channel current-signal acquisition board, current signals with large dynamic ranges are present on many of the beam diagnostic systems. Typical beam diagnostic devices (Beam Profilers) frequently used at LNL generate 40 of these signals. Beam profilers are basically grids of 6 microns wires laid on X-Y planes traverse to the beam propagation direction. Charges impinging the wires are sensed by charge amplifiers that drive 20 m cables up to the acquisition boards. Hence, having an acquisition board capable of directly accepting 40 of these kind of current signals on the control system

is of great advantage, eliminating the need of a pre-amplification, conversion and signal conditioning stage.

For this reason, an expansion board with 40-channel current-signal acquisition capabilities was designed. The operational principle of this board is similar to the previous 1-channel version. The first stage is formed by 40 parallel current to voltage converters, followed by an analog multiplexing stage, and finally an analog to digital conversion stage.

Each one of the 40 current to voltage converter is identical to one used on the previous 1-channel version, based on the LOG112 amplifier [14] showed on Figure 3.3. In the same way, it is used a 124 nA current signal as reference, generated using a temperature compensate current source. The output signal is expressed by equation (3.2). Figure 3.7 presents the circuit schematic of one of these 40 converters.

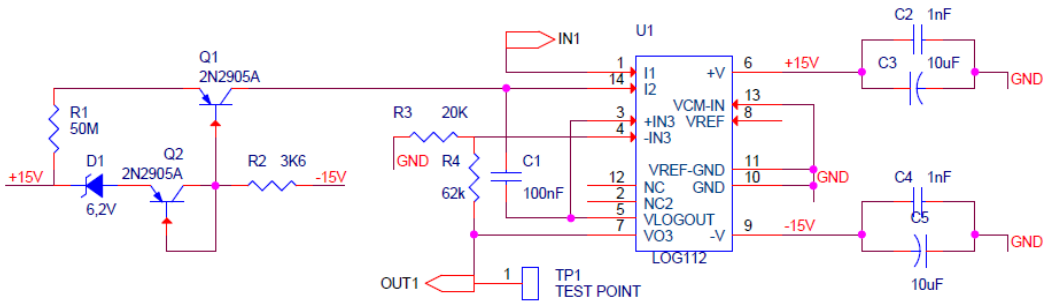


Figure 3.7. Current to voltage converter used on the 40-channel, current signal acquisition board.

The 40 resulting output signals are multiplexed using six 8-to-1 analog multiplexer (DG408 [16]). First, five multiplexers, working in parallel, multiplex the 40 signals into five. These five signals are then in turn multiplexed into one signal using the sixth multiplexer. A simplify circuit schematic of this stage is presented on Figure 3.8.

Finally, the resulting analog signal is digitalized using the same ADC showed on Figure 3.4. The ADC input range is configured to +/-10 V as before. The circuit schematic of this stage is showed on Figure 3.9.

The Raspberry Pi GPIO interface is connected to the SPI bus of the ADC and to the multiplexer control signals. Figure 3.10 shows a picture of the final board.

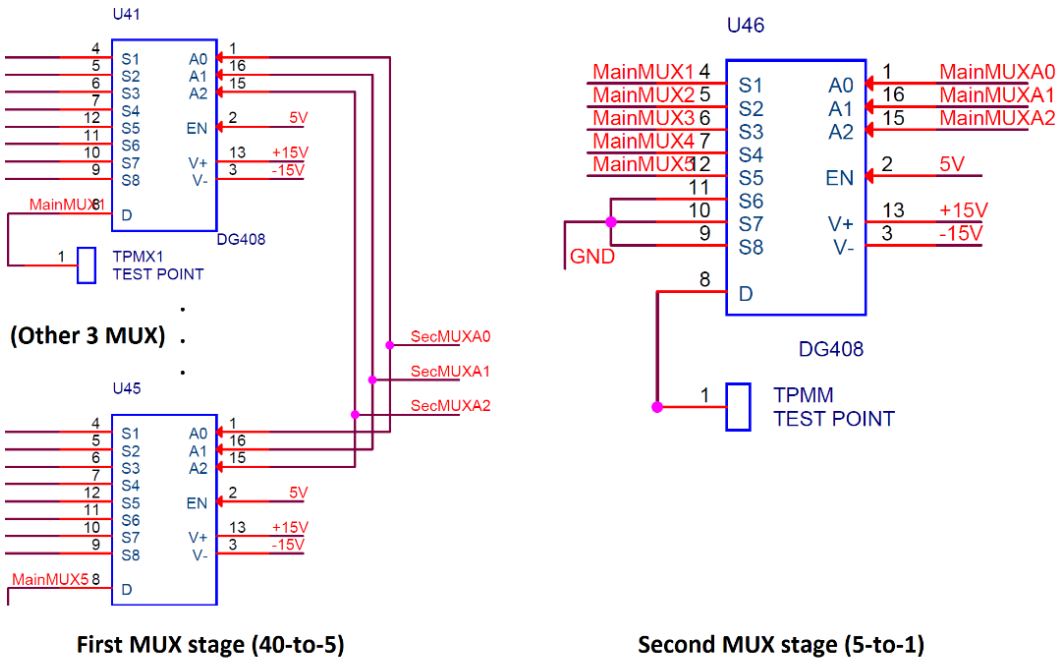


Figure 3.8. Analog multiplexing stage used on the 40-channel, current signal acquisition board. (On the first stage [left], only two of the five multiplexer are shown. Those five signal are then passed to the multiplexer on the second stage [right]).

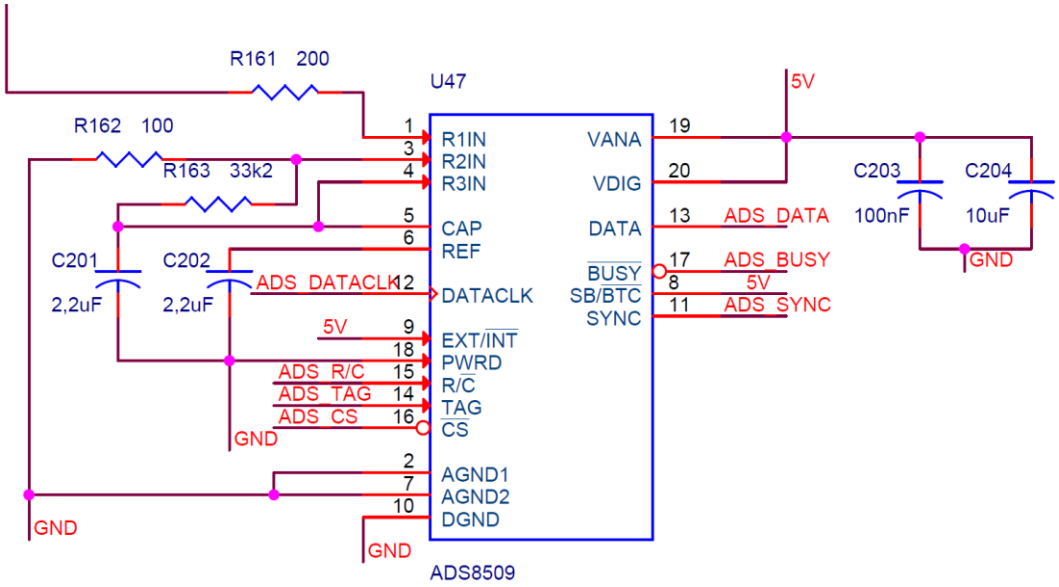


Figure 3.9. A/D conversion stage used on the 40-channel, current signal acquisition board.

Figure 3.10. 40-Channel, current signal acquisition board.

3.2.1.3 Steeper motor controller board

Steeper motor are widely used at LNL for controlling the position of detectors, slits, proportional valves, among others. For this reason, it was decided to design a stepper motor controller expansion board for these IOCs.

The core of the board is a microcontroller that is in charge of controlling the current on both phases of the stepper motor using two full-bridge power amplifiers.

The full-bridge power amplifier used is the LMD18245 [17]. This device incorporates all the circuit blocks required to driver the motor windings, using CMOS control and protection systems with DMOS power switches on the same monolithic structure. The power stage is an H-bridge that delivers continuous output current to the motor windings, up to 3 A and 55 V. A current sensing amplifier eliminates the power loss associate to a sense resistor in series with the motor, as found on some controllers. A 4-bits DAC provides a digital way of controlling the winding currents, simplifying the implementation of micro-stepping control techniques. Figure 3.11 shows a functional diagram of this power amplifier.

The microcontroller PIC18F27J13 [18] was used. This microcontroller manages two LDM18245 for controlling bipolar stepper motor using the micro-stepping technique, hence reducing mechanical noise and resonance usually found on previous controllers that use full-steps, as well as producing a gentler mechanical actuation. Many articles have been published about similar implementations [19] [20] [21].

However, in this case, the LDM18245 was used in order to simplify the control algorithm on the microcontroller, due the fact that it only control the DAC stage on the amplifiers, leaving the close loop part to the driver.

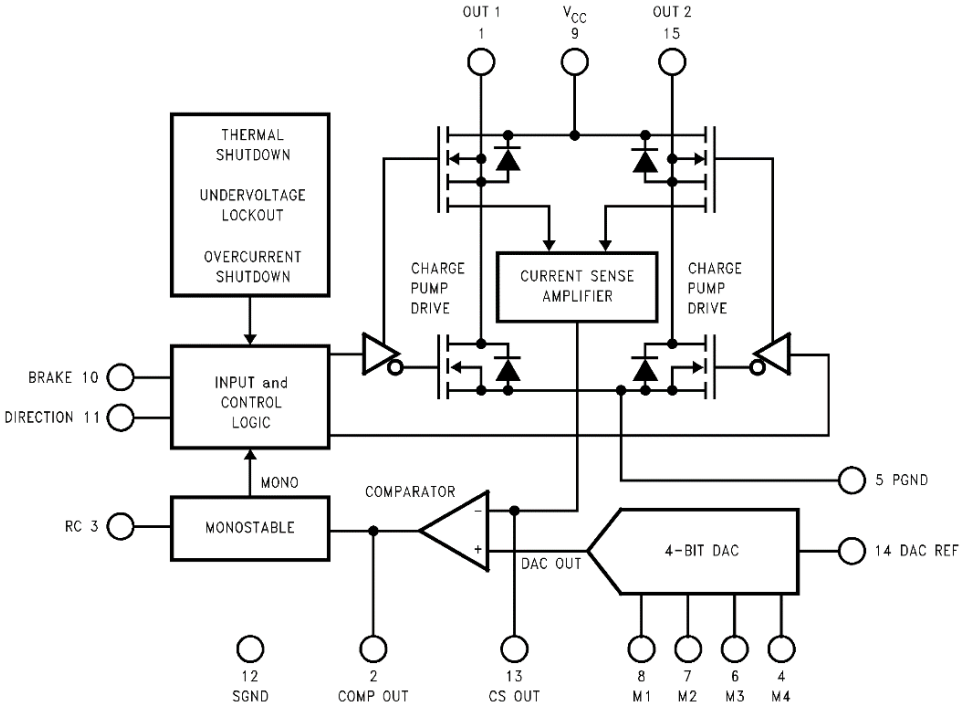


Figure 3.11. LDM18245 functional block diagram. (Taken from [17]).

The microcontroller, in order to perform micro-stepping current controls, must just manage the DAC's four inputs signals, in conjunction with the direction and brake control signals, of each power amplifier. On the other hand, the microcontroller have three control inputs, used by the Raspberry Pi. These signals are: a "direction" signal for selecting the sense of rotation of the stepper motor, a "step" signal that produce a step on the motor each time a pulse is detected, and a "chip select" signal used for controlling more than one controllers connected to the same control signals.

Furthermore, two limit switch inputs are available on the microcontroller. They are intended for reading two mechanical end of travel switches. When one of these switches is activated, the microcontroller automatically stops the movement of the motor on that direction, avoiding possible damage on the mechanical components. This two status bits are also forward to the Raspberry Pi.

On Figure 3.12 and Figure 3.13 are presented, respectively, the circuit schematic and final board of this stepper motor controller. The source code of the microcontroller program is reported on the appendixes.

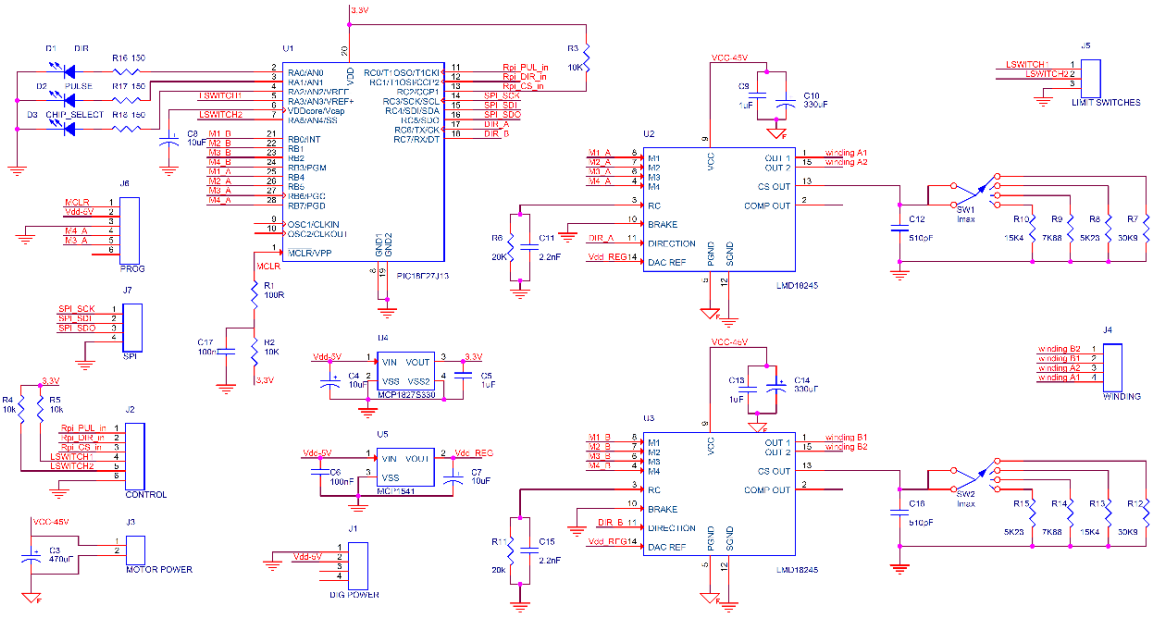


Figure 3.12. Stepper motor controller board circuit schematic.

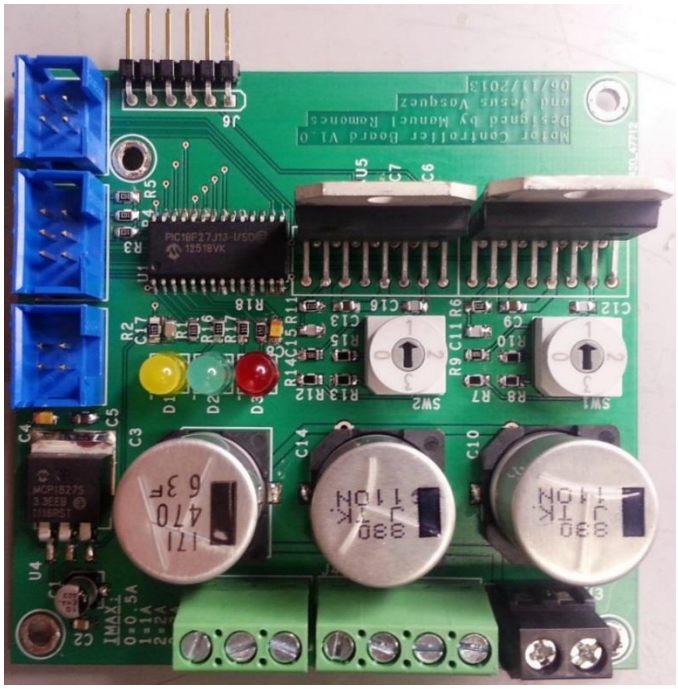


Figure 3.13. Stepper motor controller board.

3.2.1.4 General purpose IO board

A general purpose IO board was designed for applications where combination of digital and analog, inputs and outputs signals were required. The board features: eight analog inputs, eight analog outputs, 16 digital inputs, 16 digital outputs and one RS232 serial port.

For the analog inputs, it was used the same ADC showed on Figure 3.4. The resolution of this ADC is 16 bits, and, in this case, the input range can be changed using local dipswitches between 0-5 V, 0-10 V, +/-5 V and +/-10 V. The eight analog inputs are multiplexed by a DG408 [16] into the ADC input. The Raspberry Pi synchronously controls the multiplexer and ADC conversion in order to acquire all eight inputs. Figure 3.14 shows the circuit schematic of this section of the board.

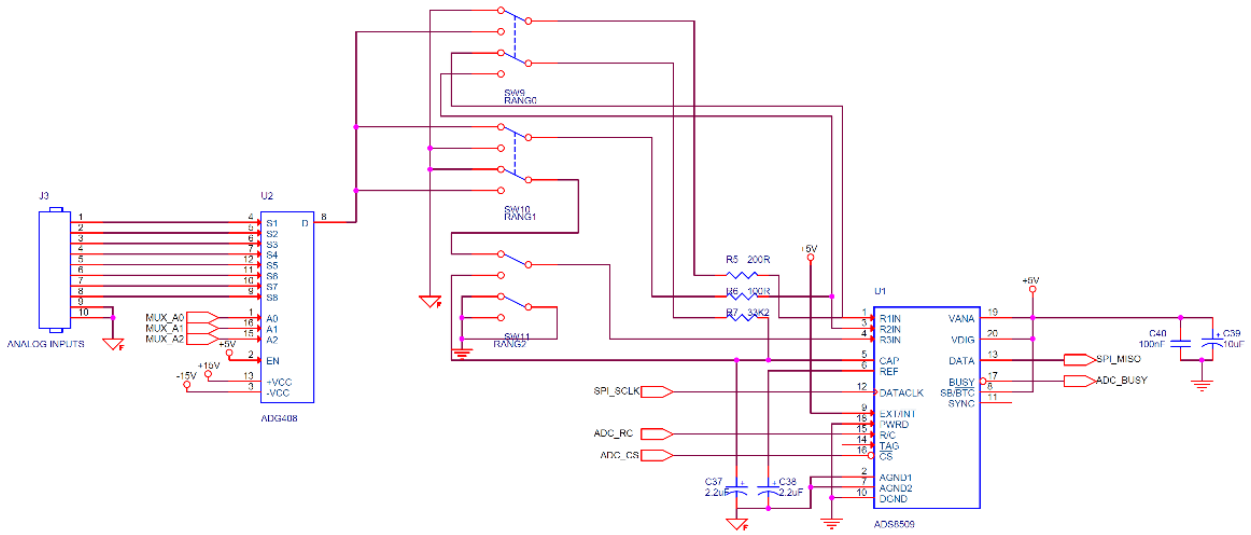


Figure 3.14. Analog input stage of the general purpose IO board.

Analog outputs are generated using two DAC8734 D/A converters [22]. The DAC8734 is a quad-channel, 16-bits digital-to-analog converter (DAC), with serial interface compatible with the SPI bus. The output range of each channel can be set using local dipswitches between 0-5 V, 0-10 V, +/-5 V and +/-10 V. The serial interface of both DACs are connected on daisy-chain mode allowing the Raspberry to update all eight outputs on a single SPI request. Figure 3.15 presents it functional block diagram. The circuit schematic of this part of the board is presented on Figure 3.16.

For digital inputs and outputs, two MCP23S17 I/O expander [23] were used. This device provides 16-bits, general purpose parallel I/O, divided on two 8-bits ports, with configurable direction and polarity. The device has a SPI interface for accessing its configuration and control registers. The functional diagram of this device is presented on Figure 3.17.

On the designed board, one MCP23S17 is used for providing 16 digital inputs (Figure 3.18), while the second one is used for proving 16 digital outputs (Figure 3.19). The Raspberry Pi controls both devices using the SPI bus interface.

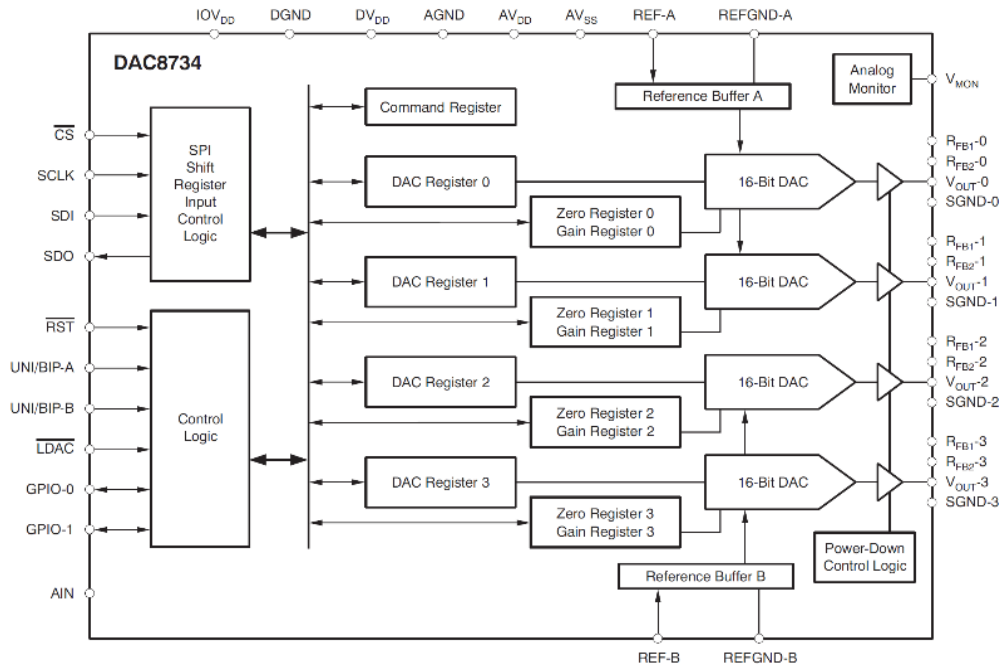


Figure 3.15. DAC8734 functional block diagram. (Taken from [22]).

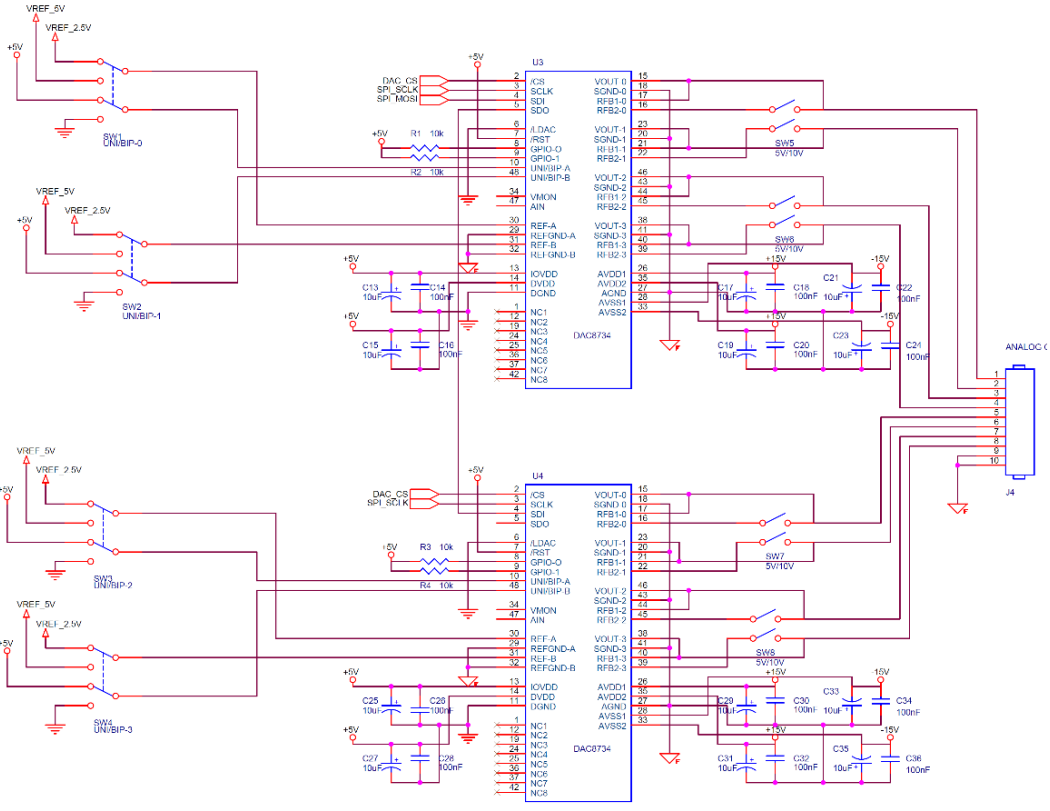


Figure 3.16. Analog output stage of the general purpose IO board.

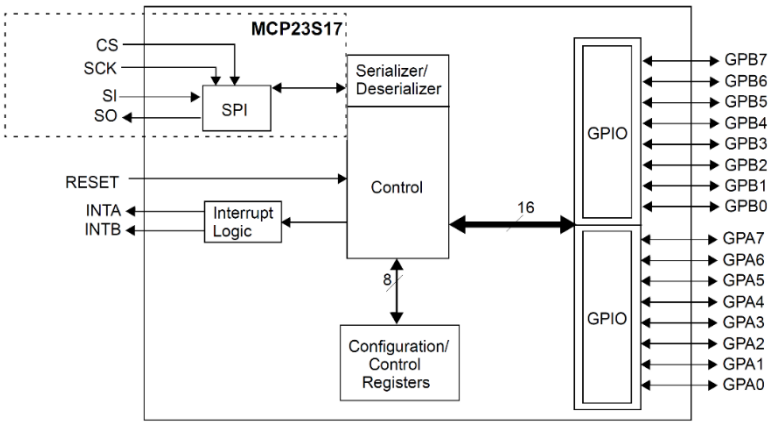


Figure 3.17. MCP23S17 functional block diagram. (Taken from [23]).

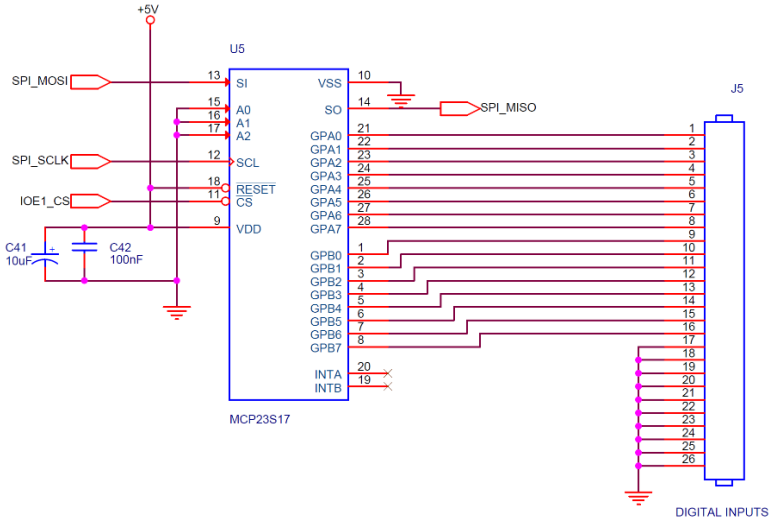


Figure 3.18. Digital input stage of the general purpose IO board.

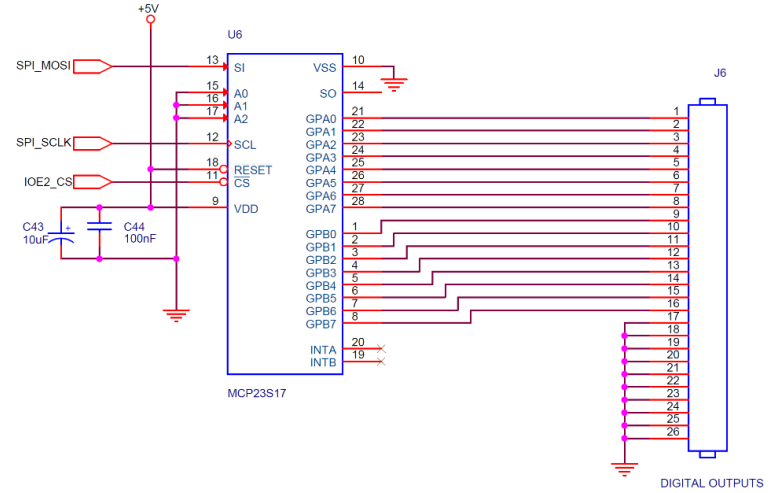


Figure 3.19. Digital output stage of the general purpose IO board.

Finally, a RS232 transceiver (MAX3227 [24]) is used to make available the Raspberry Pi's serial port on this application. Figure 3.20 presents the circuit schematic of this part of the board.

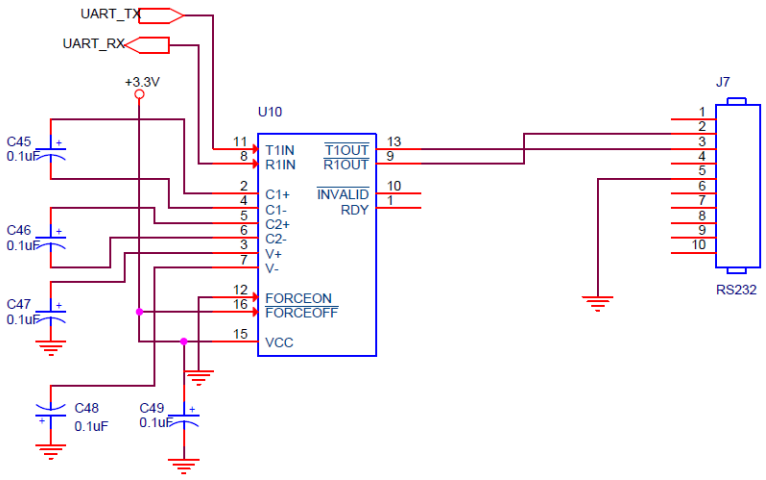


Figure 3.20. RS232 serial port of stage of the general purpose IO board.

Figure 3.21 shows a picture of the final system, formed by the general purpose IO expansion board connected to a Raspberry Pi. The Raspberry Pi GPIO port is directly connected to the board, where serial port and IO interfaces are available on separated connectors. The entire system is powered using an external power supply.

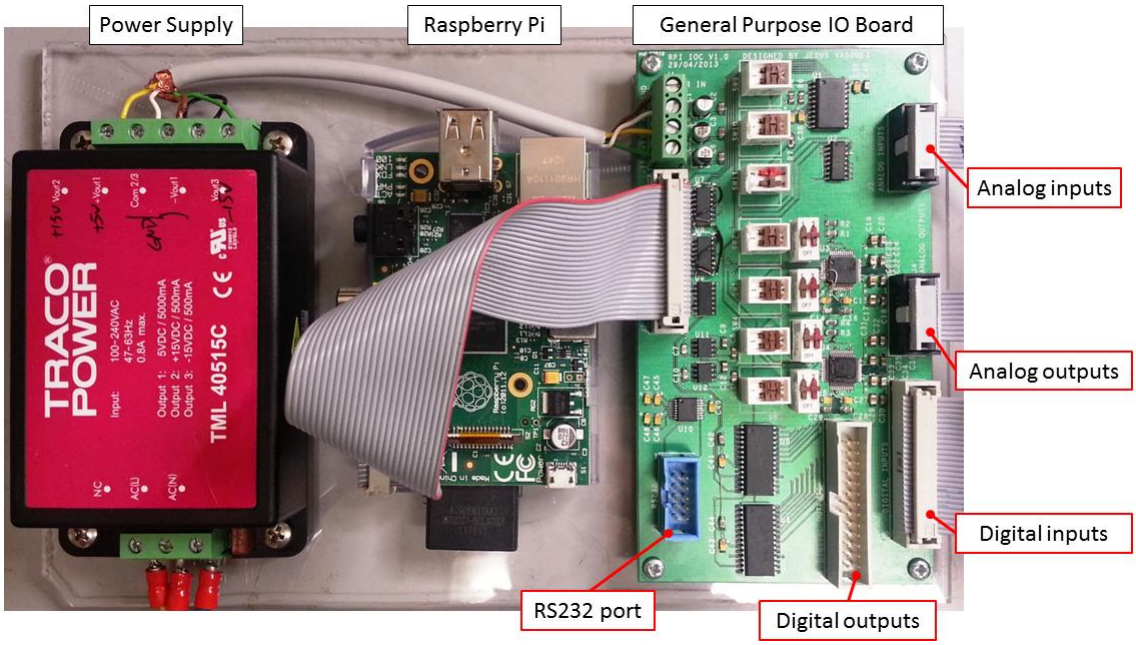


Figure 3.21. General purpose IO board.(Form left to right: external power supply, Raspberry Pi and expansion IO board. On the expansion board the different services are available through independent connectors).

3.2.2 Software Implementation

The operating system used on the Raspberry Pi is a flavor of Linux based on Debian, called Raspbian OS [13]. It is the most widely used Linux distribution tailored for this hardware platform. EPICS distribution (version R3.14.12.3) was compiled and installed on the system. Two software supplements, AsynDriver (version 4.20) [25] and StreamDevice (version 2.6) [26], where added to the EPICS installation for some applications. These two packages allow the control of new devices to be performed according to a driving paradigm that will be describe later. The control systems where developed as soft IOCs running on these systems.

In order to communicate with the custom expansion boards, a program written on C acts as interface driver. This program performs two functions: on one hand, it uses the Broadcom BCM2835 C library to access the GPIO port in order to write and read data to and from the expansion boards; on the other hand, using the EPICS Channel Access C library, to reads and writes this data to EPICS interface records on the soft IOCs running locally on the device (see Figure 3.22). Inside the IOC, these interface records are processes according to the control algorithm.

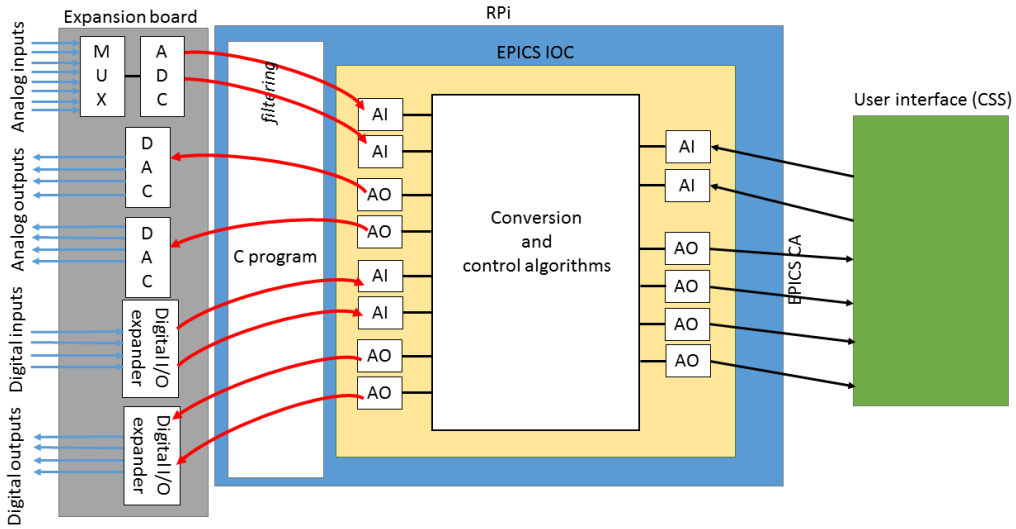


Figure 3.22. Interface between the EPICS IOC records and the expansion board using a C program.

The interface driver structure depends on the IOC application. However, in general, for example for the general purpose IO board describe on chapter 3.2.1.4, the structure is as shown on Figure 3.23. First, the

EPICS CA channels are created and configured. Then, the main loops starts, where the input signals are read and the output signals are updated at each cycle.

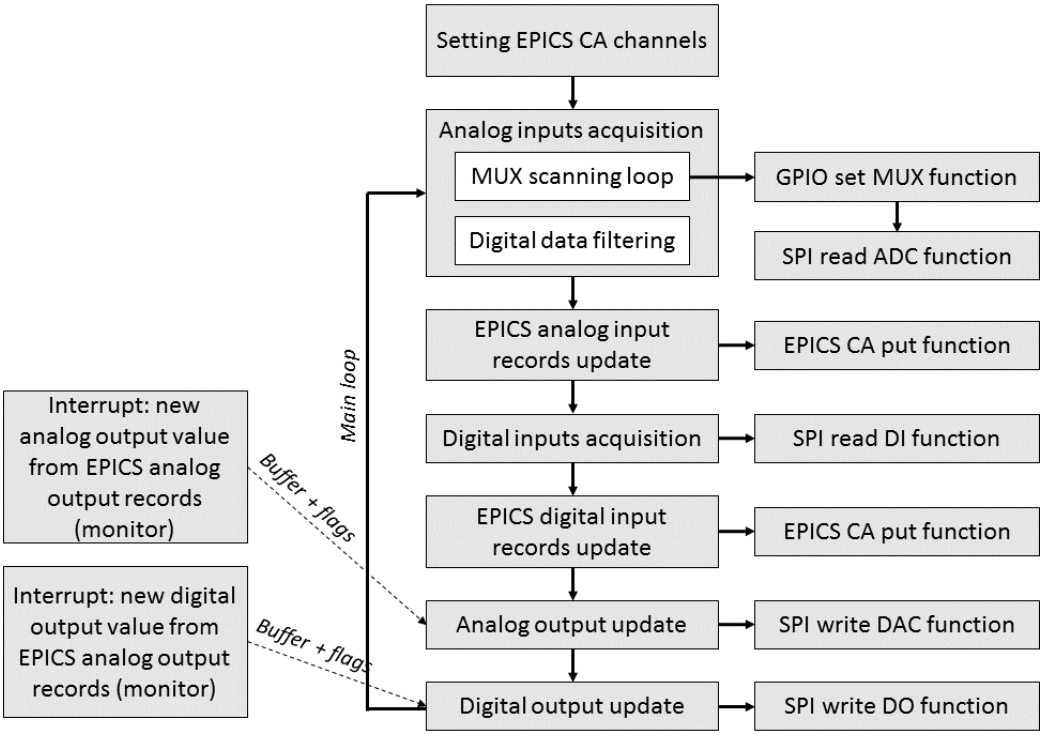


Figure 3.23. General interface driver flow diagram.

The analog input acquisition block, is itself a loop that synchronously set the multiplexer address and then reads the A/D value through the SPI interface. Moreover, a low pass digital filter is implemented, whose parameters can be selected by the user. Once that each new value is acquire, it is then written to the corresponding EPICS interface record, using the EPICS CA put function.

The digital inputs are acquired using the SPI interface. Then, as for the analog inputs, the new values are written to the EPICS interfaces records using the EPICS CA put function.

For the digital and analog outputs, EPICS CA monitors are set on the interfaces records. These monitor functions, will call an interrupt function each time a new value is written on the corresponding EPICS record. Inside this interrupt, the values are written on buffers and flags are set indicating the presence of the new values. On the main loop, when these flags are detected, the values are taken out of the buffer and then written to the DAC or digital outputs through the SPI interface.

The source code of this interface driver is reported on the appendixes.

For each application, a graphical user interface was developed using the EPICS CA client CSS (Control System Studio) [8]. From this interfaces the user can read the process values as well as requesting action to the system. Figure 3.24 shows an example of a screen of the user interface.

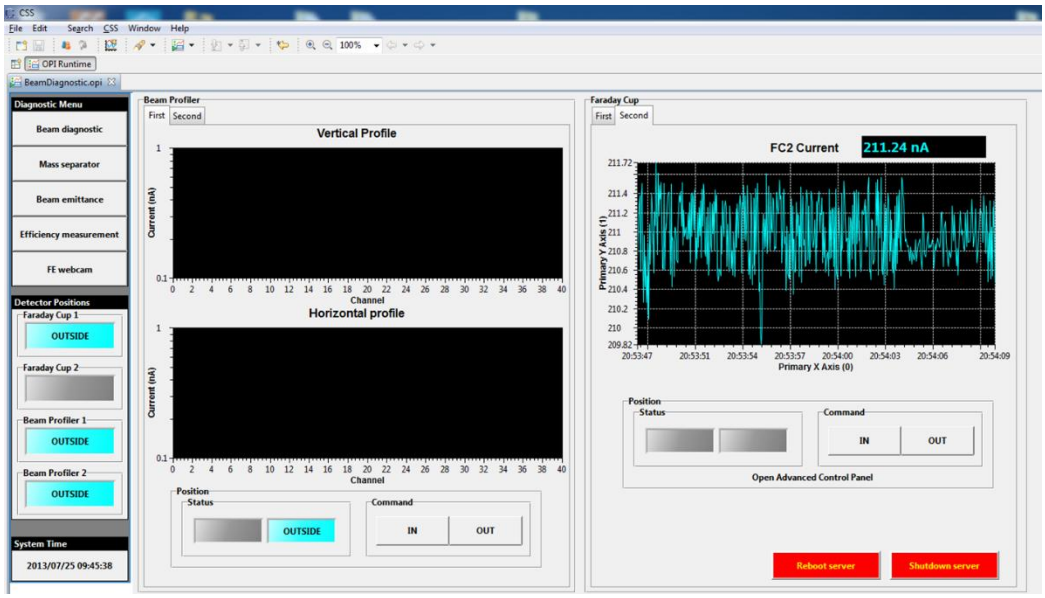


Figure 3.24. Screen of the user interface.

3.3 Experimental tests and results

In order to assess the performance of these IOCs, there were performed a set of test focused at determining four main parameters: the A/D effective resolution, the D/A effective resolution, the input current acquisition precision, and the maximum sample rate and CPU usage.

The teste were performed using the general purpose IO boards, described on chapter 3.2.1.4, except the current acquisition test, were the acquisition boards described on chapter 3.2.1.1 was used.

In the following chapters, the experimental tests will be explained and the obtained result will be presented for each case.

3.3.1 Acquisition rate and CPU usage performance

This test was performed in order to estimate the relationship between the acquisition rate and the CPU usage by the developed applications. The test consists on measure the CPU usage both by the interface driver and the EPICS application for different sample periods.

The interface driver is custom made for each application. Therefore, this test was performed only on the interface driver described on chapter 3.2.2, e.g. for the general purpose IO board presented on chapter 3.2.1.4, whose source code is reported on the appendixes.

On each acquisition loop, the interface driver read the eight analog and 16 digital inputs, and update the eight analog and 16 digital outputs. For varying the sample period, a delay was embedded on the main loop of the acquisition thread of the interface driver.

Three different sample period were selected. The first one corresponds to the faster sample period that can be achieved (setting the delay to zero). The last one correspond to 10 Sample/s, a usual value chosen as sample frequency on several applications at LNL. Finally, the third value was selected in between the other two values. The results are presented on Table 3.1. Figure 3.25 shows the CPU usage versus the real measured sample frequency for all three cases. The sample frequency presented on these results represents the frequency for each channel.

Table 3.1. CPU usage performance results.

Sample period average (ms)	Sample period standard deviation (ms)	Sample frequency (S/s)	Interface driver CPU usage (%)	EPICS IOC application CPU usage (%)
11.94	2.07	83.76	45	45
17.87	1.99	55.96	31	30
72.06	1.96	13.88	8.5	8

From the results, it can be seen that the faster acquisition period for each analog input channel is 83.76 Sample/s. Moreover, Figure 3.25 shows how the CPU usage increases for faster acquisition periods, as expected.

For the faster case, the interface driver and EPICS application use around 90% of the CPU. This is due the fact that the loop it is constantly executed, without any delay (time where the CPU is free). On the other hand, much less CPU is used, around the 17%, for a sample frequency of 13.88 Sample/s.

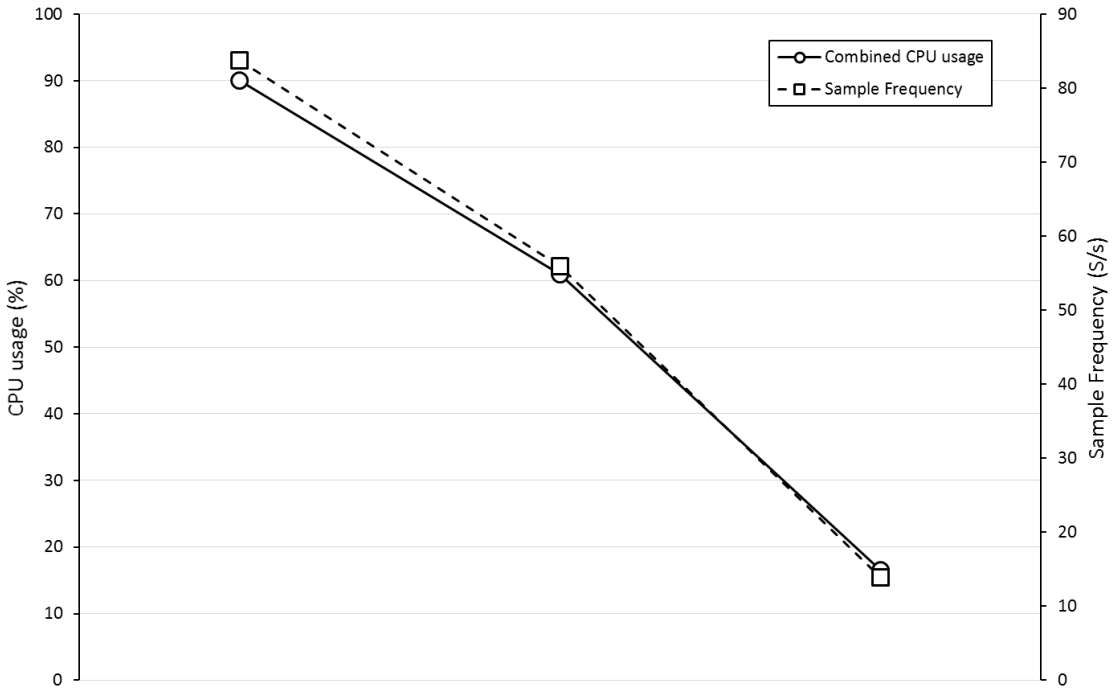


Figure 3.25. CPU usage versus sample frequency.

3.3.2 Analog-to-Digital circuit performance

A test was performed in order to determine the Effective Number of Bits (ENOB) of resolution of the analog inputs on the boards presented on chapter 3.2.1.4. The ENOB definition according to [27] is defined as,

$$ENOB = 0.5 \cdot log_2(SINAD) - 0.5 \cdot log_2(1.5) - log_2\left(\frac{A}{V}\right) \quad (3.3)$$

Where *A* is the peak to peak amplitude of a sine wave apply to the input of the ADC; *V* is the full-scale range of the ADC input; and *SINAD* is the signal to noise and distortion ratio, which is defined as,

$$SINAD = \frac{P_S}{P_{NAD}} \quad (3.4)$$

Where P_S is the power of the input signal; and P_{NAD} is the noise and distortion power.

In equation (3.3), SINAD is defined as a ratio of rms values according to equation (3.4). We can rewrite this definition of ENOB in function of SINAD expressed in dBs resulting in,

$$ENOB = \frac{SINAD - 1.76 + 20 \cdot log\left(\frac{V}{A}\right)}{6.02} \quad (3.5)$$

A method for calculating the ENOB using frequency domain analysis is defined in [27]. It consists in applying a sine wave signal at the input of the ADC, and then calculating the Fast Fourier Transform (FFT) of the acquired data (after subtracting the DC component, if any). From the power spectrum obtained through the FFT, the signal power is obtained from the bin corresponding to the input signal frequency, while the noise and distortion power is represented as the sum of power in all other frequency bins excluding the zero frequency and the signal frequency bins.

The minim number of samples that must be taken and analyzed, are defined in [27] as,

$$M = \pi 2^B \quad (3.6)$$

Where *B* denotes the number of bits of the ADC. In our case, as the ADC used has a resolution of 16 bits, the number of samples taken were more than 205888.

There is no specification in [27] about the amplitude of the input signal, as the difference between the full-scale range and the actual signal amplitude is taken into account in the ENOB definition (equation (3.3)). In the test performed, it was select -/+10 V as the ADC input range, with a 90% full-scale input signal.

The input signal frequency was chosen in order to be lower than the Nyquist frequency [28]. It was chosen the maximum sample rate obtained on chapter 3.3.1, that is, 83 Sample/s as sample frequency for this test. Subsequently, the input signal frequency was set to 35Hz, which satisfies this condition. Furthermore, the test was also performed for other three lower frequencies: 25Hz, 10Hz and 5Hz.

The input sine wave signals were produce using a Hewlett Packard 33120A waveform generator and the samples values were acquired and stored using EPICS CA tools. Data have been processed using Scilab [29], calculating the sample average rate, the signal input average frequency, the FTT and the SINAD and ENOB defined on equations (3.5) and (3.4) respectively. The Scilab code used on these calculations is reported on the appendixes.

The power spectrum obtained calculating the FFT for the 35Hz input signal is presented on Figure 3.26, as an example. Similar results were obtained for the other cases. The calculate results are presented on Table

3.2, while a plot of the obtained ENOB for the different input signal frequencies is presented on Figure 3.27.

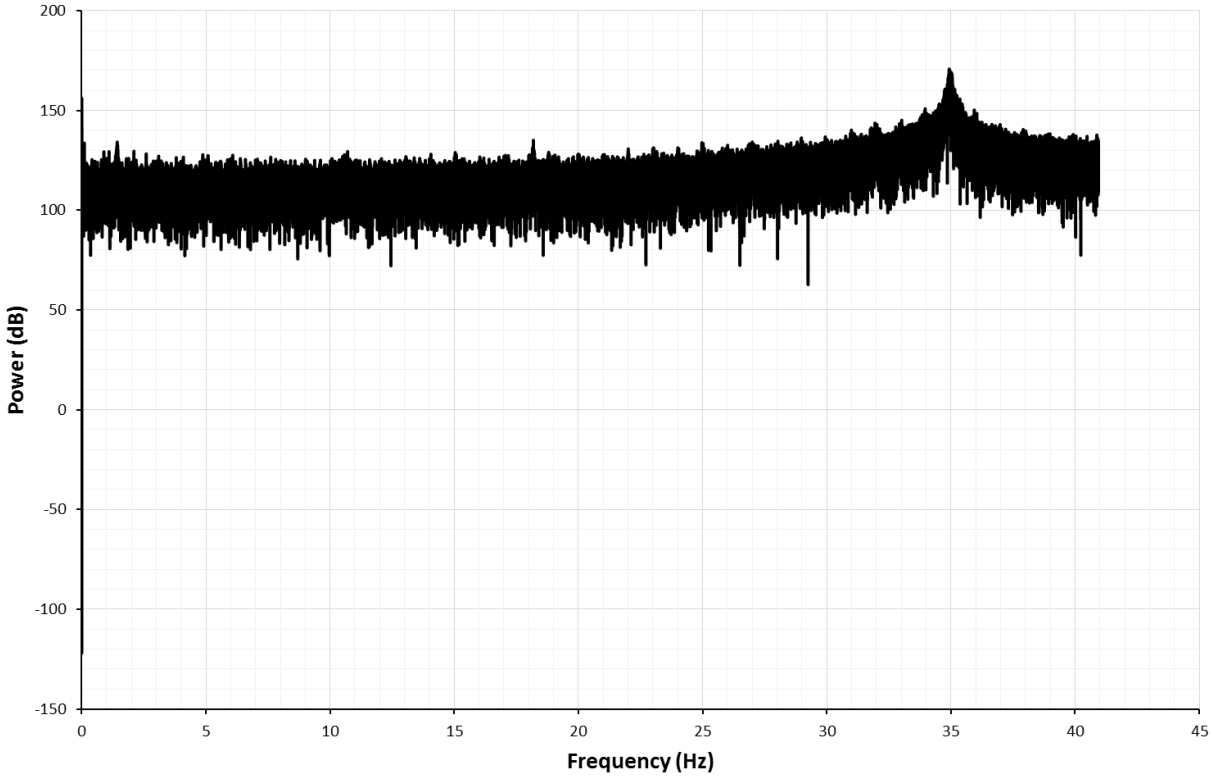


Figure 3.26. Power spectrum for a 35Hz sine wave signal. (Sample rate = 85 S/s, ADC range = +/-10V, input signal 90% full-scale).

Table 3.2. ADC resolution performance results.

Reference Frequency (Hz)	Measured Input Frequency (Hz)	Measure Sample Rate (S/s)	SINAD (dB)	ENOB (bits)
5	81.81	5	90.31	14.86
15	81.83	14.99	87.67	14.42
25	81.92	24.96	86.69	14.26
35	81.88	34.93	84.68	13.93

The higher ENOB obtained from the experiments is 14.86 bits, for the lower frequency input signal. In addition, it can be appreciated that the SINAD degrades for higher frequencies as expected, consequently reducing the ENOB, until a minimum value of 13.93 bits.

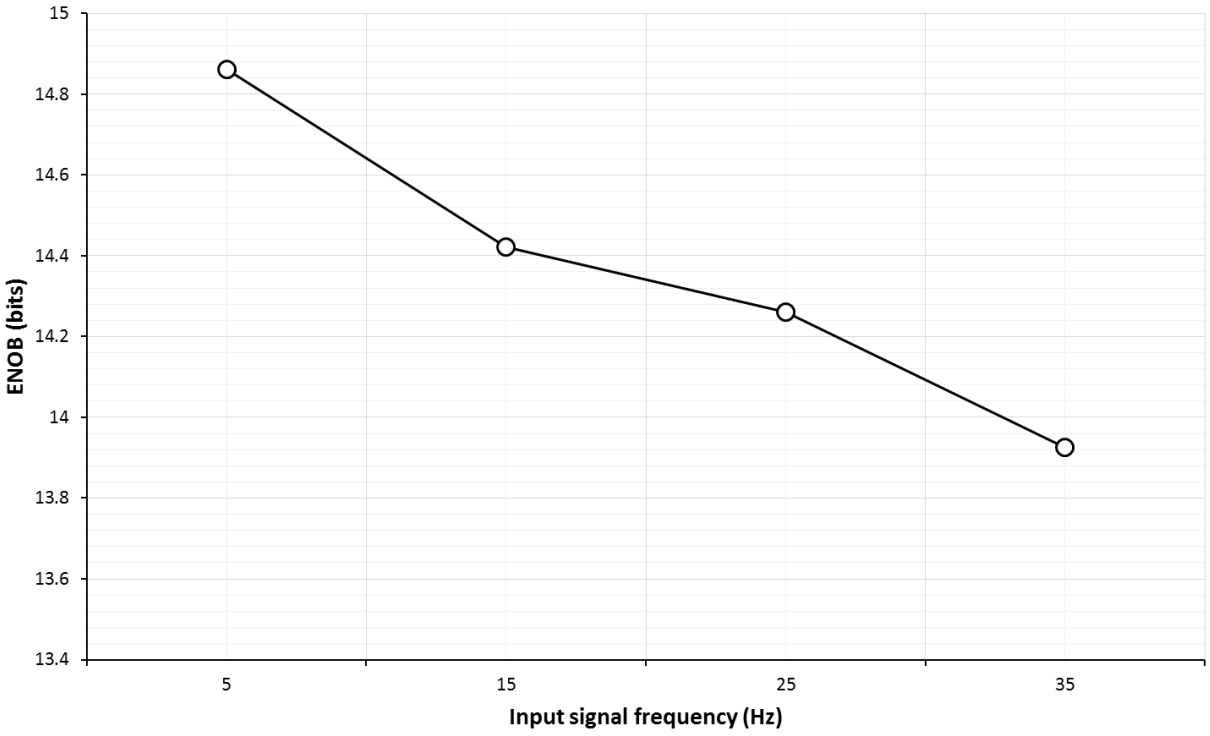


Figure 3.27. Estimated number of bits of resolution obtained for the analog inputs.

A second test was performed in order to estimate the bias error for DC input signals. This test consisted on applying constant values to the analog inputs of the board and archiving a set of data for statistical analysis.

A Tektronik PWS4305 programmable DC power supply was used for producing seven constant values, from -7.5 V to 7.5 V with steps of 2.5 V. For each input value, according to equation (3.6), more than 205888 points were acquired and archived using EPICS CA tools.

The ADC channel has a resolution of 16 bits and its input range was set to +/-10 V. For this conditions, the resulting ADC word equivalent to a given *X* volt inputs is expressed by,

$$ADCword(X) = \frac{(X + 10)}{20} \cdot 2^{16} \qquad (3.7)$$

Table 3.3 presents average, standard deviation, minimum, maximum, and bias of the measured data, for all the reference input voltages.

A plot of the average of the input measured vales is presented on Figure 3.28, while the standard deviation is presented on Figure 3.29. Figure 3.30 shows the calculated bias error values.

From Figure 3.28 and Figure 3.29 it is evident that the measure values fit the ideal curve in a very accurate way, with a standard deviation of the values between 0.6 and 1 mV. Moreover, the bias error showed on Figure 3.30 has values between the 40 and 60 mV approximately, on all cases.

Table 3.3. ADC bias error performance test results.

Reference Voltage	Average (V)	Standard Deviation (µV)	Minimum (V)	Maximum (V)	Bias (mV)
-7.5	-7.461	634	-7.468	-7.455	38.97
-5	-4.958	603	-4.964	-4.951	42.24
-2.5	-2.453	630	-2.459	-2.447	46.86
0	0.052	1046	0.045	0.059	51.72
2.5	2.552	680	2.545	2.558	51.72
5	5.056	661	5.050	5.062	56.36
7.5	7.561	699	7.554	7.567	60.99

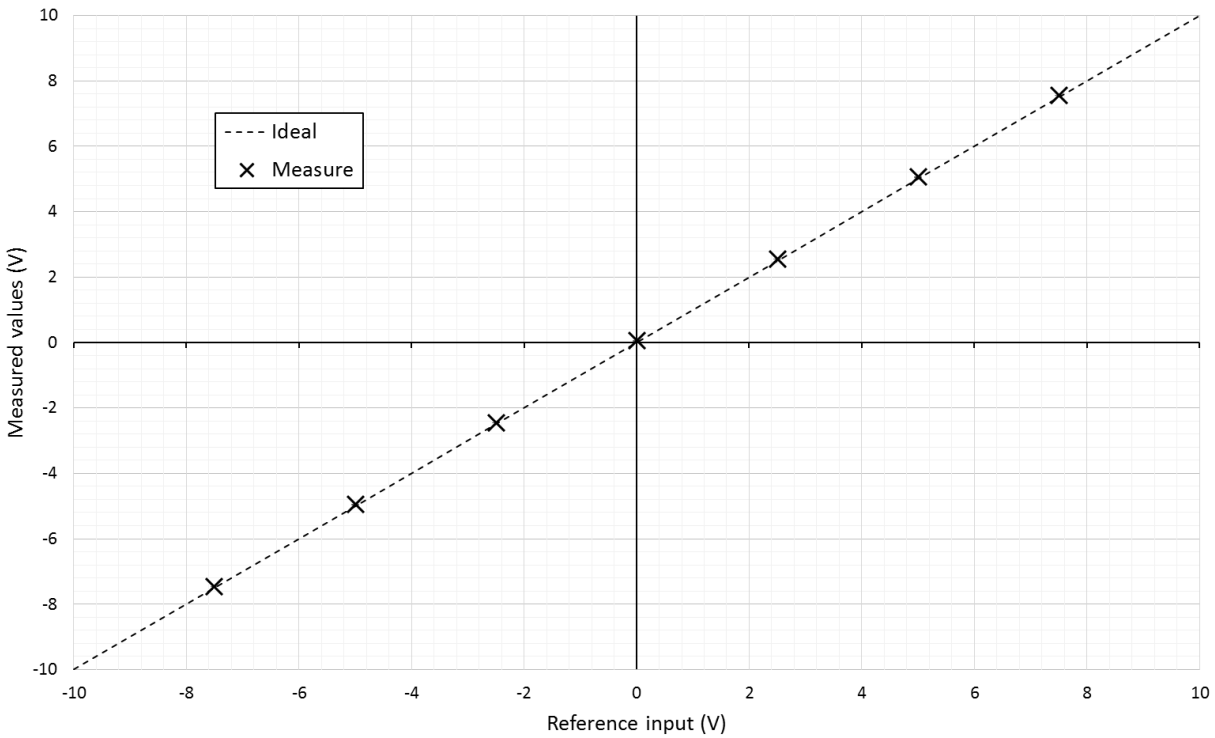


Figure 3.28. ADC measure average values. (The ideal case is presented as a reference).

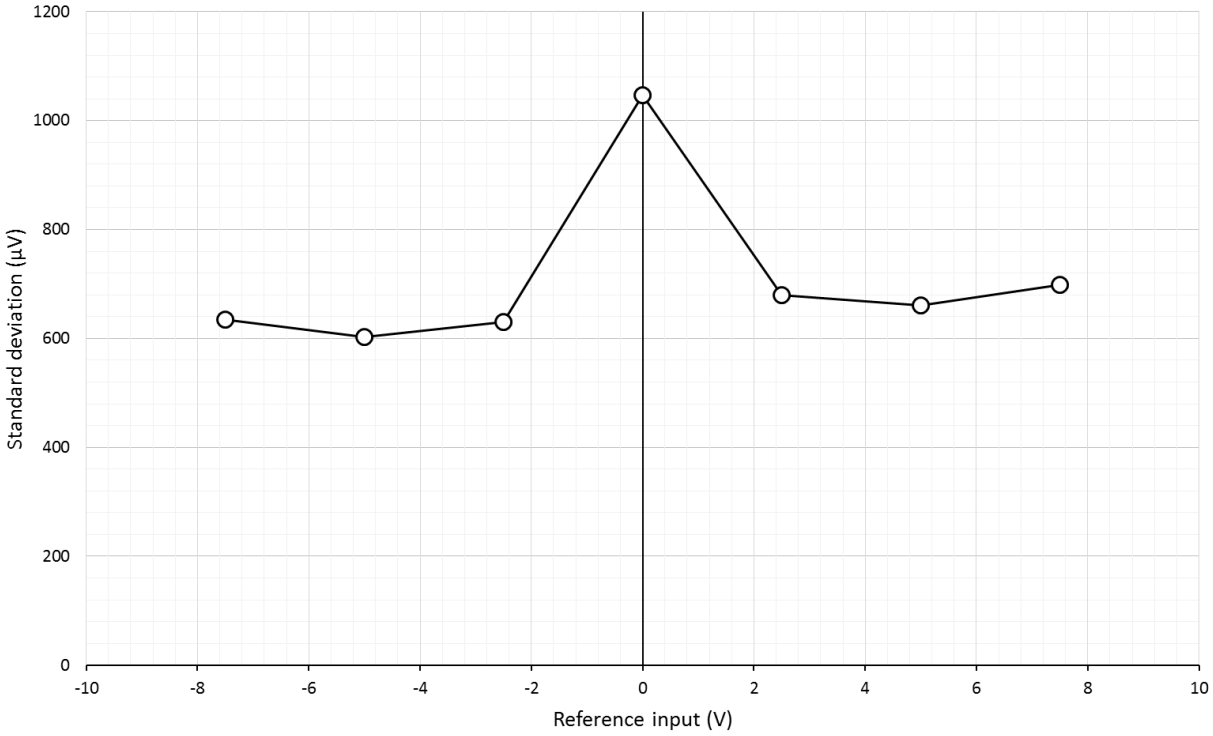


Figure 3.29. ADC measured standard deviation values.

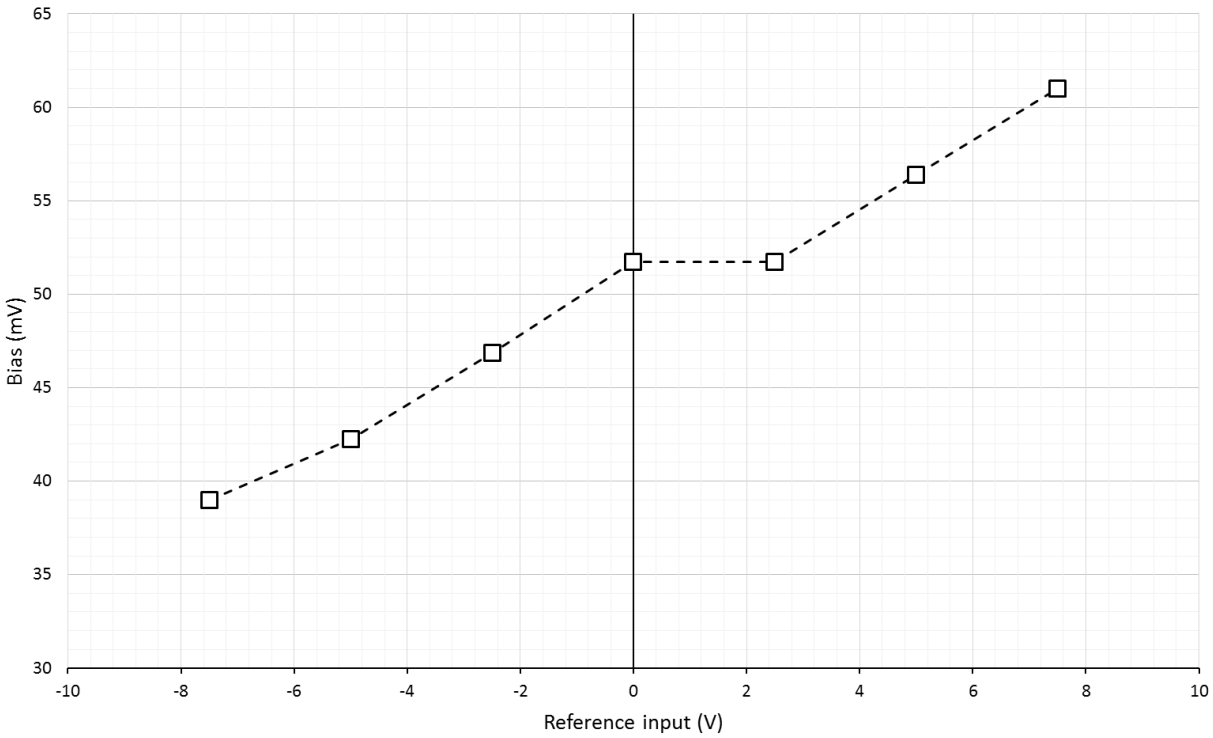


Figure 3.30. ADC input bias error values.

3.3.3 Digital-to-Analog circuit performance

A third test was performed in order to estimate the final resolution of the analog outputs presented on chapter 3.2.1.4. This test consisted on generating a constant output and using a commercial data acquisition system for measuring the output values.

Three constant values where generated using an analog output (set on range 0-10 V), 2.5 V, 5 V and 7.5 V. The output values were measured using a Tektronix DMM4020 digital multimeter. The data was collected through the multimeter serial port. Average, standard deviation, minimum, maximum, and bias error values where calculated and are presented on Table 3.4.

Table 3.4. DAC performance test results.

Reference Voltage	Average (V)	Standard Deviation (µV)	Minimum (V)	Maximum (V)	Bias (mV)
2.5 V	2.50	51	2.50	2.50	-0.73
5 V	4.99	91	4.99	4.99	-9.83
7.5 V	7.48	20	7.48	7.49	-15.10

Figure 3.31 and Figure 3.32 present a plot of the average of the measured output values and the standard deviation respectively. The bias error calculated values are shown on Figure 3.33.

It can be seem on Figure 3.31 that the measured values accurately fit the ideal curve, with a standard deviation nearly lower than 90 µV, as presented on Figure 3.32. The bias error values are below 16 mV, as seems on Figure 3.33.

The average standard deviation of the data presented on Table 3.4 is 54 µV, which on a 0-10 V, 16-bits DAC output represents 0.35 words. If we suppose that the output signal has a normal distribution, the 99.7% of the data points are within six standard deviations, which it is equivalent to 1.1 bits of resolution, approximatively. This gives us an estimation of 14.9 effective bits of resolution on the system analog outputs.

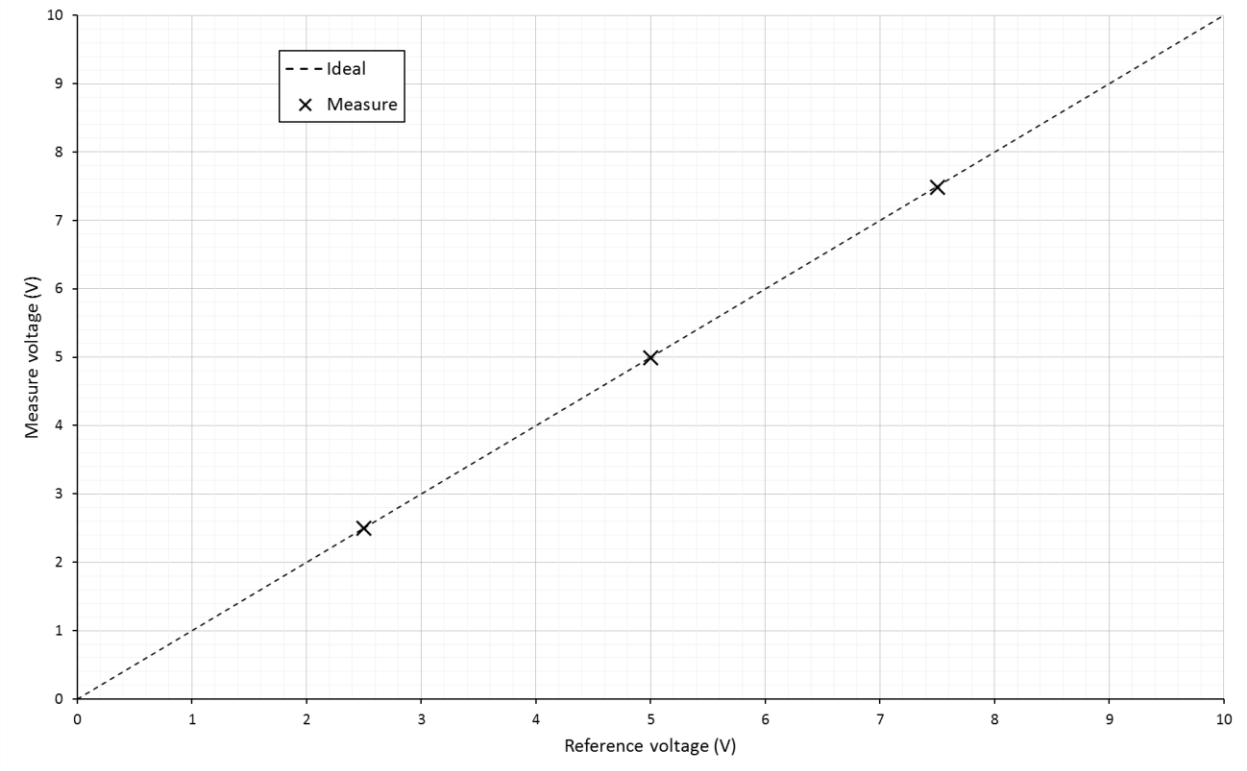


Figure 3.31. Measured DAC output voltages. (The ideal case is shown as reference).

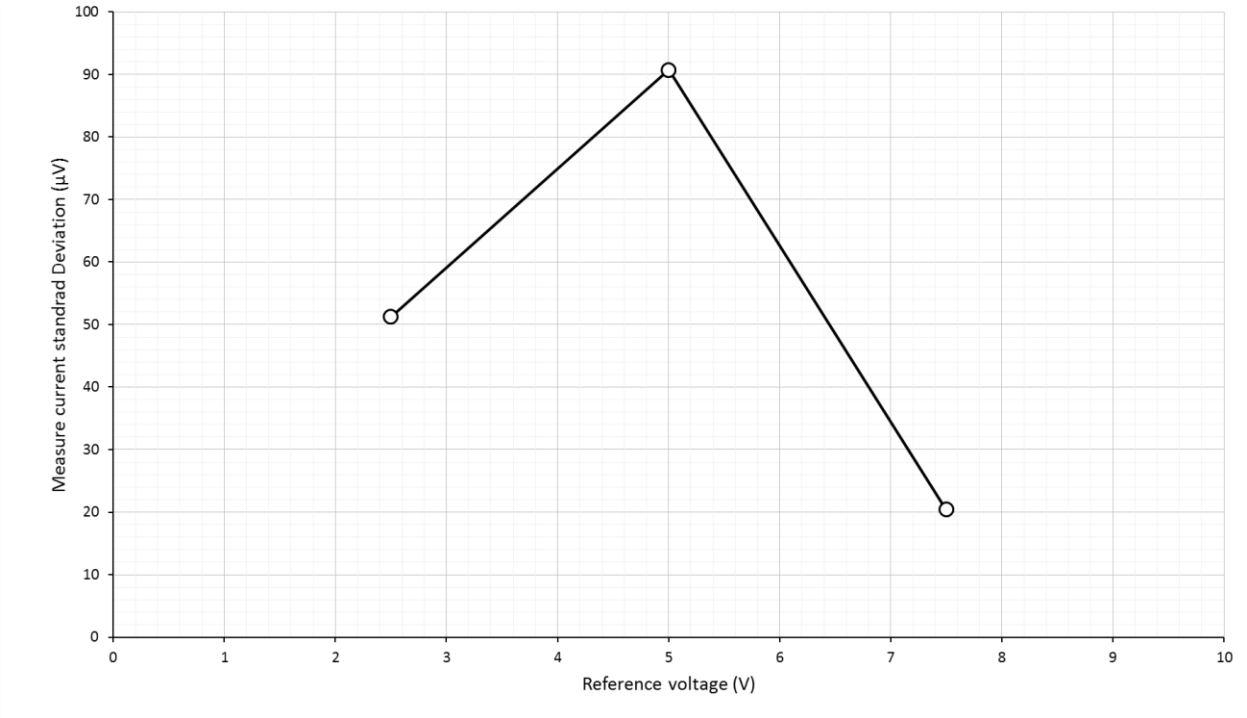


Figure 3.32. DAC standard deviation measured values.

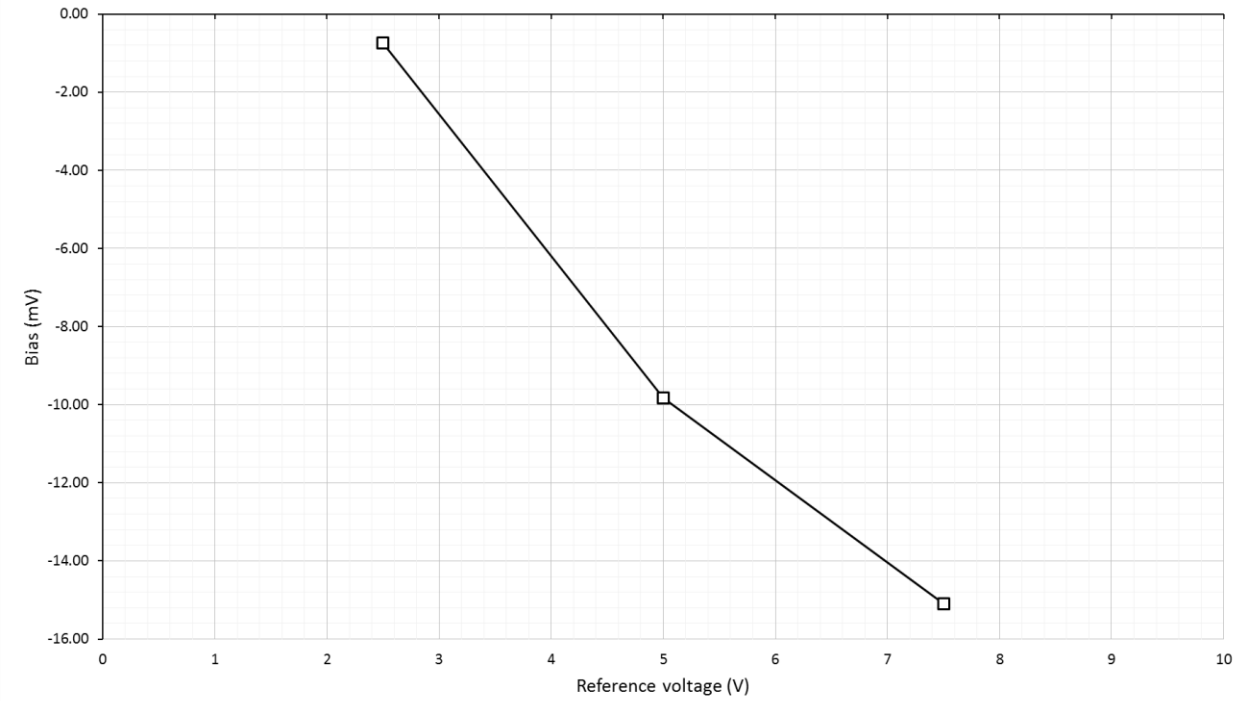


Figure 3.33. DAC output bias error values.

3.3.4 Current acquisition device performance

In order to verify the correct operation of the current acquisition circuit described on chapter 3.2.1.1 and 3.2.1.2, a constant current signal was applied on the input of the device using a commercial current source (a Keithley 6220 precision current source), while the resulting output data was acquired and archived using EPICS tools on the IOC.

Five current values were applied at the input: 1nA, 100nA, 10µA and 1mA. For each value, according to equation (3.6), more than 205888 points were archived. For each set of data, the average, standard deviation, minimum, maximum, error and bias were calculated and are presented on Table 3.5.

Table 3.5. Current input performance test results.

Reference Current	Average	Standard Deviation	Minimum	Maximum	Bias (%)
1 nA	1.01 nA	4.60E-03 nA	0.99 nA	1.04 nA	1.48
10 nA	10.15 nA	9.02E-03 nA	10.1 nA	10.18 nA	1.46
100 nA	101.54 nA	4.85E-02 nA	101.34 nA	101.7 nA	1.54
10 µA	10.18 µA	5.07E+00 nA	10.16 µA	10.2 µA	1.83
1 mA	1.05 mA	6.39E+02 nA	1.05 mA	1.056 mA	5.38

Figure 3.34 presents a plot of the average values versus the reference input current. The bias error computed values (expressed on percent respect to the reference input value) are shown on Figure 3.35.

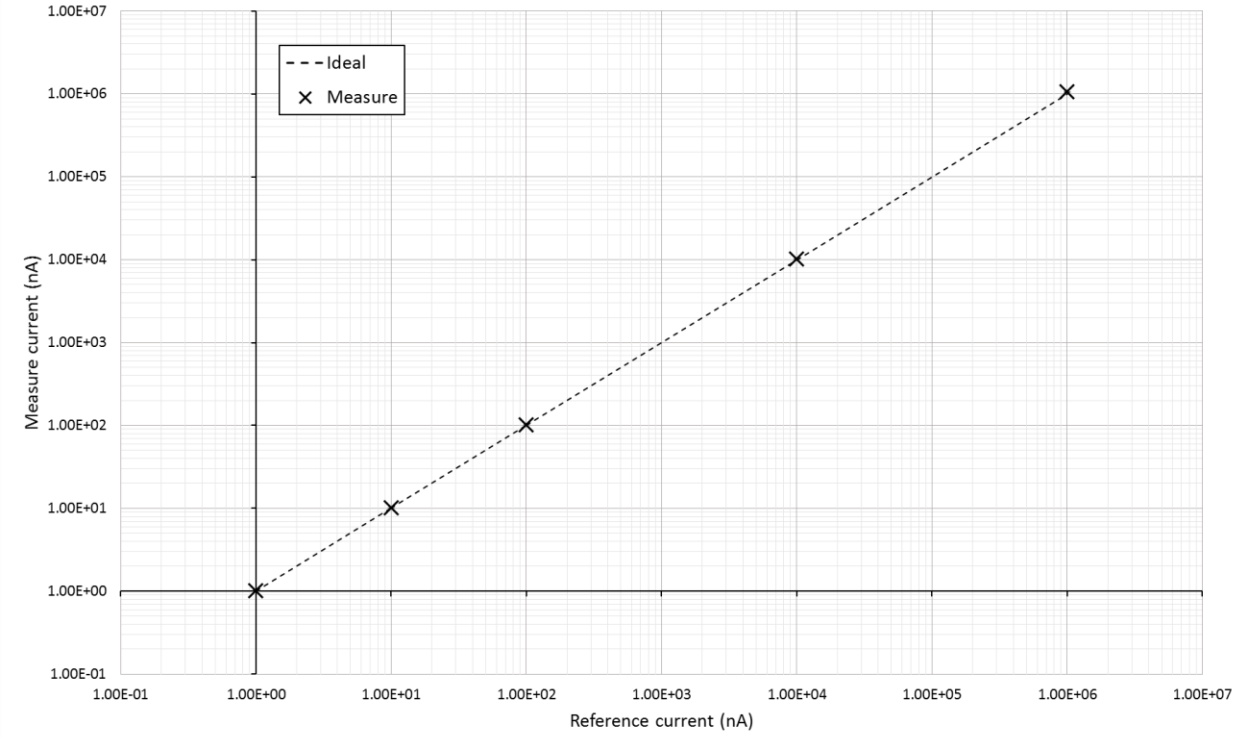


Figure 3.34. Measured current values. (The ideal case is plot as reference).

It can be seen on Figure 3.34 that the measured values fits the ideal curve. The bias error is practically below the 5% in all cases, as seem on Figure 3.35. Moreover, on real applications were these devices will be implemented at the off-line laboratory, they are usually used for measuring currents from tens of nA until some µA; in these cases the bias error is much lower, around the 1.7%.

One interesting aspect of the result showed on Table 3.5 is the increase on the standard deviation on the data as the reference input current increases. This behavior is plot on Figure 3.36.

The reason of this increase on the standard deviation of the data is the exponential calculation made on the acquisition software in order to transform the input voltage measured by the ADC, back to the original input current value.

For these acquisition circuits, we can refer the noise sources mainly to two categories, one before and one after the logarithmic amplifier, as shown on Figure 3.37. The effect of the exponential transformation on the noise before the logarithmic amplifier are canceled out by the logarithmic transfer function of the

amplifier. However, the noise at the input of the ADC (as for example, the ADC quantization noise) is only affected by the effect of the exponential transformation.

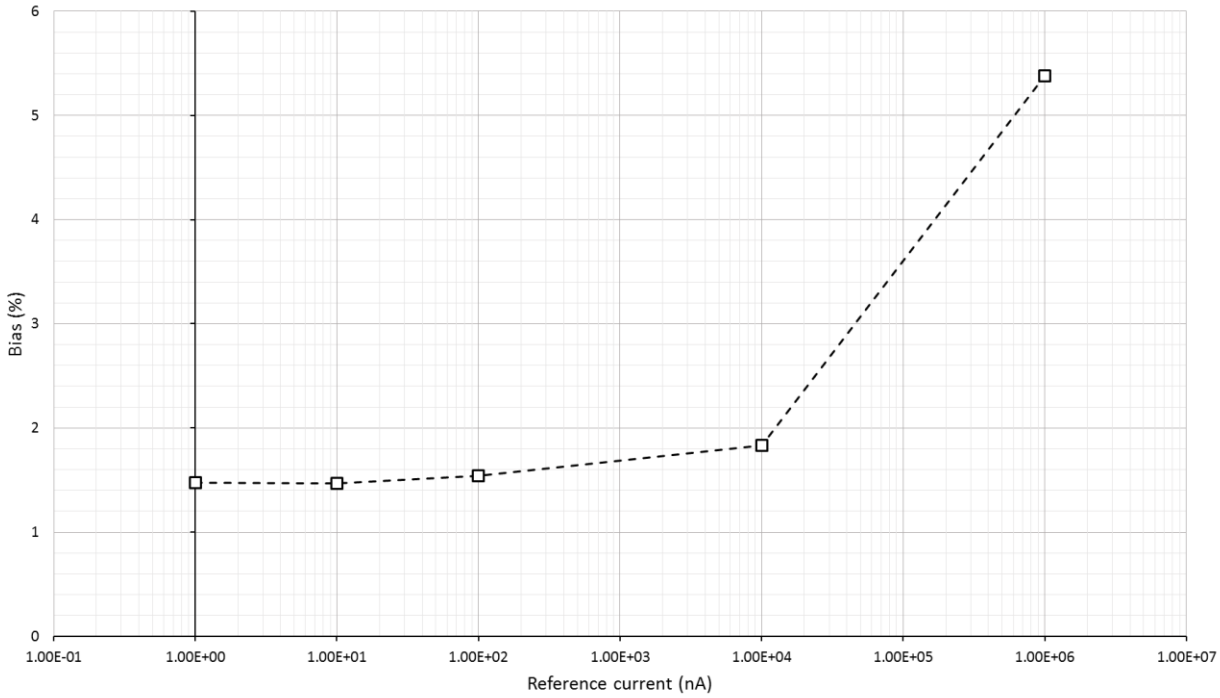


Figure 3.35. Measure current bias error.

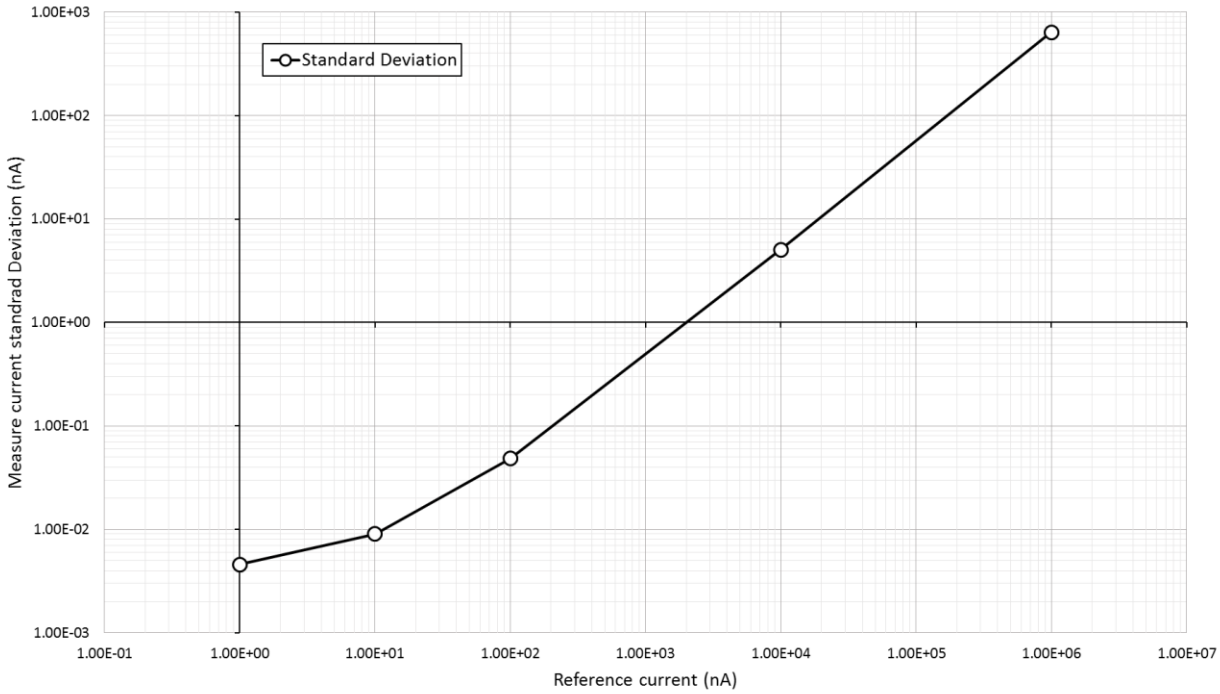


Figure 3.36. Measure current standard deviation values.

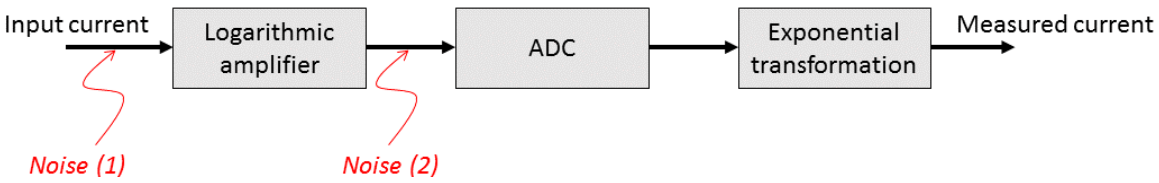


Figure 3.37. Noise source on the current acquisition devices. (The noise after the logarithmic amplifier are modified by the non-linear exponential transformation).

This transformation has the transfer function plotted on Figure 3.38. It is evident that it is a non-linear transformation, with higher amplification gains for higher voltage inputs. Consequently, for higher input currents (that in turn produce higher input voltages) the transfer function is steeper, thus amplifying more the noise and producing the effect showed on Figure 3.36.

Another effect produced by the exponential transformation can be appreciate plotting the histogram of the data acquired with a constant input current. In Figure 3.39 is presented the histogram of the data taken with a 1mA at the input of the circuit (the higher input current was chosen as the effect is larger and easier to visualize). The data should have, roughly, a normal distribution as the one plotted in the figure. However, it can be seem that the data distribution is shifted to the right. This effect is again, caused by the non-linear transfer function of the exponential transformation showed on Figure 3.38. The transformation gain is higher for higher input values, so the noise with larger value is amplified with a larger gain respect to the noise with smaller value, producing the effect shows on Figure 3.39.

A possible solution that was tested is the application of a median filter to the converted values. This filter should approximately reduce this unequal effect on larger and smaller values. A median filter of size 9 was apply to the data presented on Figure 3.39. The original and resulting data distributions are presented on Figure 3.40.

It is evident how the resulting data has a more normal distribution respect at the un-filtered data. Moreover, the standard deviation of the data was decreased from an original value of 639 to a value of 458, a reduction of almost 30%.

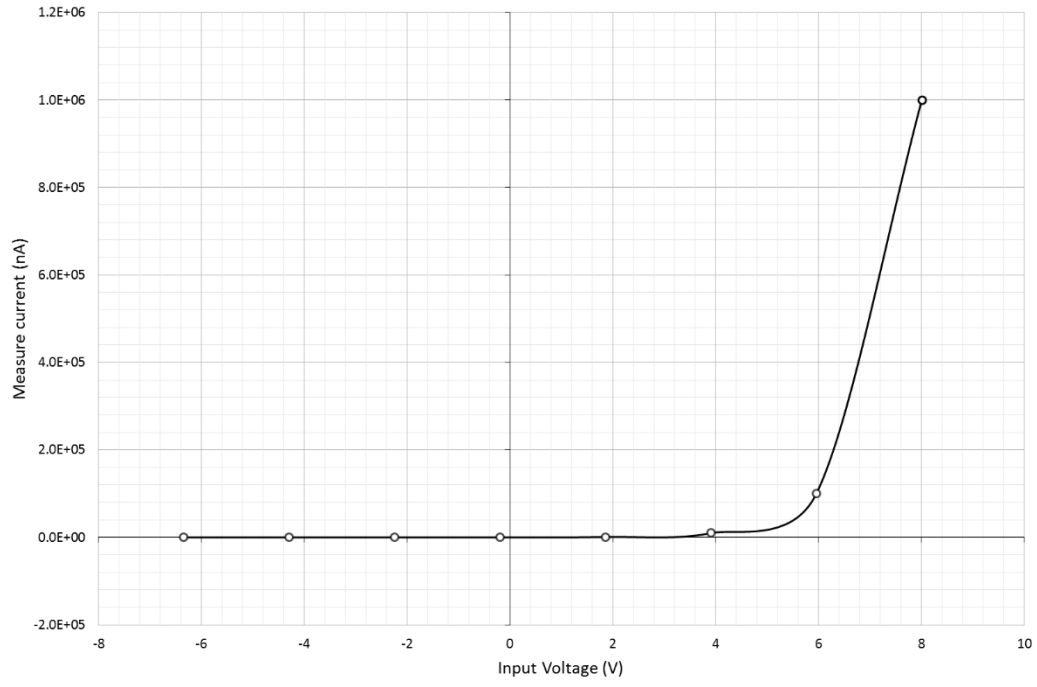


Figure 3.38. Exponential transfer function performed via software.

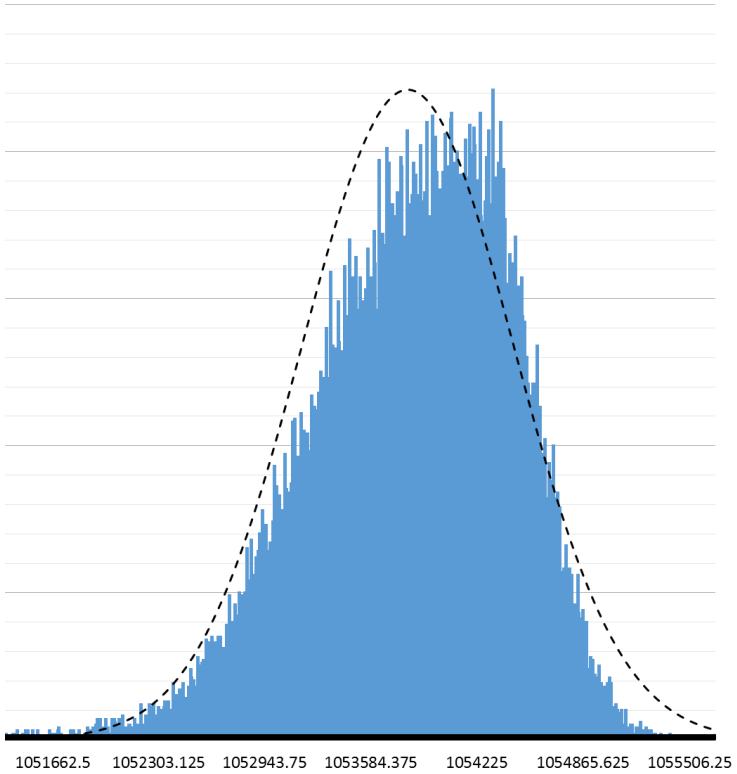


Figure 3.39. Histogram of data measured with 1mA at the input of the device. (A normal distribution with the average and standard deviation calculated from the data is show as reference).

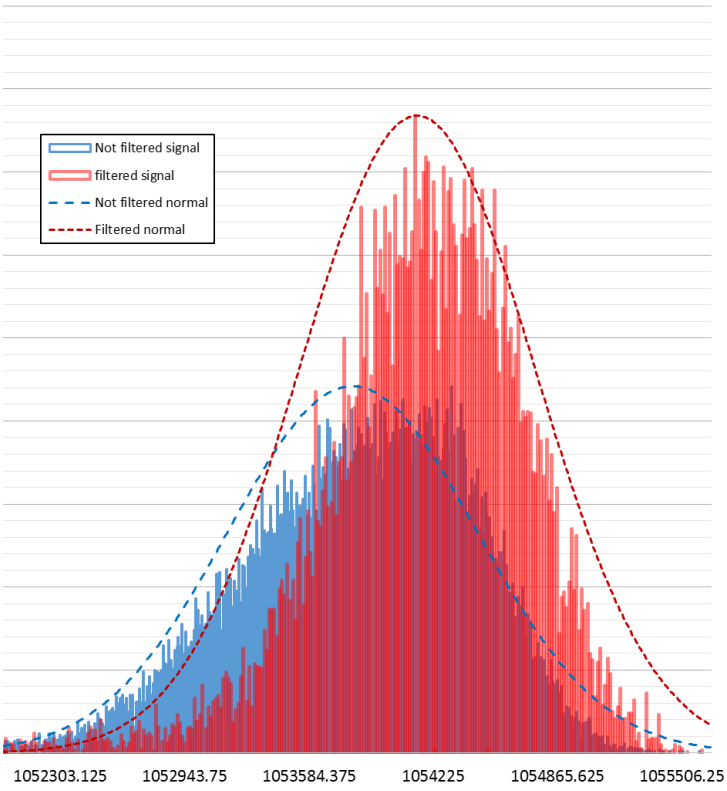


Figure 3.40. Histogram of data measured with 1mA at the input of the device, before and after the median filter. (Filter size 9).

3.4 Control System Implementation at the SPES off-line laboratory

Several IOCs have been implemented and are currently under test on several systems of the front-end apparatus, inside the SPES off-line laboratory. They control instrumentation devices as well as acquire data from detectors. They form a distributed control system using the EPICS channel access as communication protocol.

For each instrument or detector, a tailored IOC was designed and implemented, using the expansion boards explained on previous chapter as construction blocks. Each of these IOC will be describe on the following chapters.

3.4.1 Beam diagnostic data acquisition

The basic diagnostic unit of the front-end consists on a faraday cup, for beam current measurements, and a beam profile monitor, for the beam particle spatial distribution estimation. Both elements generate

analog current signals that represent the beam physical property. Furthermore, the positioning system of both detectors uses a stepper motor as actuator. Figure 3.41 presents a picture of the beam diagnostic unit installed at the SPES off-line laboratory.

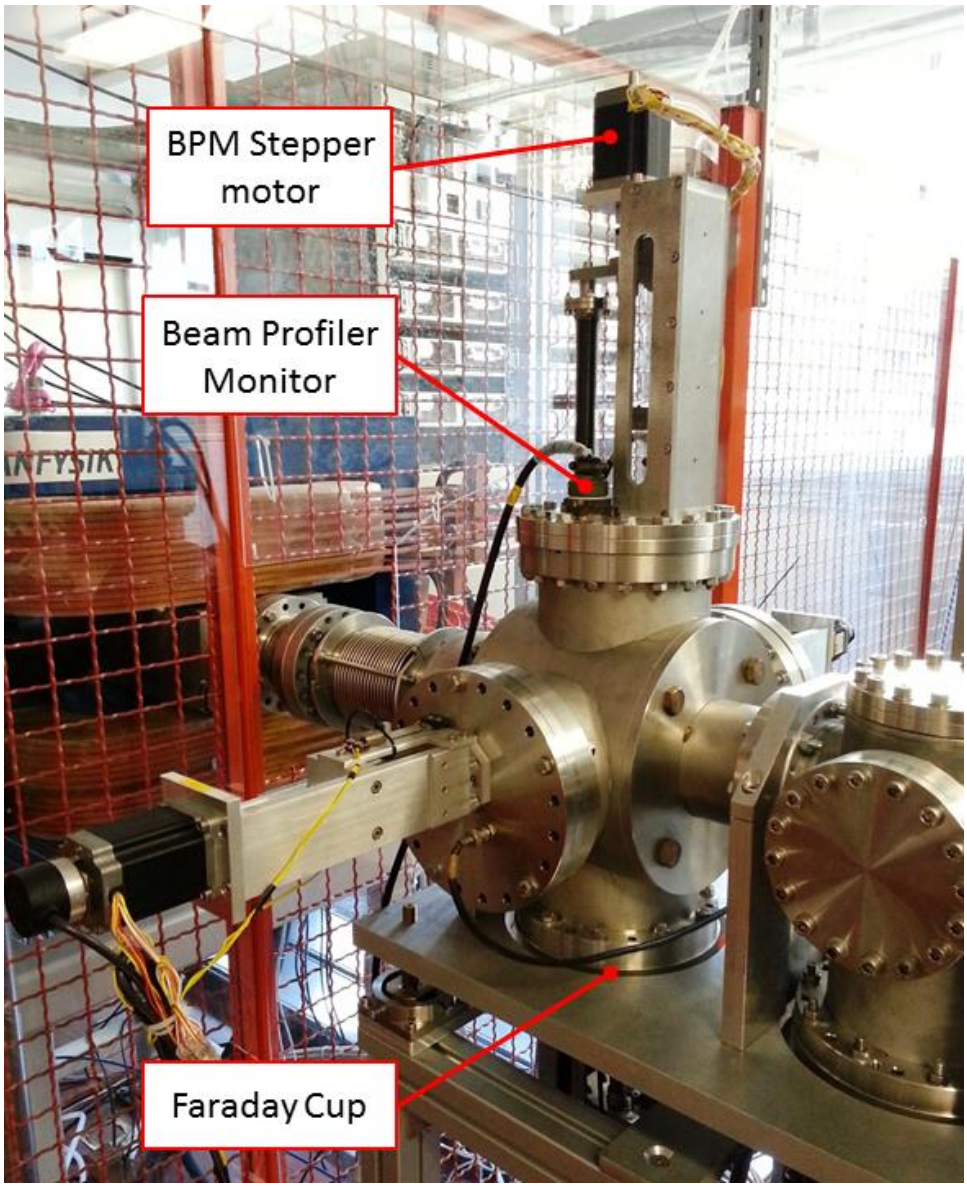


Figure 3.41. Beam diagnostic unit installed at the SPES off-line laboratory.

As the current intensity of the RIBs produce by SPES will be much lower respect to the stable beam produced at the moment at LNL, the existing detectors will not be suitable for detecting them. For that reason, a special beam profile detector was developed. This new beam profile detector presents two detection areas: a first one form by a wire grid following the design of the detectors at LNL; the second

detection area is form by a pad matrix, which acts as position-sensitive electron collection anode for a Microchannel Plate (MCP) [30]. The spacing of wires and pads on both detectors is 250 µm. Both system are designed on a single PCB (Printed Circuit Board), in order to use a single positioning system. Figure 3.42 presents a picture of the developed dual beam profile detector.

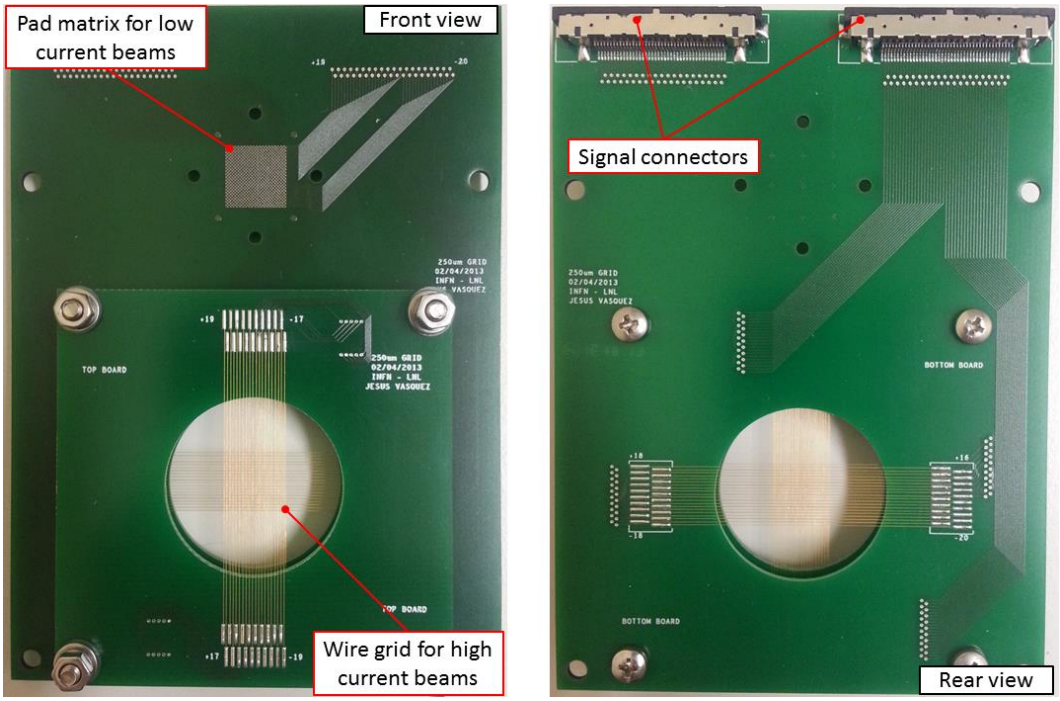


Figure 3.42. New dual beam profile detector. (Left: front view, right: rear view. On top, there is the MCP pad matrix anode, and on bottom, the conventional wire grid detector).

The IOC uses: one 1-channel, current-signal acquisition board for acquiring the data from the faraday cup; two 40-channel, current-signal acquisition boards for acquiring the data from the beam profile monitor (one for the X-plane and the other one for the Y-Plane data acquisition); and two stepper motor controller boards for controlling the position of both detectors. Figure 3.43 presents a block diagram of the IOC as described before, while Figure 3.44 shows the final IOC implementation.

All the boards are connected to the Raspberry Pi GPIO port. An ad-hoc interface driver was written for this board, whose flow diagram is presented on Figure 3.45 and source code can be found on the appendixes.

Three ADCs, on three different acquisition boards, are connected in a daisy-chain configuration, allowing reading the three values on a single SPI transaction. Forty readings, synchronized with the control of the

multiplexers, are needed to perform a full scanning of all inputs. The read data are filtered using a low-pass digital filter whose parameters can be selected by the user.

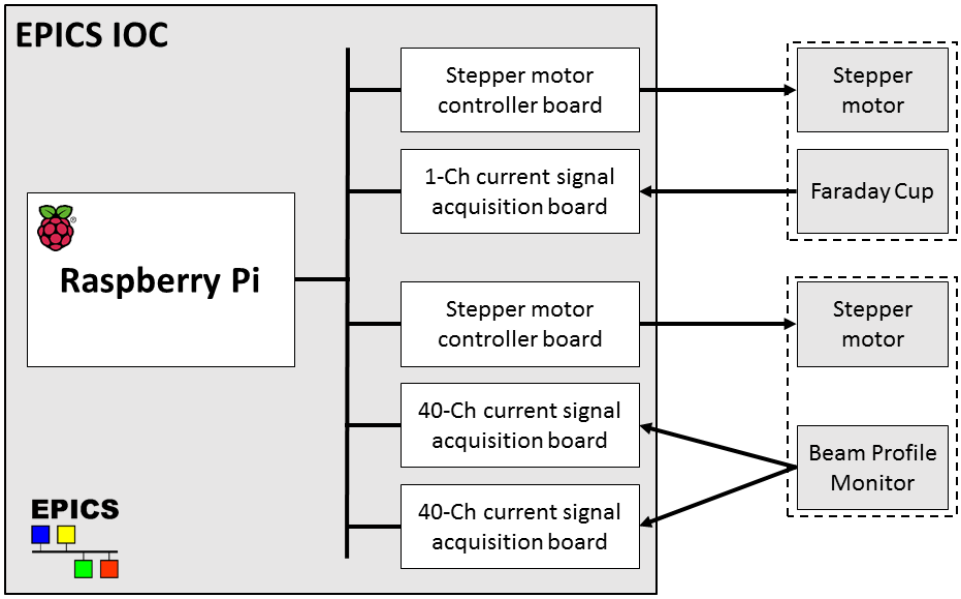


Figure 3.43. Block diagram of the beam diagnostic IOC.

Once that the data are acquired, an exponential conversion is performed in order to calculate the input current. This value is then written to the corresponding EPICS interface record through the EPICS CA put function.

Independently of the acquisition process, the two stepper motor controller boards are handled using digital outputs on the GPIO port. EPICS CA monitors are set on the position records. Each time a new position is written, an interrupt function is called and the requested new position is written into a buffer and flags are set indicating the presence of a new request. On the main loop, when these flags are detected, the motors are moved accordingly using the corresponding GPIO pins.

The user interface for the beam diagnostic IOC was developed using CSS, and it was integrated to the main Front-end control panel [31]. The operator can handle and read the values read from both detectors through this interface. A screenshot is shown in Figure 3.46.

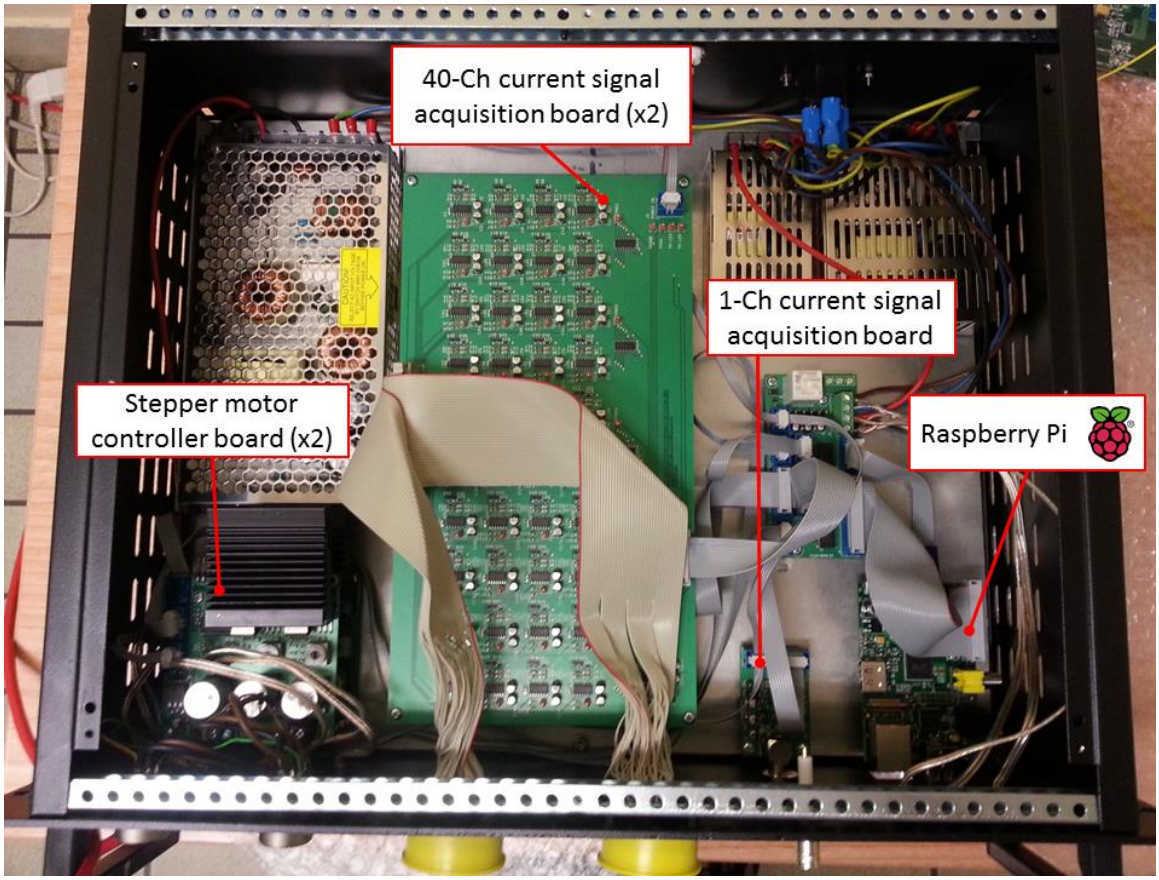


Figure 3.44. Beam diagnostic IOC.

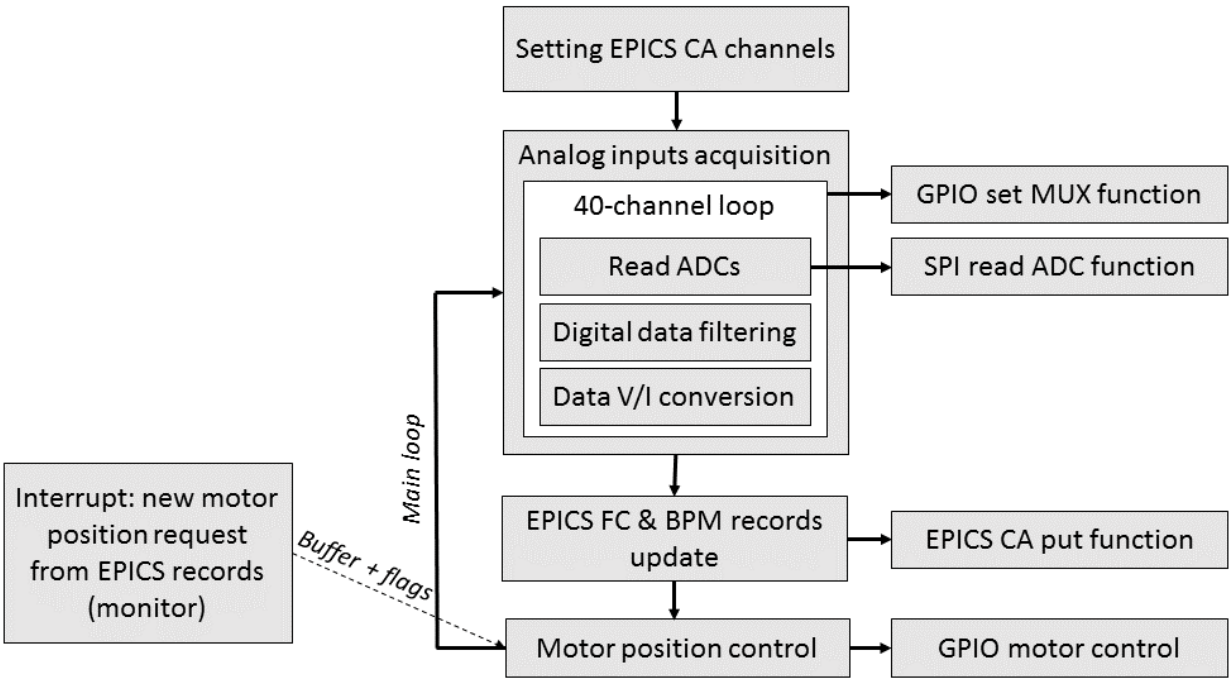


Figure 3.45. Beam diagnostic IOC interface driver flow diagram.

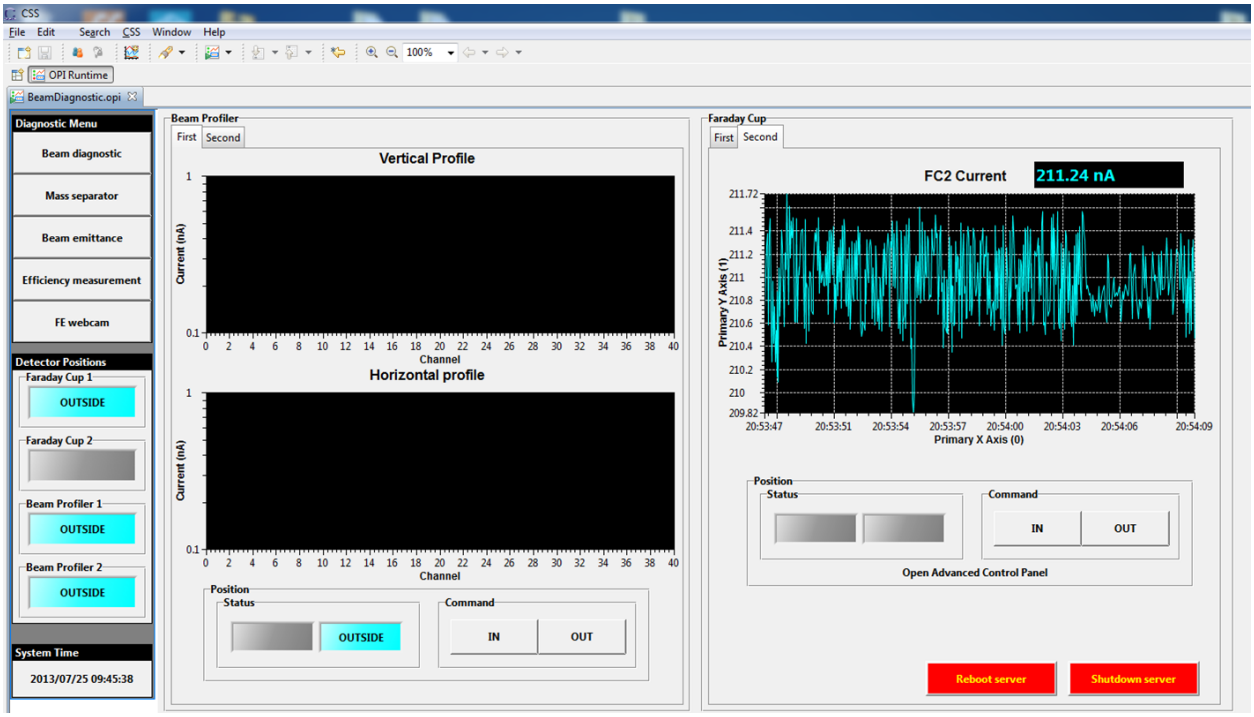


Figure 3.46. Beam diagnostic IOC user interface. (Developed using CSS).

3.4.2 Mass Separator

The front-end mass separator (Figure 3.47) consists on an electrostatic dipole and a magnetic coil that produce an electric and a magnetic field inside a vacuum chamber on the path of the beam. Two high voltage power supplies drive the dipole while the coil is driven by a high current power supply. Additionally, the magnetic field is measured using a hall effect probe. The three power supplies and the hall effect probe measurement instrument have a serial port (UART RS232) communication interface.

An IOC was designed for controlling the mass separator instrumentation. It uses a quad USB-to-serial converter (USB-COM485-PLUS4 from FTDI Chip [32]). This provide four serial port to the IOC in order to communicate with the four instruments.

Moreover, a series of temperature sensors (DS1822 from Maxim Integrated [33]) were installed on the magnetic coil and high current power supply water cooling circuits in order to monitor the temperature of the system (Figure 3.48). The sensors are read directly though the GPIO port using the 1-wire interface. On the Linux operating system, 1-wire devices are access through the 1-Wire File System (OWFS). This method allows 1-wire devices to appear as files on a directory.

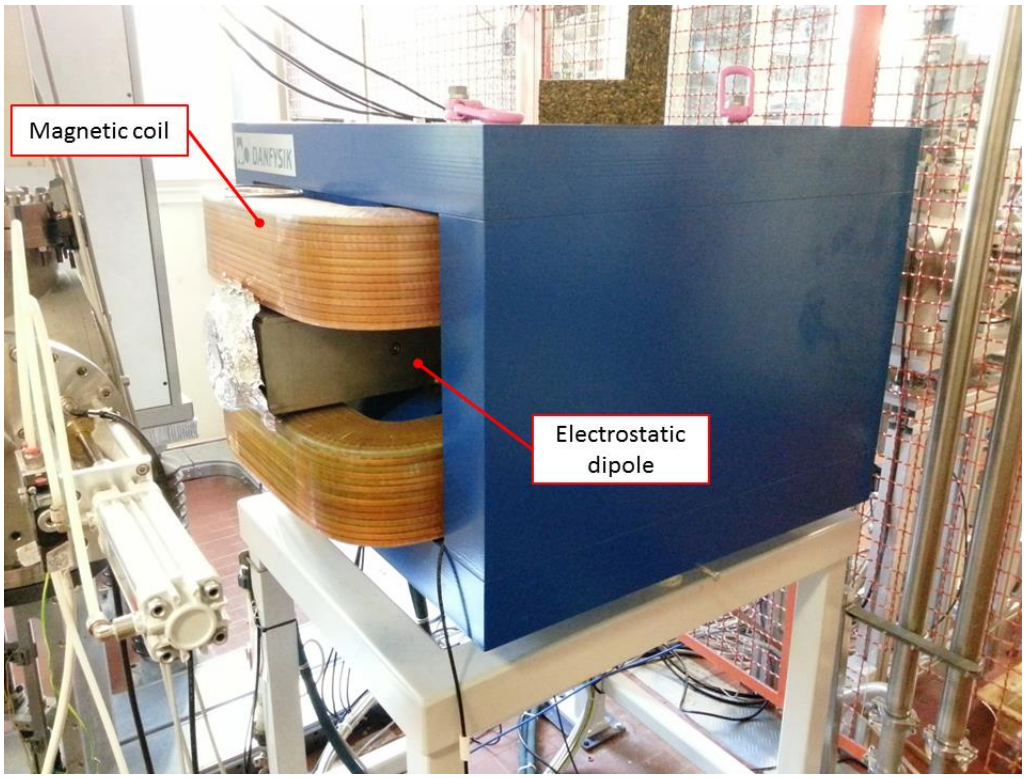


Figure 3.47. Mass separator at the SPES off-line laboratory.

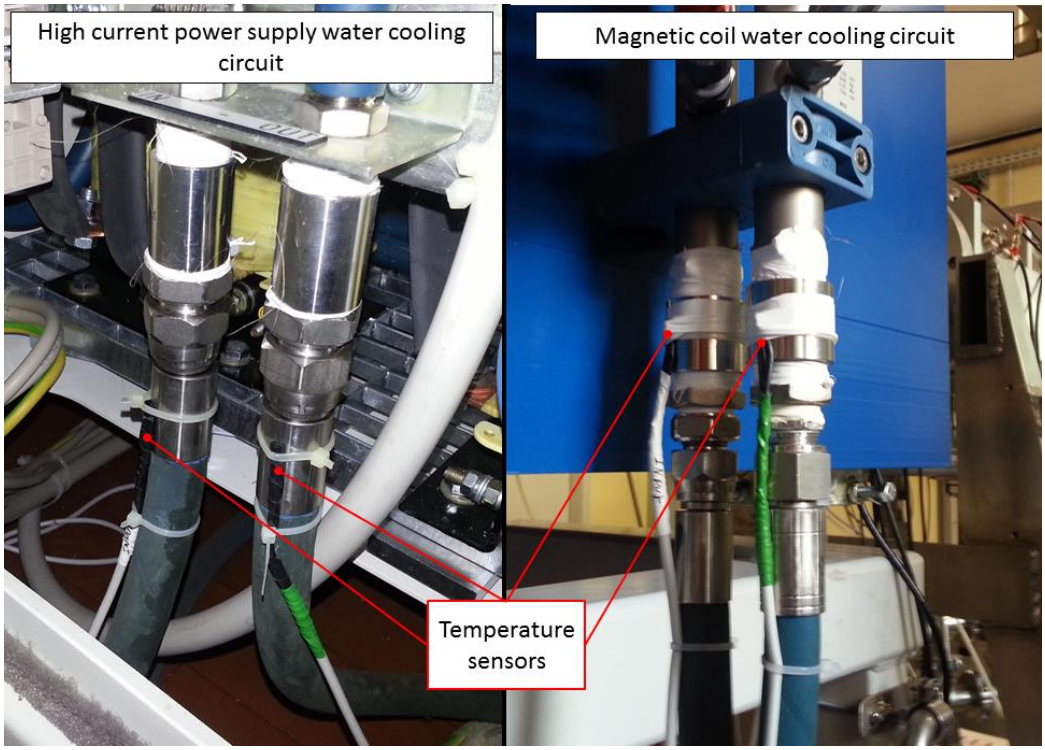


Figure 3.48. Temperature sensors installed on the mass separator system.

On Figure 3.49 is presents a block diagram of the IOC as described above, while Figure 3.50 shows a picture of the final implemented IOC.

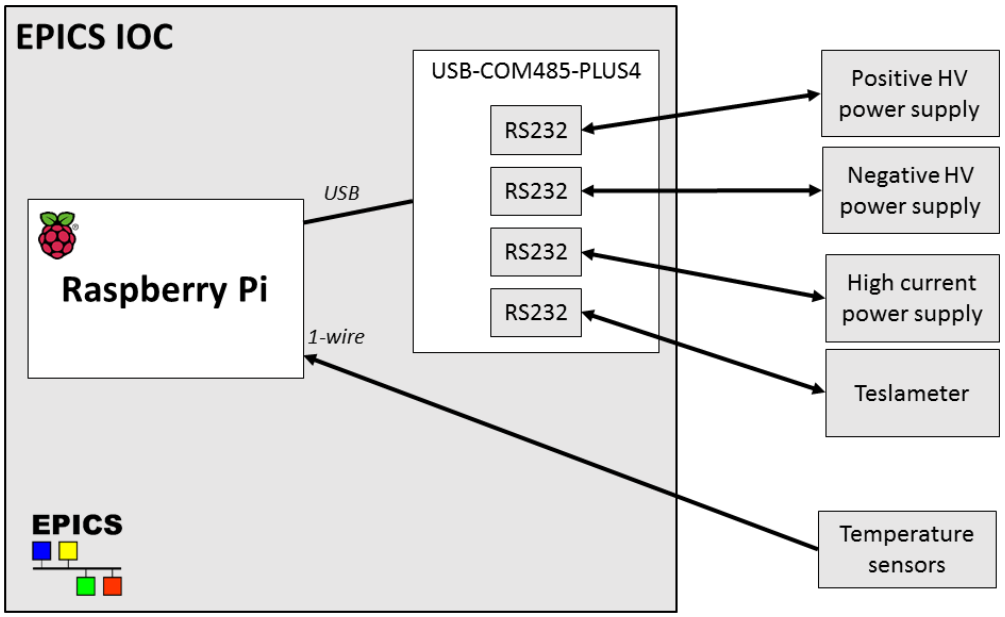


Figure 3.49. Block diagram of the IOC implemented for controlling the mass separator.

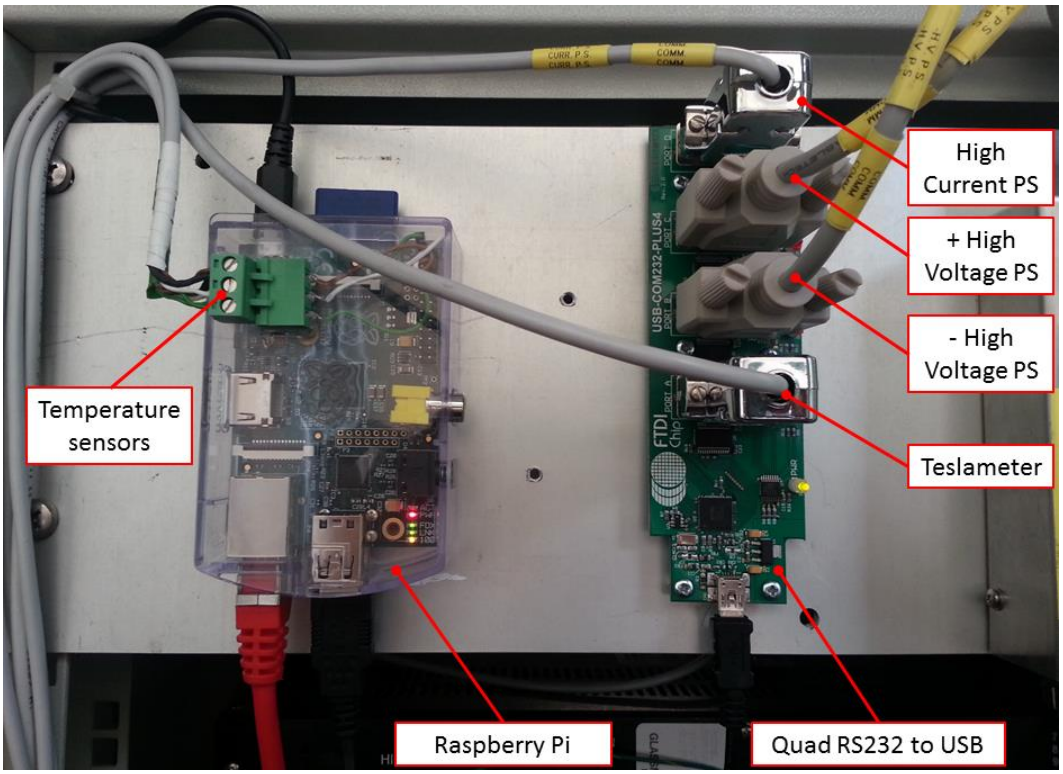


Figure 3.50. IOC implemented for controlling the mass separator.

For this application, the interface driver consist only on reading the temperature values from the corresponding sensor files. The flow diagram of this interface is presented on Figure 3.51, while the source code is reported on the appendixes.

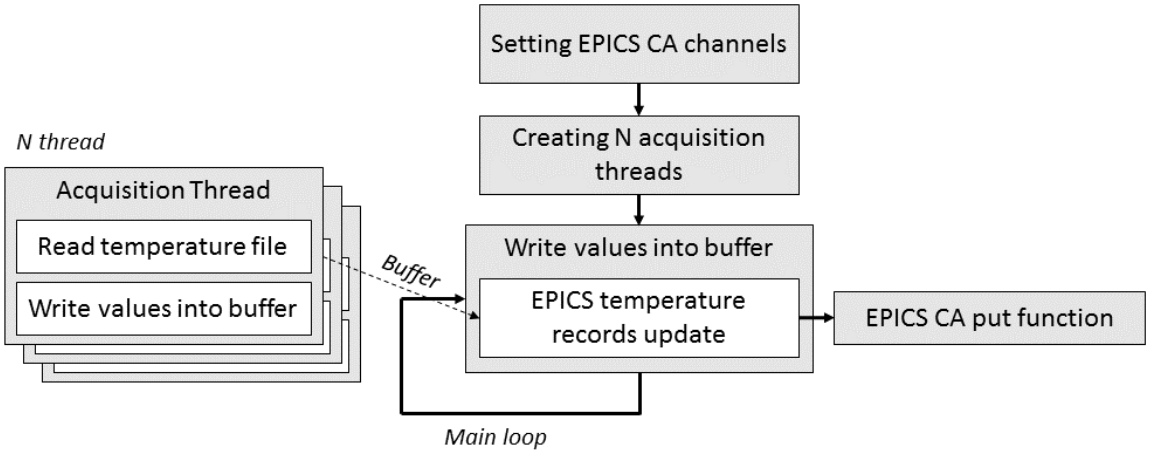


Figure 3.51. Mass separator IOC interface driver flow diagram. (In this case, the interface read only the temperature sensors).

On the interface driver, an independent thread is created for each temperature sensor. Inside the thread, the corresponding sensor file is open and continuously read. The values are written into buffers that are checked on main loop of the main program. When a new value is detected, it is then written to the corresponding EPICS interface record, using the EPICS CA put function.

For controlling the rest of the instrumentation, the EPICS application was developed using the EPICS device support StreamDevice [26], very especially useful for devices with stream based communication interface, as it is the case of the instrumentation present on this system.

For this particular application, a scan functions was implemented in the EPICS IOC application. This function allows the user to automatically perform a mass scanning procedure, in order to obtain information about the composition of the ion beam.

The scan function vary the coil current on a defined interval, using a constant electric field. The beam current is measured at each step. The ratio between the electric and magnetic field defines the mass of the element that it is allowed to pass through the filter, into the Faraday Cup. At the end of the procedure, a plot of the beam current versus element mass is presented to the operator. For developing this application, the EPICS SSCAN record was used [34].

The user interface for the mass separator IOC was developed using CSS, and integrated to the main Front-end control panel [31]. A screenshot of the interface, corresponding to the mass scanning function, is presented on Figure 3.52.

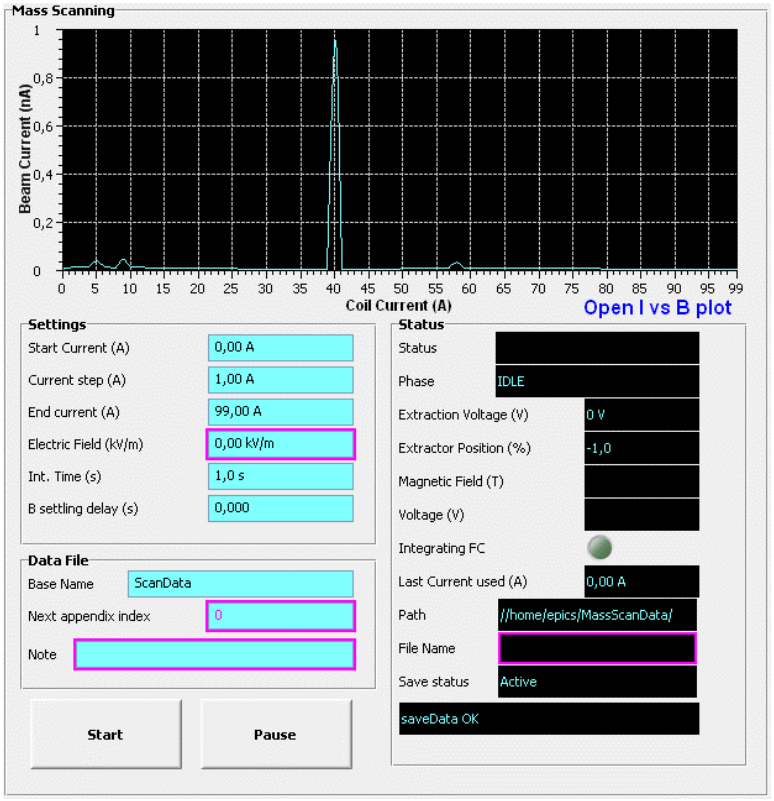


Figure 3.52. User interface developed for the mass separator IOC. (Mass scanning screen)

3.4.3 PLC communication interface and vacuum instrumentation data acquisition

In the off-line laboratory, two Programmable Logic Controllers (PLCs - BMXP342020 from the Schneider Electrics Modicon M340 family) are in charge of the control of the vacuum system and the safety control system [35]. The vacuum control PLC is shown on Figure 3.53, as an example.

Pfeiffer TPG300 vacuum measurements instrument (Figure 3.54) are installed on the vacuum control system. They have a serial port (UART RS232) for configuration and data acquisition.

An IOC was designed with a dual function, acts as interface between the PLC network and the EPICS network, and acquire the data from four vacuum measurement instruments. This IOC uses a USB-to-Ethernet converter (DUB-E100 from D-Link [36]) in order to add a second Ethernet interface to the system

dedicated to communicate with the PLC; and a quad USB-to-serial converter (USB-COM485-PLUS4 from FTDI Chip [32]) which provide four serial port to the IOC in order to communicate with the four vacuum instruments.

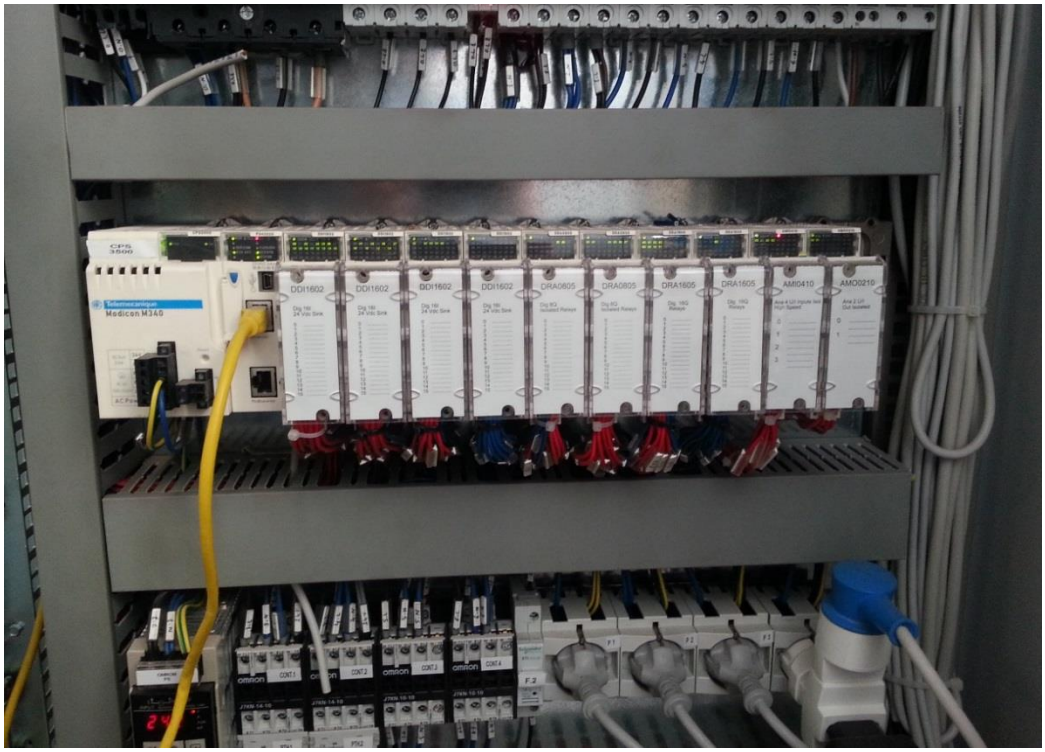


Figure 3.53. Vacuum control PLC at the off-line laboratory.

Figure 3.54. Vacuum measure instrument used at the off-line laboratory.

The IOC communicates with the PLC using the EPICS driver support for Modbus [37], the native communication protocol of the M340 PLCs. The IOC reads some defined variables from the memory of the PLC and writes them into EPICS records. Likewise, the content of some EPICS record is transferred to the memory of the PLC. On the other hand, the EPICS application for the vacuum value acquisition was developed using the EPICS device support StreamDevice [26].

In this case, as no custom expansion board was used, no interface driver was implemented. The Ethernet interface, as well as the serial ports, are automatically detected and made available by the operating system.

A block diagram of this IOC is presented on Figure 3.55, while Figure 3.56 shows a picture of the final implemented IOC.

The user interface for this IOC was developed using CSS, and it was integrated to the main Front-end control panel [31]. From this interface, the operator can see status of the PLC systems, as well as read the vacuum levels and change settings on the vacuum instruments. A screenshot of the vacuum interface is presented on Figure 3.57, while Figure 3.58 shows the vacuum instrument acquisition screen.

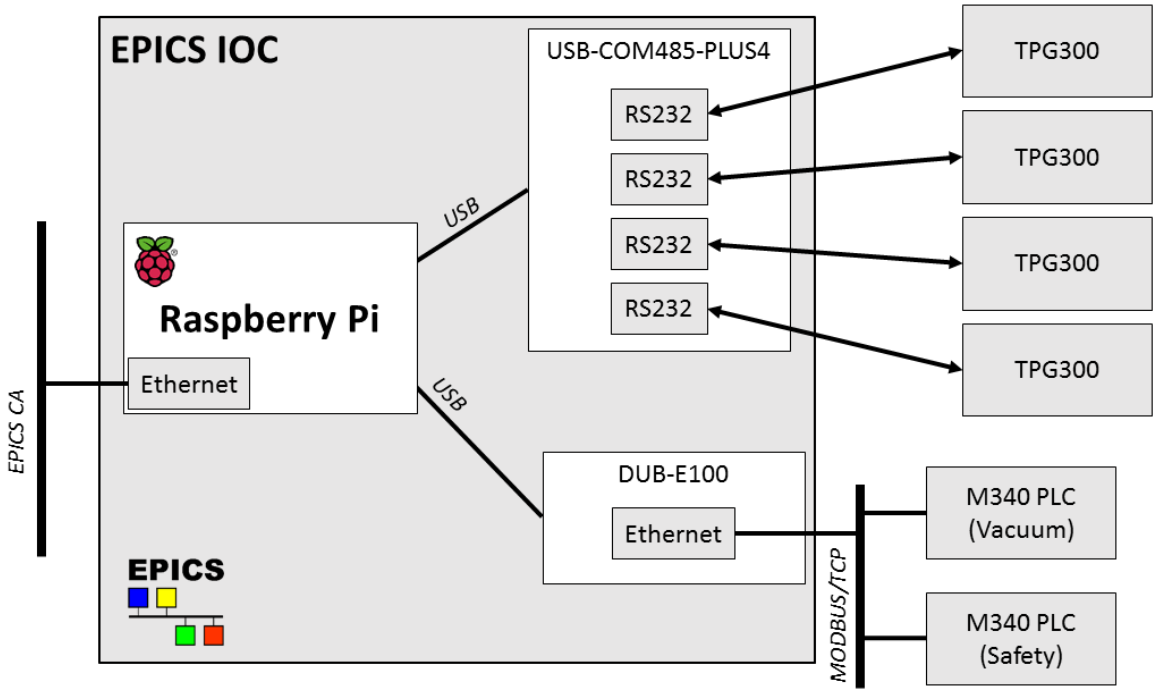


Figure 3.55. Block diagram of the IOC implemented for the PLC interface to EPICS and for the vacuum measurement data acquisition.

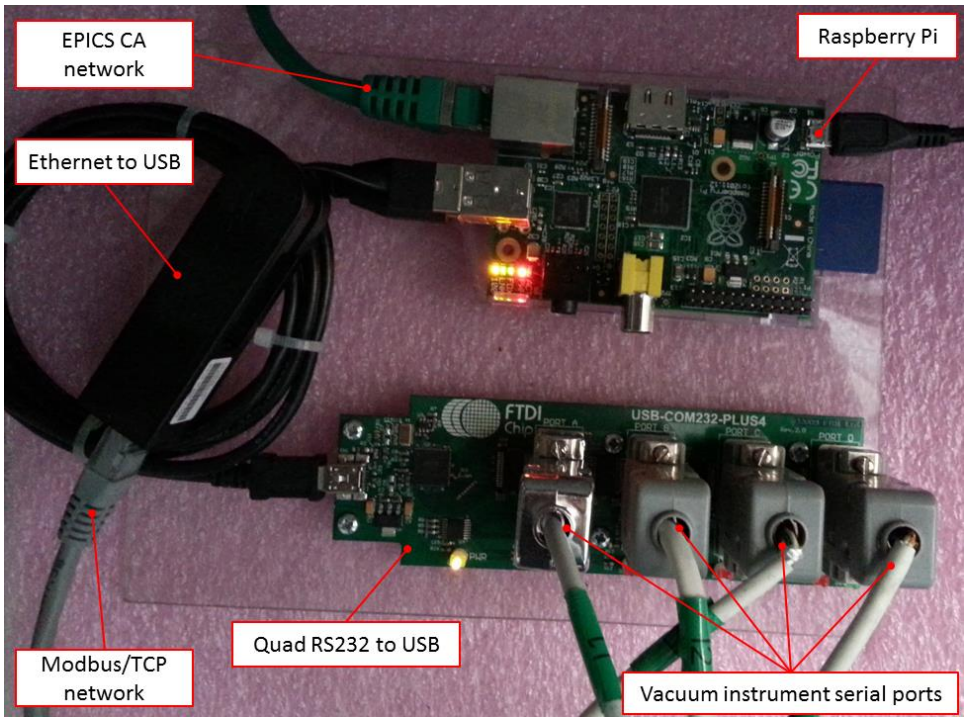


Figure 3.56. IOC implemented for the PLC interface to EPICS and for the vacuum measurement data acquisition

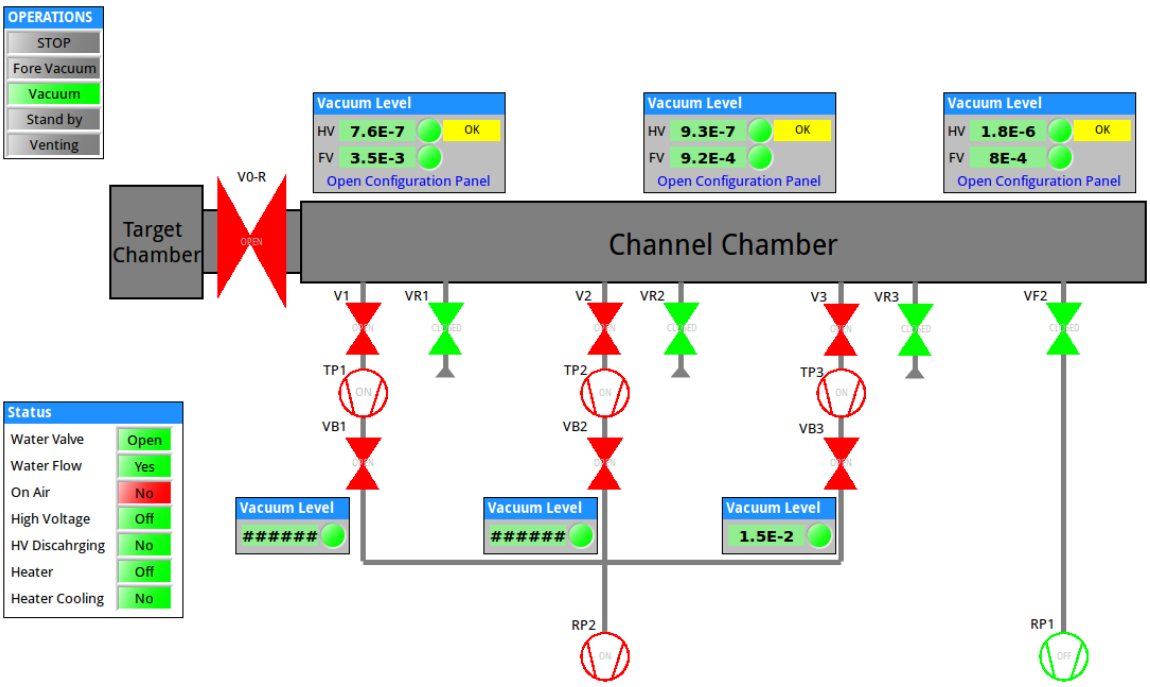


Figure 3.57. User interface developed for the vacuum control PLC communication IOC.

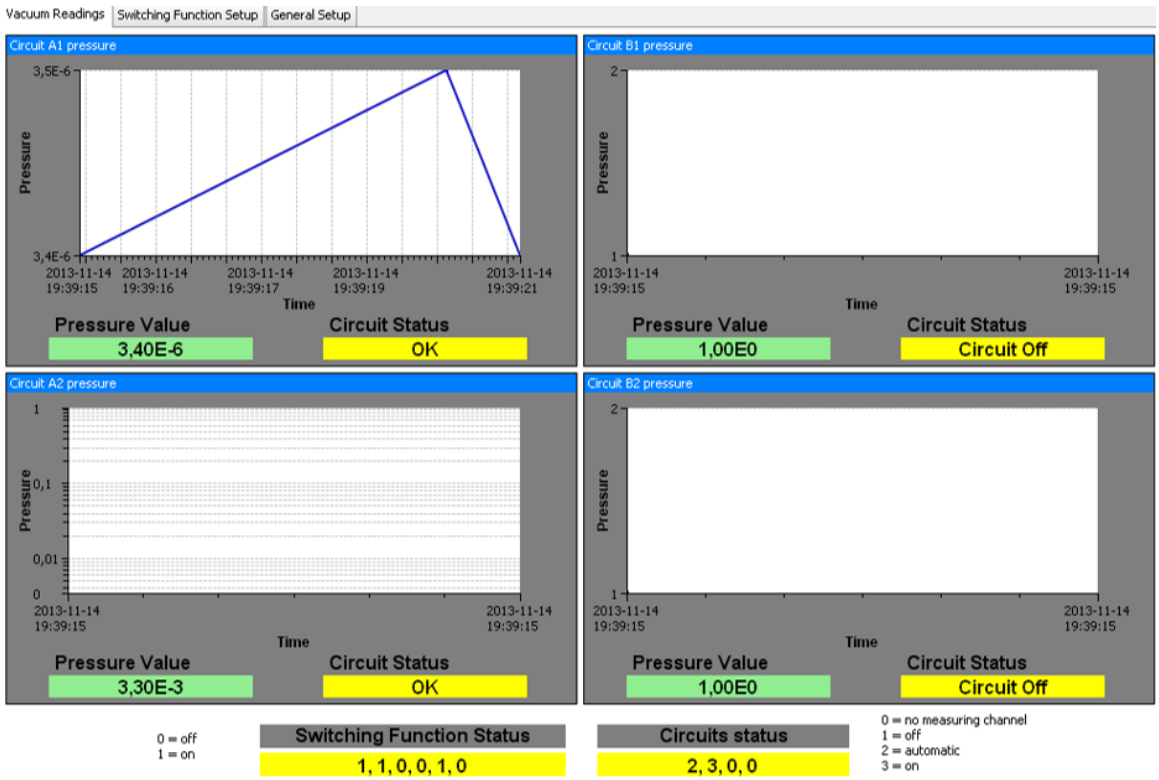


Figure 3.58. User interface developed for the vacuum data acquisition IOC.

3.5 Conclusions

A new kind of EPICS IOC was presented. It is based on the low-cost computer board Raspberry Pi in conjunction with standard USB converters and tailored IO expansion boards. These IOCs were tested and implemented in the SPES off-line laboratory. Four types of expansion boards where developed in function of the applications found on the off-line laboratory: a 1-channel, current-signal acquisition board; a 40-channel, current-signal acquisition board; a stepper motor controller board; and a general-purpose IO board.

Using these boards, three different IOCs were developed and used to implement four different control systems: the beam diagnostic data acquisition system, the mass separator control system, the vacuum measurement acquisition system, and the PLC-to-EPICS interface.

Tests were carried out in order to determinate the performance of the IOC. It was shown that the ADC inputs have an ENOB between 13.93 and 14.86 bits, for signal frequencies lower than 35 Hz. Additionally, the bias error for DC signals was lower than 60 mV.

Similarly, the DAC output test results showed that their estimated resolution is around 14.9 bits, with less than 16 mV of bias error, for DC output signals.

For the case of input current signals, widely presented on the beam diagnostic system, the IOC is capable of acquiring current from 100pA until 3.5mA, with less than 2% of bias error for most cases, using a logarithm current-to-voltage conversion. It was shown the negative effect of noise on this kind of solution. Although on the test applications this effect was still acceptable, it was presented a possible solution for possible future applications.

It was indicated that the maximum achievable acquisition rate is around 83.8 Samples/s using a high CPU load of around 90%. However, for most application at the off-line laboratory, a slower acquisition rate of 13.9 Samples/s can be use, significantly reducing the CPU load to 16.5%.

These IOCs represent a very flexible and easy to adapt and customize solution for EPICS control system applications. They can be implemented on a large number of different applications. Moreover, this solution can be very cost effective due to the low cost of the components used, while maintaining performance suitable for most application at the SPES project.

Chapter 4

The standard EPICS IOC for the LNL

4.1 Introduction

The arrival of the SPES project to the LNL has triggered an upgrade campaign of the control system of the accelerator complex. The reason for this is the fact that the beam produced at SPES will be injected on ALPI, the LNL superconductive LINAC. This implies that both, SPES's and ALPI's control systems must be able to interact with each other. Due to the complexity of both tasks -development of the new control system for SPES and upgrading the existing control system for ALPI- it was almost inevitable to conclude that new developments must be applicable to both systems.

On the other hand, EPICS has been chosen as the standard framework for developing the new control system for SPES [11] [10]. The architecture of this control system consists on a distributed series of controllers (IOCs) interconnected using the EPICS Channel Access protocol.

In that sense, one of the decision taken was to develop a custom EPICS IOC that were able to satisfy all the common needs of the control systems to be developed. The aim of this IOC is to become a standard construction block for developing all the future control systems at LNL. With the use of this IOC at LNL the main goals are standardization of hardware and software as well as system interoperability.

The developing of this IOC it is a very time consuming task, and approximately two years are expected for the first version to be ready. Therefore, some mockups and prototypes have been developed in order to validate the hardware platform, testing them on real applications.

On this chapter the new IOC is described, and the results obtained using prototypes on some applications at LNL are presented.

4.2 IOC Description

The IOC is intended as a standard system with the necessary interfaces in order to be able to control the instrumentation present on all the foreseen system at LNL. EPICS tools will be used for software

development in a standard way. For each application, a custom-made control algorithm will be developed under the framework.

The COM (Computer-on-Modules) form factor has been chosen as the hardware platform for the IOC. COMs are highly integrated and compact PCs that include core CPU, memory and common IO interfaces found on standards PCs (as USB, SATA, audio, video, Ethernet, PCI, PCI express, among others). In particular, the type 6 COM Express standard was selected [38]. It defines physical sizes, interconnections, and thermal interfaces. Figure 4.1 shows a picture of a model of type 6 COM Express and Figure 4.2 shows its functional diagram.

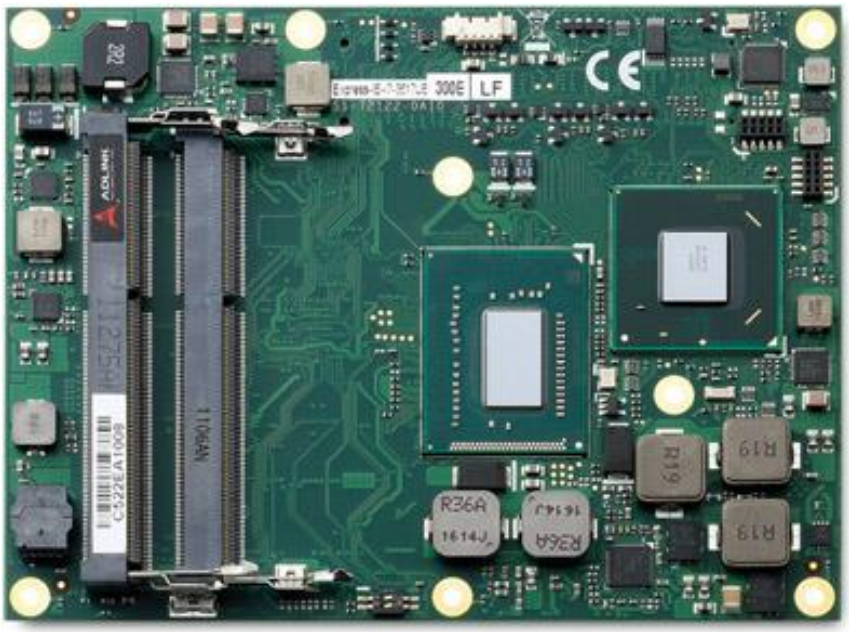


Figure 4.1. Computer-on-Module board *(Adlink's type 6 Express-IB)* [39].

The COM will be installed on a custom carrier board, designed at LNL. This carrier board will contained peripheral devices such as digital IO, ADCs, DACs, stepper motor controllers, and communication interfaces, among others, connected to the COM. Its design is under carefully considerations in order to cover all the common needs for the current and the foreseen future control systems at LNL. Essentially, it will contained: eight 16-bits, 1 MSample/s and two 24-bits, 312 KSample/s ADCs; eight 16-bits and two 24-bits DACs; eight RS232/RS485 serial ports; eight stepper motor controllers; one 1-wire interface; one CAN bus interface; and standard PC services as VGA, USB, Sata, PCIe, Audio, Ethernet, WiFi, SD among others. PoE (Power over Ethernet) will be available for simple, low power applications.

Some interfaces are already available directly from the COM. Others will be controlled using USB ports. Many others will be controlled by a FPGA that will communicate with the COM through a PICe (PIC Express) lane. Figure 4.3 shows a simplify block diagram of the IOC.

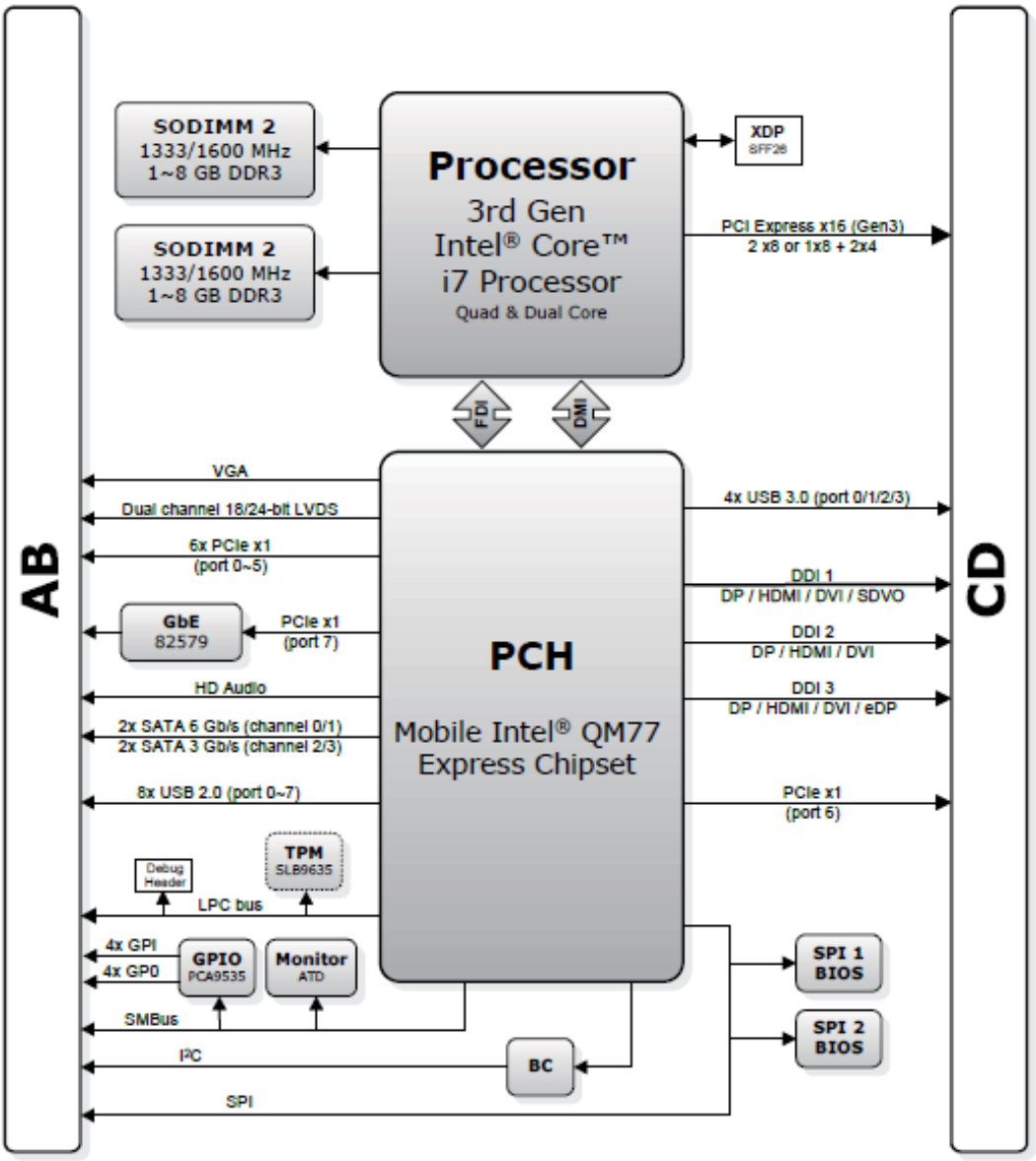


Figure 4.2. COM functional diagram (Adlink's type 6 Express-IB) [39].

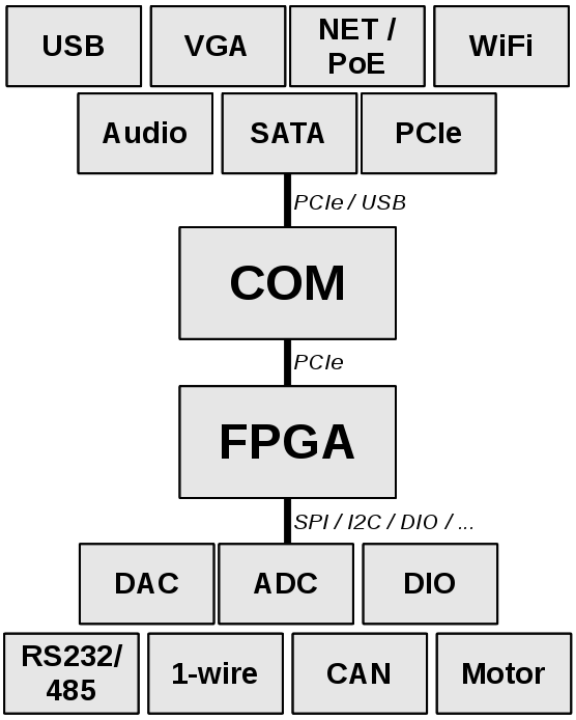


Figure 4.3. IOC simplify block diagram (services implemented on the IOC. Some taken directly from the COM, while other through a USB port or FPGA).

4.3 IOC Prototypes Implementation at LNL

In order to test the validity of the COM as hardware platform for IOC developments, prototypes were developed using commercial devices. These prototypes, at the same time, allowed developing the software part of control systems.

The developed software is mostly independent from the hardware implementation, due the fact that the selected COM uses Intel CPUs allowing the use of a standard Linux environment for the implementation. In this way, once the custom IOC will be ready, the software would be easy transport to the new hardware platform.

On the other hand, prototypes were used to develop and implement diverse control systems at LNL. This allowed testing them under real operative conditions, as well as to develop critical control system without waiting for the final custom IOC to be available.

4.3.1 Hardware Architecture

The prototypes have been developed using Adlink [40] Type 6 COM Express module on a generic carrier board. Commercial Digital IO, ADCs and DACs PCIe boards were installed on the carrier board in order to provide IO capabilities to the IOC.

Adlink's DAQe-2214 boards were used as ADC inputs. Each board uses one x1 PCIe interface and provides 16 16-bits, 250 KSample/s, +/-10 V channels. It also provides 24 digital IO channels.

In the same way, Adlink's PCIe-6216 boards were used as DAC outputs. Each board provides 16 16-bits, +/-10 V channels, using a 1x PCIe line.

On the appendixes, the technical specification of all the boards used can be found.

On Figure 4.4 it is shown a picture of a prototype with the COM on a generic carrier board with a DAQe-2214 and a PICe-6216 boars.

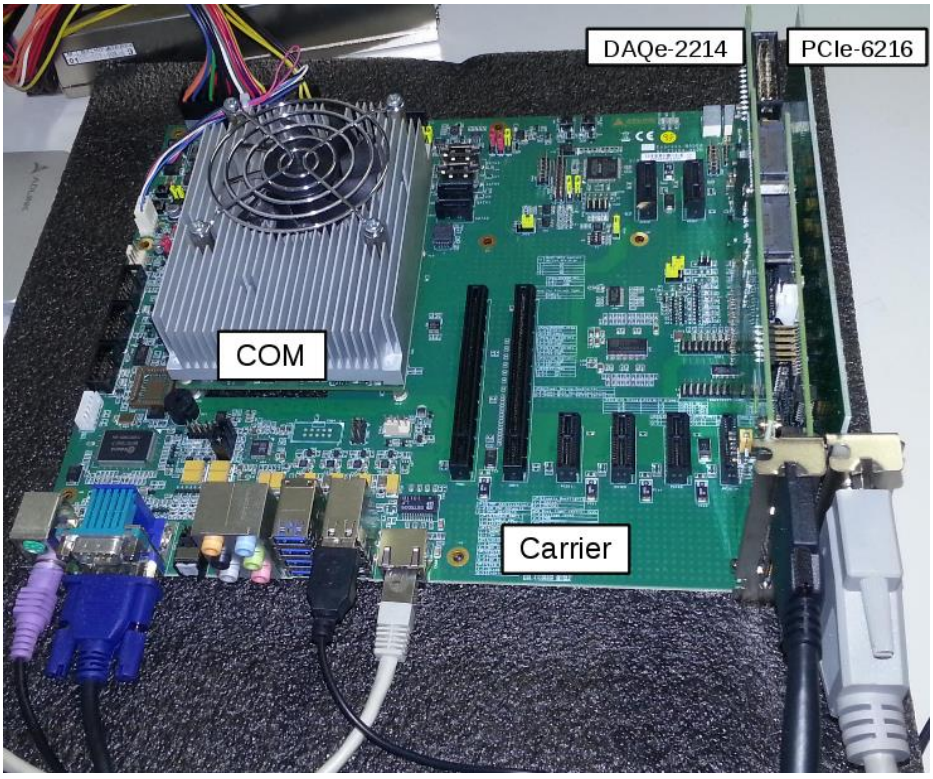


Figure 4.4. IOC Prototype (The COM is installed on a generic carrier board with a DAQe-2214 and a PCIe-6216 boards).

4.3.2 Software Implementation

Adlink provides linux drivers for the expansion boards used on the prototypes. The drivers provide common APIs (Application Programming Interfaces).

In order to interface the EPICS IOC device supports to Adlink drivers, asynDriver [25] was used. AsynDriver is a general purpose facility for interfacing device specific code to low level communication drivers. With this facility, interfacing drivers were developed using C and C++ programming languages.

The analog input acquisition is performed using a periodic scanning function on the asynDriver software. This function reads the ADC inputs and do callbacks updating the recently read values into the parameter library. In this way, the EPICS record attached to an analog input receives an interruption request each time a new value is available. Inside the scanning function loop, an adjustable delay was introduced in order to allow changing the sample rate of the inputs. The same procedure is done for digital inputs acquisition. Figure 4.5 shows a flow diagram of this acquisition algorithm.

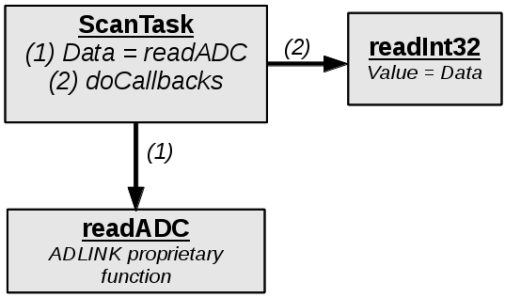


Figure 4.5. Analog input acquisition algorithm (The periodic function ScanTask first calls readADC which uses the ADLINK proprietary API. Then it calls callback functions which updates the EPICS records through readInt32).

On the other hand, for the analog outputs, a function that writes the DAC channels was implemented on the asynDriver software. This function is called each time a new value is written to an EPICS record attached to an analog output, passing it the written value. The same procedure applies for digital outputs. Figure 4.6 shows a flow diagram of this output update algorithm.

On the appendixes is presented the source code of the developed interface drivers.

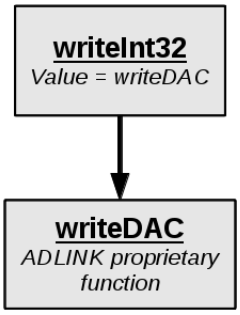


Figure 4.6. Analog output update algorithm (When a new value is written to the EPICS record, writeInt32 calls writeDAC which uses the ADLINK proprietary API).

4.4 Experimental test and results

A soft IOC was deployed on a prototype IOC in order to test its performance. The COM used on the prototype was the model "Express-IB-i3-3120ME" which has an Intel Core i3-3120ME CPU at 2.4GHz and 4Gb of RAM. The COM was installed on a generic carrier board model "Express-BASE6" with a DAQ-2214 and a PCIe-6216 boards. A picture of this prototype is shown on Figure 4.4.

Fedora Core 17 with kernel versions 3.3.4-5.fc17.x86_64 was installed on the IOC with the board drivers installed on the system. EPICS R3.14.12 and asynDriver R4-23 were also compiled and installed. On this system, an EPICS IOC application consisting only on of analog inputs and analog outputs records was linked to the physical ADCs and DACs using the asynDriver software explained on the previous chapters.

This IOC application was used to compute the resolution of the ADC channels and the CPU usage by the application. In the following chapters, the experimental tests will be explained and the obtained results will be present for each case.

4.4.1 Acquisition rate and CPU usage performance

This test was performed in order to estimate the maximum sample rate that be achieved with this IOC, as well as the correlation between the sample rate and CPU usage by the application. The test consisted on measure the CPU usage by the EPICS application (which include the driver interface) for different samples periods.

The sample period was changed using the delay inserted on the scanning function on the asynDriver described on chapter 4.3.2. Four different scanning rate where targeted. The first one correspond to the

faster scanning rate achievable, setting the delay to zero. The latest one was set to around 10 Samples/s, which it is a usual value used as sample frequency on several applications at LNL. Finally, other two rates were selected in between these values, i.e. 200 Samples/s and 1000 Samples/s. The test was perfomed both, only acquiring a single analog input channel, and acquiring all 16 analog input channels. The results are presented on Table 4.1 and on Figure 4.7.

Table 4.1. CPU usage performance results.

Number of channels	Sample period average (ms)	Sample period standard deviation (ms)	Sample frequency (S/s)	EPICS IOC application CPU usage (%)
1	0.23	0.38	4397.54	100
	1.12	0.02	893.77	18
	18.32	19.04	134.75	3
	348.79	359.96	7.91	0.1
16	0.90	0.35	1111.32	100
	2.17	0.75	460.87	46.5
	7.42	2.61	54.59	15
	126.38	43.91	2.87	1

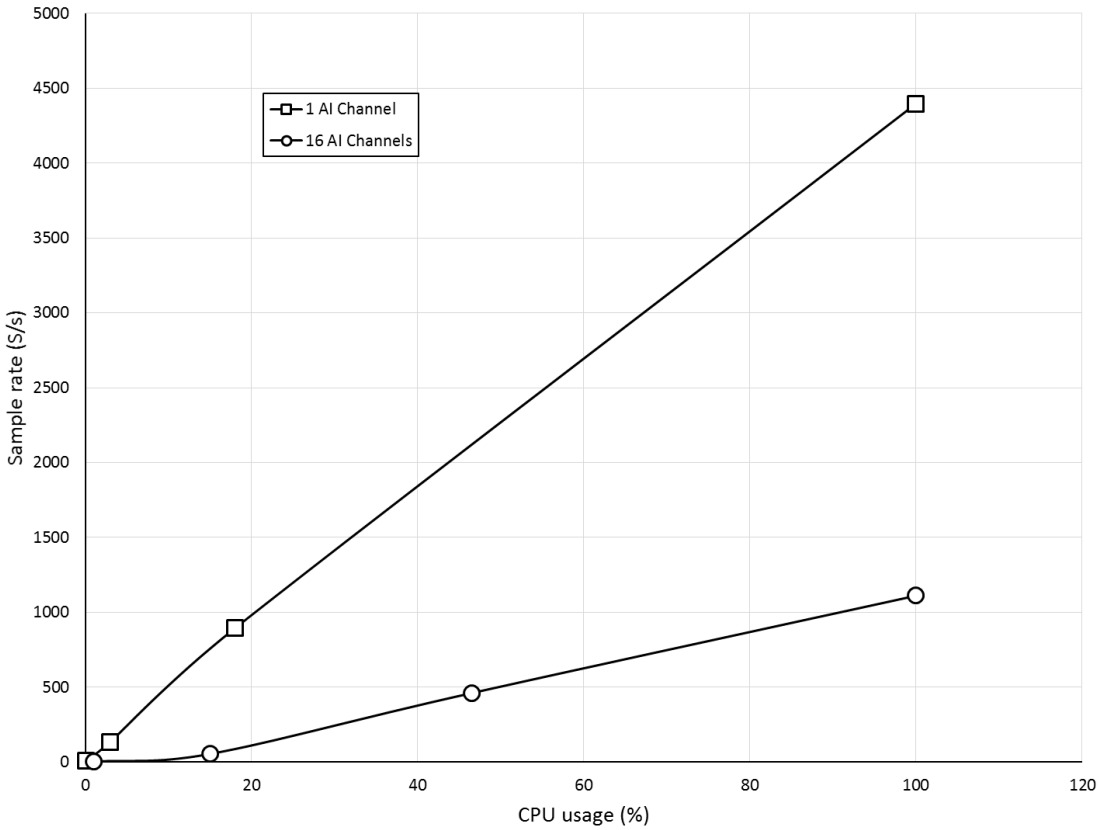


Figure 4.7. CPU usage versus sample frequency results.

The results show that the highest sample rate that can be reached scanning only one channel is almost 4.4 KSample/s, while when scanning all 16 channel it is around 1.1 KSample/s. However, for reaching these rates, the application uses all the available CPU, due to the fact that the acquisition loop continuously read the input channels, without any delay.

The CPU usage decreases drastically when a delay is set, lowering also of course the acquisition rates. Furthermore, it is evident that more CPU is needed for achieving higher rates on all channel respect to the case with only one channel, as it is expected.

Particularly, for application when only one channel is needed, the second obtained value is a good compromise, achieving a sample rate of almost 900 Sample/s using only 18% of CPU. For application when all channels are needed, the third case is a good choice, using only 15% of CPU reaching rates of almost 55 Samples/s. Finally, low acquisition rates of some units of Sample/s, typically used on the LNL applications, the CPU usage is negligible.

The CPU usage can be considered high for the moderate scanning rate obtains (less than 5 KSamples/s). This happens because the scan function on the asynDriver application calls the Adlink's driver API each time a new value is requested, which is very inefficient. However, on the custom IOC design, the ADC inputs will be handled by the FPGA, which will hold the data on a buffer. In this way, the driver application will reduce the number of request to the hardware, reducing at the same time the CPU usage, even at higher scanning frequencies.

4.4.2 Analog-to-Digital performance

In order to determinate the performance of the ADC inputs, test were performed for estimating its effective number of bits (ENOB) of resolution. The procedure is identical to the one presented previously, on chapter 3.3.2.

As on chapter 3.3.2, the ADC input range was set to +/-10 V, with a 90% full-scale input sine wave signal. For each test, more than 205888 samples were taken, as expressed by equation (3.6). For the test, only one input channel was used, at the second higher sample rate report on the Table 4.1 presented on the previous chapter, which it is around 890 Sample/s.

The method for calculation the ENOB, defined by equation (3.5), is the one defined on [27], i.e. calculating the FFT of the acquired data, without the DC component. Then obtaining the signal power (signal

frequency bin) and the noise and distortion power (all frequency bin except zero and the signal's one) for calculating the SINAD expressed by equation (3.4).

The frequency of the input signal was 400 Hz, chosen in order to be lower that the Nyquist frequency [28]. The calculation were performed for other six lower frequencies: 200 Hz, 100 Hz, 50 Hz, 10 Hz, 5 Hz and 1 Hz.

The input sine wave signals were produce using a Hewlett Packard 33120A waveform generator. The samples values were acquired and store using EPICS CA tools. Then, the data was process using Scilab [29], calculating the sample average rate, the signal input average frequency, the FTT and the SINAD and ENOB defined on equations (3.5) and (3.4) respectively. The Scilab code used on these calculations is reported on the appendixes.

The power spectrum obtained calculating the FFT for the 400Hz input signal is presented on Figure 4.8, as an example. Similar results were obtained for the other cases. The calculate results are presented on Table 4.2, while a plot of the obtained ENOB for the different input signal frequencies is presented on Figure 4.9.

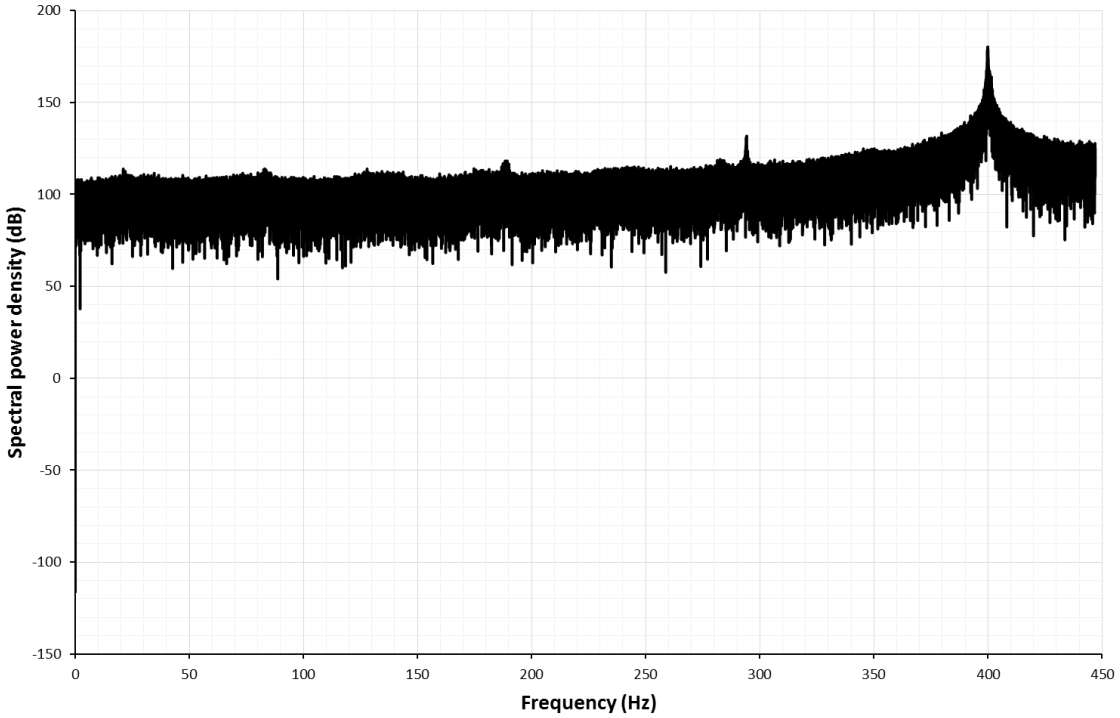


Figure 4.8. Power spectrum for a 400Hz sine wave signal. (Sample rate = 893 S/s, ADC range = +/-10V, input signal 90% full-scale).

Table 4.2. ADC resolution performance results.

Reference Frequency (Hz)	Measured Input Frequency (Hz)	Measure Sample Rate (S/s)	SINAD (dB)	ENOB (bits)
1	1.00	892.14	112.03	18.47
5	5.00	893.82	116.01	19.13
10	10.00	893.96	110.36	18.19
50	49.99	894.04	105.20	17.33
100	99.96	893.80	105.06	17.31
200	199.92	893.83	101.23	16.68
400	399.88	893.92	93.87	15.45

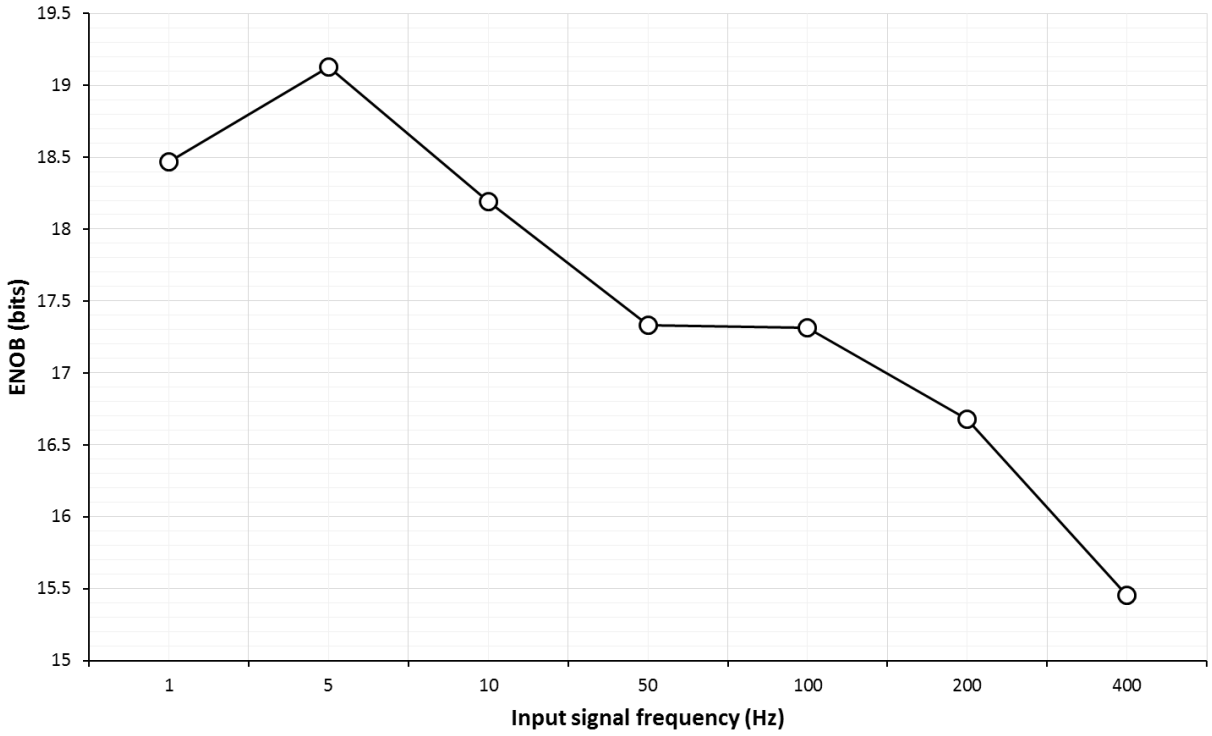


Figure 4.9. Estimated number of bits of resolution obtained for the analog inputs.

It can be see that for low frequency signals (lower than 150 Hz approximately) the obtained ENOB is higher than 17 bits, reaching even 19 bits. For a signal of 400 Hz, the ENOB is around 15.5 bits.

The reason for accomplishing resolution greater the 16 bits (the ADC resolution) for the lower frequency signals is the effect of oversampling the input signal. When the sampling rate is greater that the Nyquist sampling rate by a factor *N*, then the SINAD is improved by a factor [28],

$$SNR_{oversamplig} = 10 \cdot log(N) \qquad (4.1)$$

This equation implies that each time the oversampling factor is quadrupled, the SINAD increases 6 dB and, according to equation (3.5), an extra bit of ENOB is gained.

From the results on Table 4.2, for the 1 Hz inputs signal, the oversampling factor is N=446, which correspond to an increase on SINAD, according to equation (4.1), of 26.5 dB and, according to equation (3.5), to 4.4 additional bits of ENOB. Thus, for this case the resulting ENOB without the effect of the oversampling would have been of only 14 bits. For the signal at 400 Hz, the oversampling is near 1 (as the sampling frequency is only slightly greater than the Nyquist frequency), hence the ENOB is minor than 16 bits.

A second test was performed in order to estimate the bias error for DC input signals. This test consisted on applying constant values to the analog inputs of the board and archive a set of data for statistic analysis.

A Tektronik PWS4305 programmable DC power supply was used for producing seven constant values, from -7.5 V to 7.5 V with steps of 2.5 V. For each input value, according to equation (3.6), more than 205888 points were acquired and archived using EPICS CA tools.

The ADC channel has a resolution of 16-bits and its input range was set to +/-10 V. For this conditions, the resulting ADC word equivalent to a given X volt inputs is expressed by the equations (3.7), presented previously, on chapter 3.3.2.

Table 4.3 presents average, standard deviation, minimum, maximum, and bias of the measured data, for all the reference input voltages.

Table 4.3. ADC bias error performance test results.

Reference Voltage	Average (V)	Standard Deviation (µV)	Minimum (V)	Maximum (V)	Bias (mV)
-7.5	-7.498	702	2.498	2.506	1.57
-5	-4.998	703	4.975	5.012	2.17
-2.5	-2.497	708	7.488	7.516	2.69
0	0.001	717	9.996	10.005	0.56
2.5	2.498	722	12.495	12.502	-1.60
5	4.999	729	14.995	15.004	-1.05
7.5	7.500	731	17.496	17.503	-0.45

A plot of the average of the input measured vales is presented on Figure 4.10, while the standard deviation is presented on Figure 4.11. Figure 4.12, on the other hand, shows the calculated bias error values.

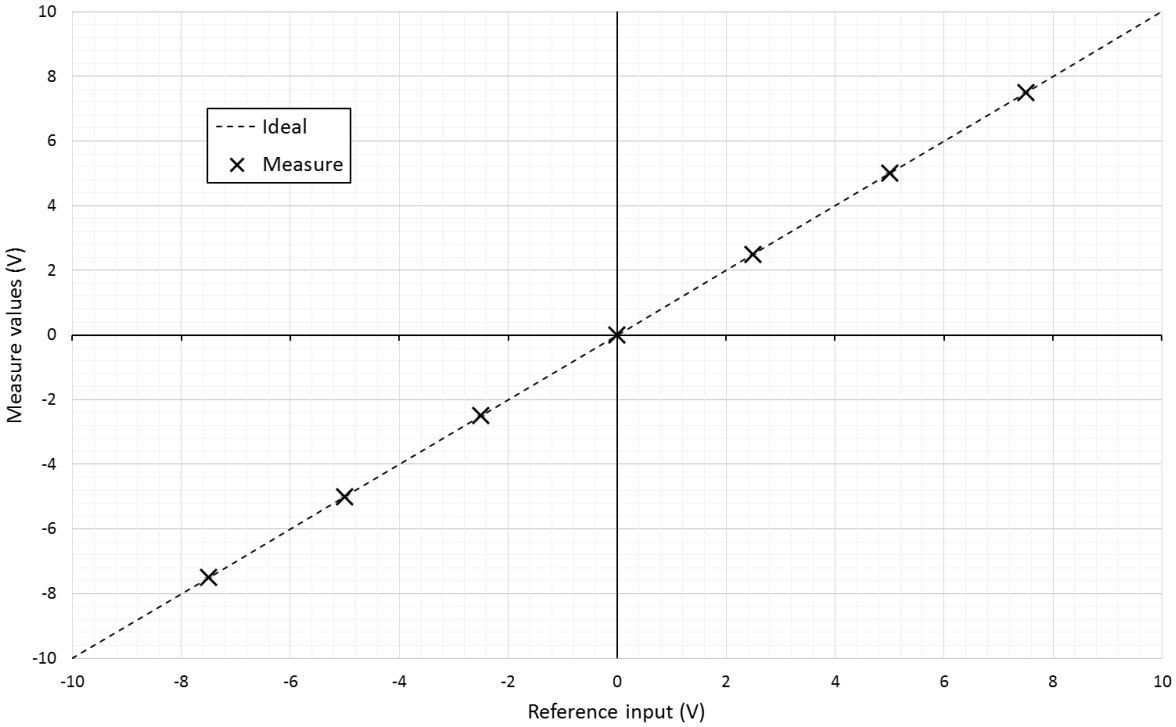


Figure 4.10. ADC measure average values. (The ideal case is presented as a reference).

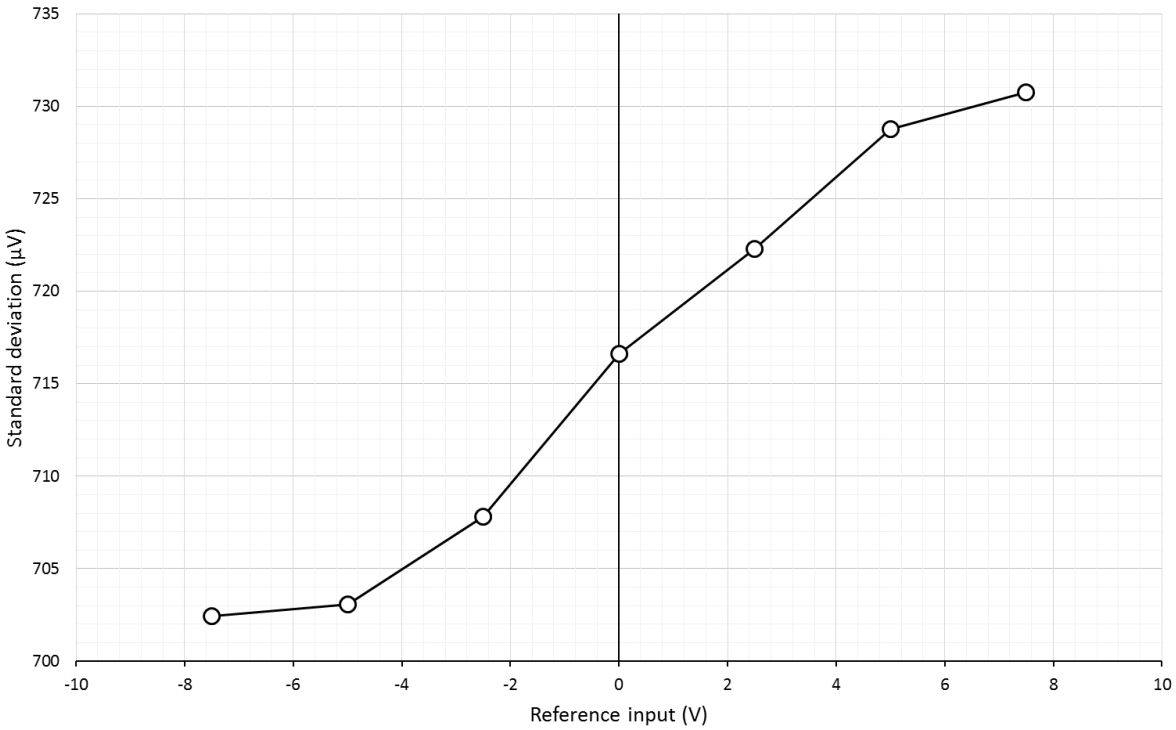


Figure 4.11. ADC measured standard deviation values.

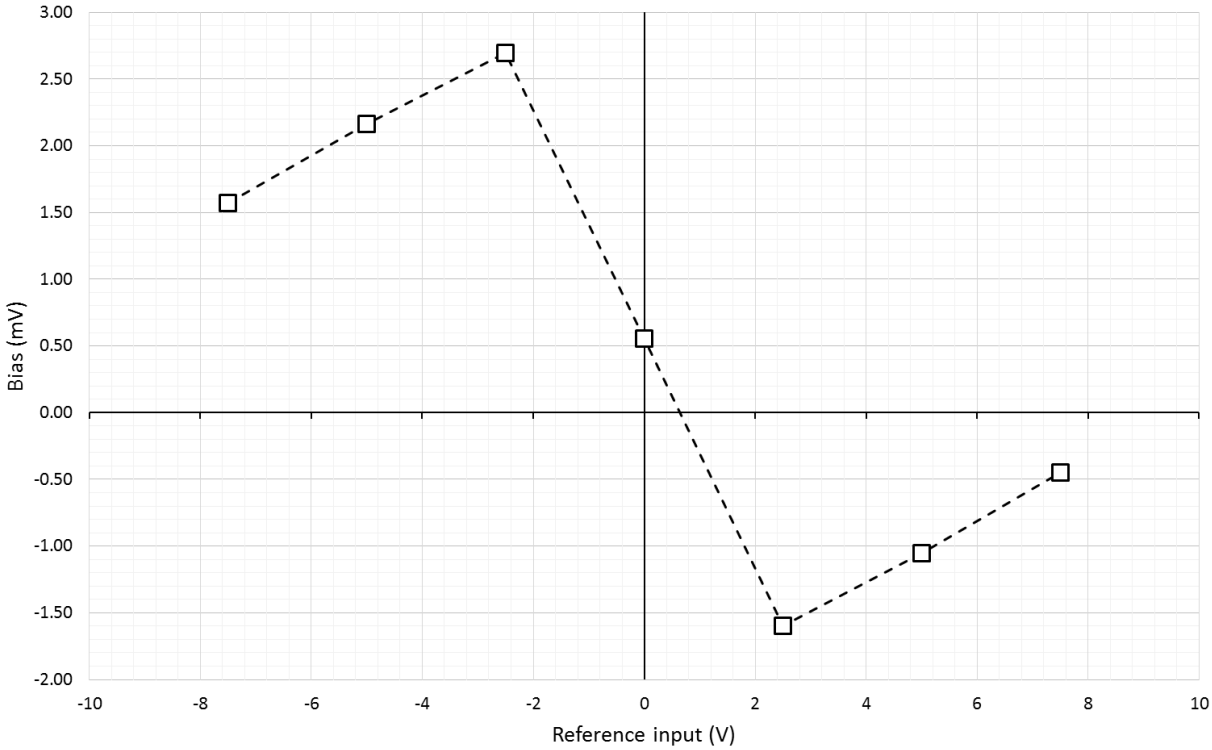


Figure 4.12. ADC input bias error values.

From Figure 4.10 and Figure 4.11 it is evident that the measure values fit the ideal curve in a very accurate way, with a standard deviation of the values between 700 and 730 µV approximately. Moreover, the bias error showed on Figure 4.12 has values below the 2.7mV, on all cases.

4.4.3 Digital-to-Analog performance

A third test was performed in order to estimate the resolution of the DAC outputs if the IOC. The teste was performed at DC level, generating constant values and acquiring them with a commercial instrument.

Three constant values where generated using an analog output (set on range 0-10 V), 2.5 V, 5 V and 7.5 V. The output values were measured using a Tektronix DMM4020 digital multimeter. The data was collected through the multimeter serial port. Average, standard deviation, minimum, maximum, and bias error values where calculated and are presented on Table 4.4.

Figure 4.13 and Figure 4.14 present a plot of the average of the measured output values and the standard deviation respectively. The bias error calculated values are shown on Figure 4.15.

Table 4.4. DAC performance test results.

Reference Voltage	Average (V)	Standard Deviation (µV)	Minimum (V)	Maximum (V)	Bias (mV)
2.5 V	2.49	52	2.49	2.49	-8.35
5 V	4.98	63	4.98	4.98	-17.41
7.5 V	7.47	48	7.47	7.47	-26.92

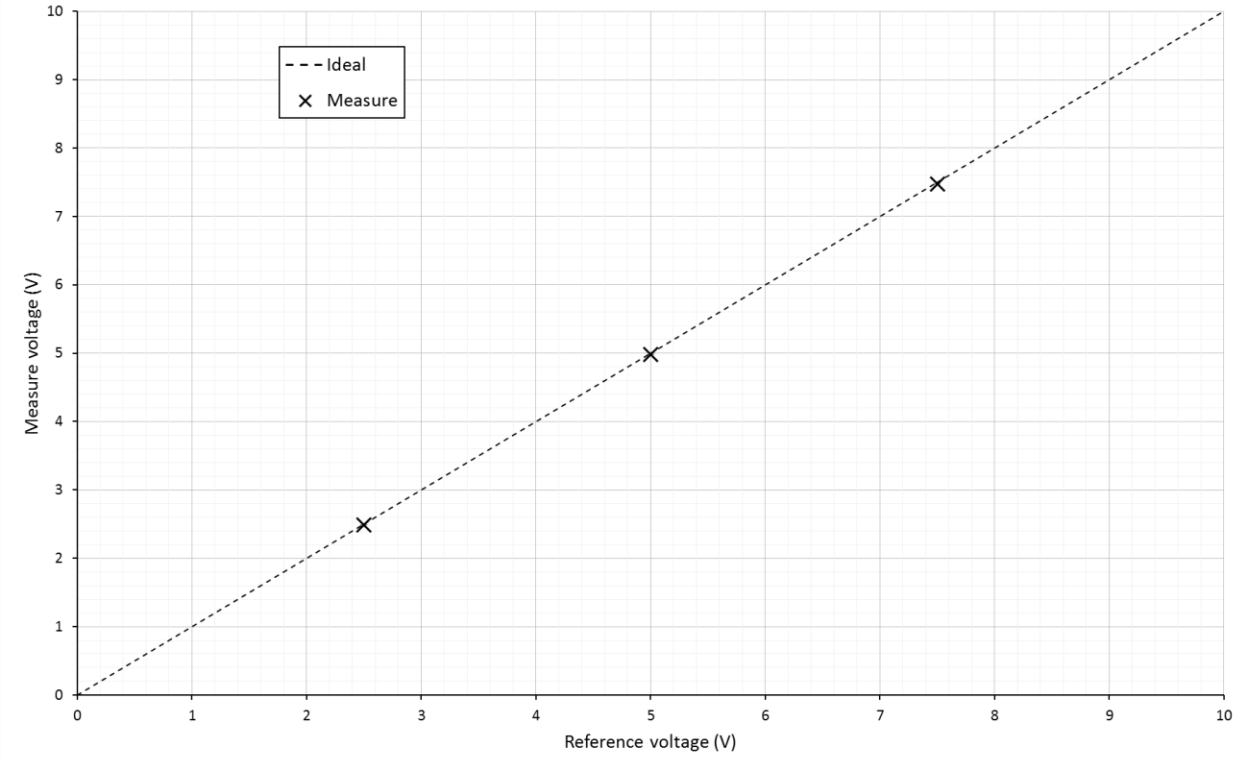


Figure 4.13. Measured DAC output voltages. *(The ideal case is shown as reference).*

It can be seem on Figure 4.13 that the measured values accurately fit the ideal curve, with a standard deviation nearly between the 45 and 65 µV, as presented on Figure 4.14. The bias error values are below 30 mV, as seems on Figure 4.15.

The average standard deviation of the data presented on Table 4.4 is 54 µV, which on a 0-10 V, 16-bits DAC output represents 0.35 words. If we considered that the output signal has a normal distribution, the 99.7% of the data points are within six standard deviations, which it is equivalent to 1.1 bits of resolution, approximatively. This gives us an estimation of 14.9 effective bits of resolution on the system analog outputs.

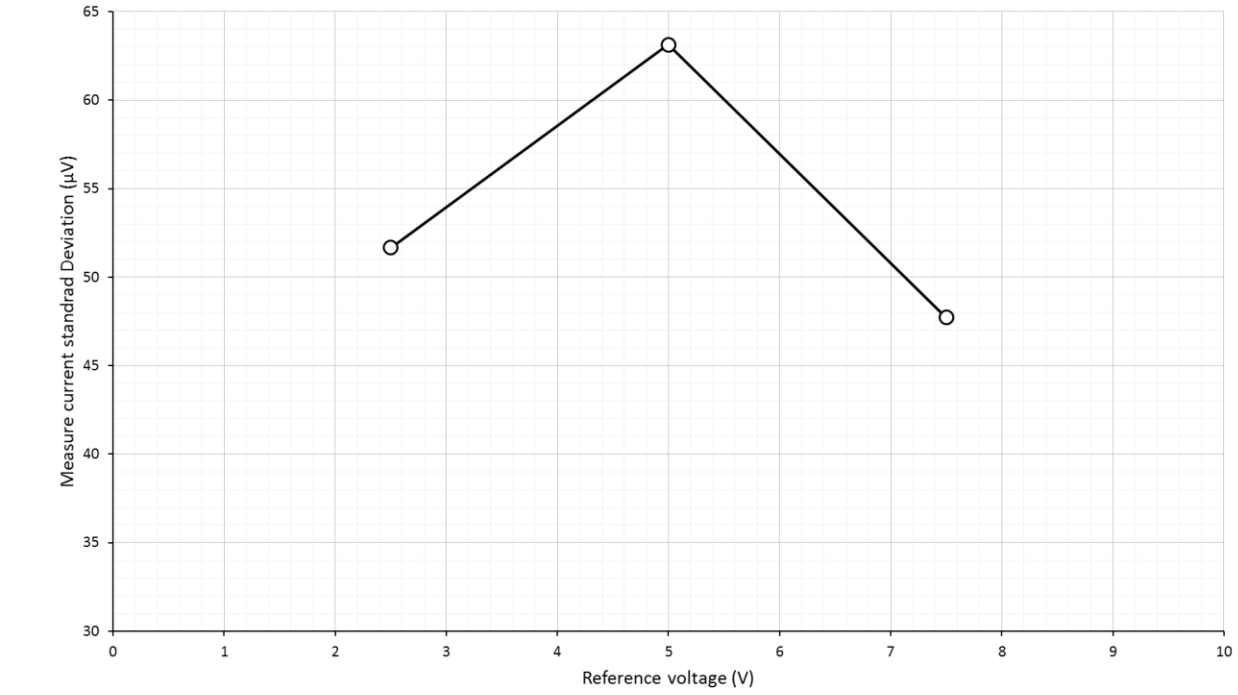


Figure 4.14. DAC standard deviation measured values.

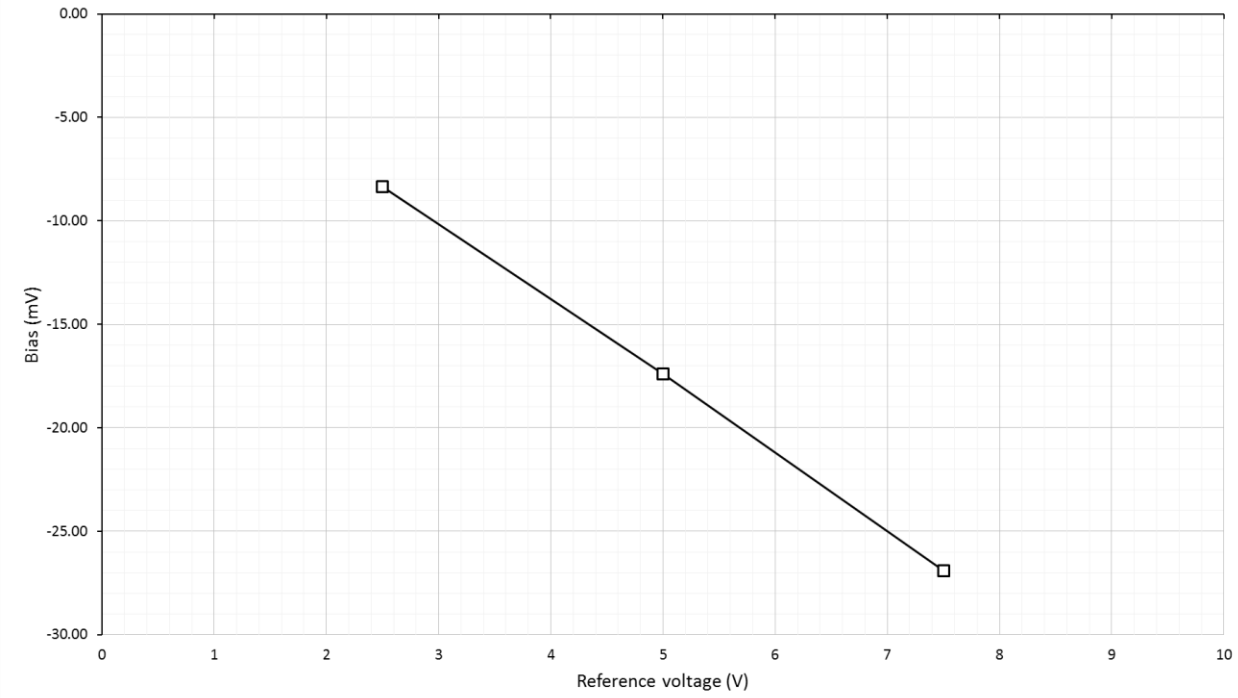


Figure 4.15. DAC output bias error values.

4.5 Control system implementation at LNL

Four different system were selected for the prototype implementation. The control system for these four systems contains most of elements that will be necessary on all the future implementations. Therefore, testing the prototype on all of them will demonstrate that the IOC could be implemented on any other system at LNL.

These systems are the beam diagnostic data acquisition (faraday cup and beam profile monitor), the electrostatic beam focalization (steerers and quadrupole triplets) and beam extraction, the magnetic beam steerers, and the ECR (Electron Cyclotron Resonance) negative beam source.

4.5.1 The beam diagnostic data acquisition

Beam diagnostic units are one of the most important and widely use elements on any accelerator facility. They allow the operators to measure beam parameters in order to be used as feedback for setting the beam current, position, direction, among others. At LNL, a standard beam diagnostic unit is formed by a Faraday Cup for measuring the beam current, and a Beam Profiler for measuring the spatial distribution particles of the beam [41].

An IOC prototype was designed for acquiring the data from this beam diagnostic unit. The beam diagnostic was installed on the beam line of an experimental hall and the IOC was integrated in the existing EPICS network, which is based on VME IOCs [31]. Figure 4.16 shows a picture of the installed IOC prototype.

Both, the Faraday Cup and the Beam Profiler detectors, use a pre-amplifier system in order to condition the signal before its acquisition. The Faraday Cup pre-amplifier is formed by a current to voltage linear converter with variable gain. On the IOC an analog input channel is used to read the output of the pre-amplifier while two digital outputs are used to set its gain. On the EPICS database, the read signal is converted back to the corresponding beam current.

The pre-amplifier of the beam profiler is formed by forty transconductance amplifiers with variable gain, similar to the one use on the Faraday Cup pre-amplifier. The 40 signals are multiplexed to a single analog output. On the IOC, an analog input channel is used to read the output of the pre-amplifier, while one digital output is used to generate a clock signal to control the analog multiplexer and two digital outputs are used to set the gain. In this case, a scanning function was implemented on the asynDriver driver that

generates the clock and synchronously reads the analog input. At the end of a scanning sequence, the data are placed on a 40-value vector, which is passed to an EPICS waveform record.

On Figure 4.17 is presented a schematic description of this data acquisition system.

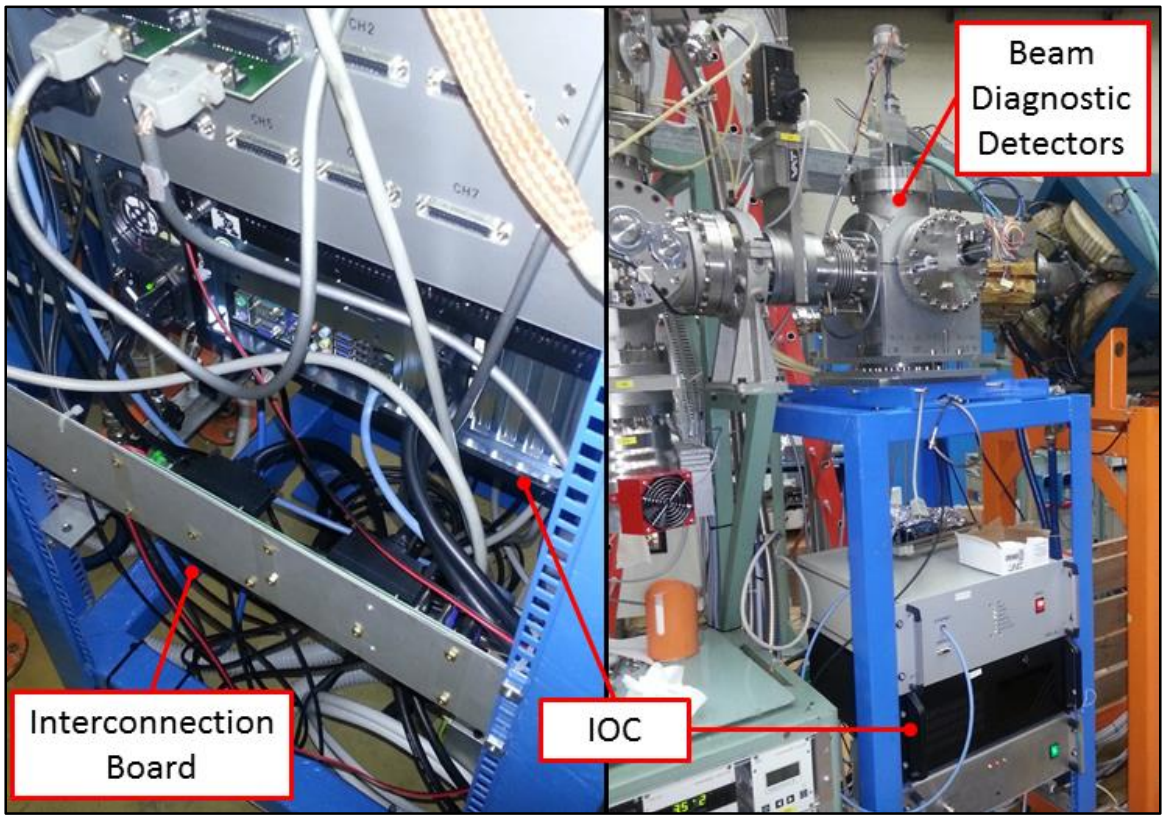


Figure 4.16. Beam diagnostic IOC prototype.

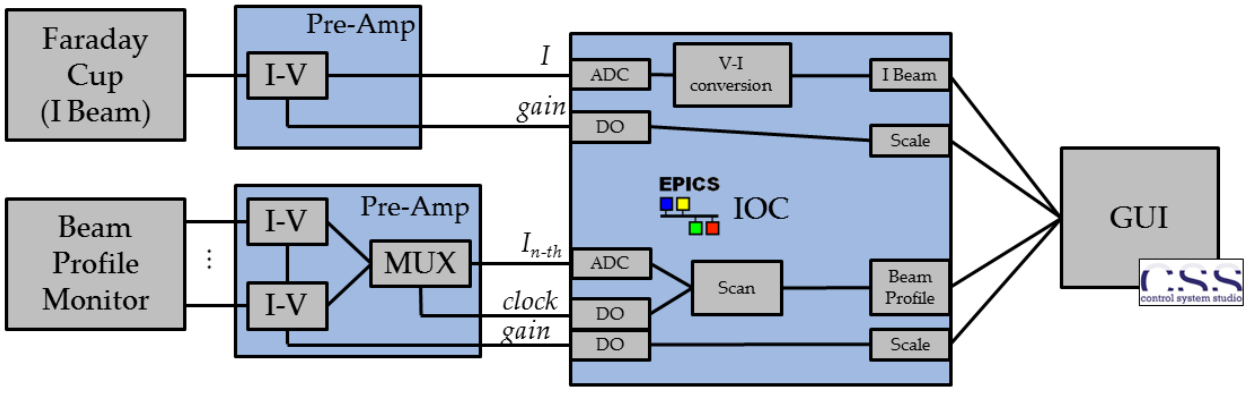


Figure 4.17. Beam diagnostic data acquisition description.

The EPICS implementation of the acquisition system consist on an Analog Input (AI) record which read the output value of the Faraday Cup pre-amplifier using a ADC channel through the asynDriver interface driver. On the other hand, two Waveform record read the signals from the pre-amplifiers of the beam profile monitors (X and Y plane), using two ADC channels through thr asynDriver interface driver (which also multiplex the channels). Finally, the gain bits of all pre-amplifiers are controlled using a Multi-Bit Binary Output (MBBO) record, which in turn controls the digital output port through the asynDriver interface driver. Figure 4.18 shown this system implantation.

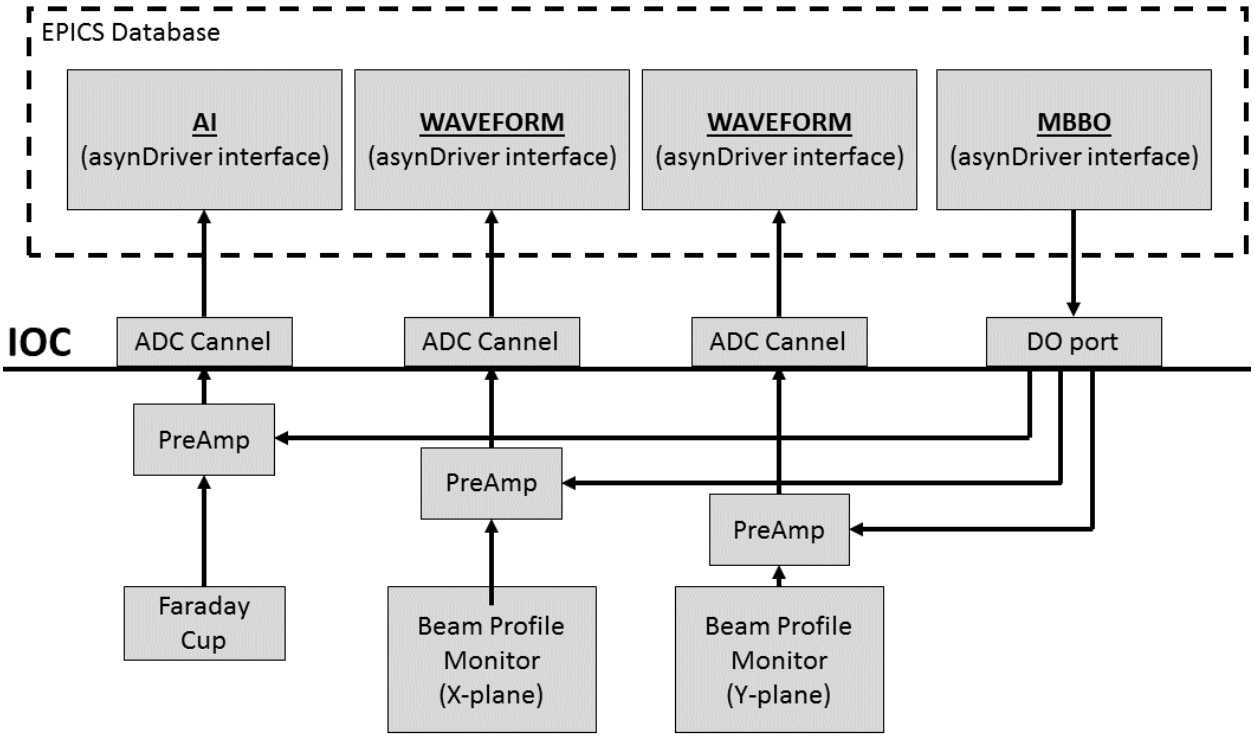


Figure 4.18. The EPIC implementation of the beam diagnostic data acquisition system.

For this acquisition system, a GUI was designed using CCSand integrated in the LNL main diagnostic operator screen. The operator is presented with the beam current and profile values as well as with the gain controls. Figure 4.19 shows a screenshot of the GUI screen.

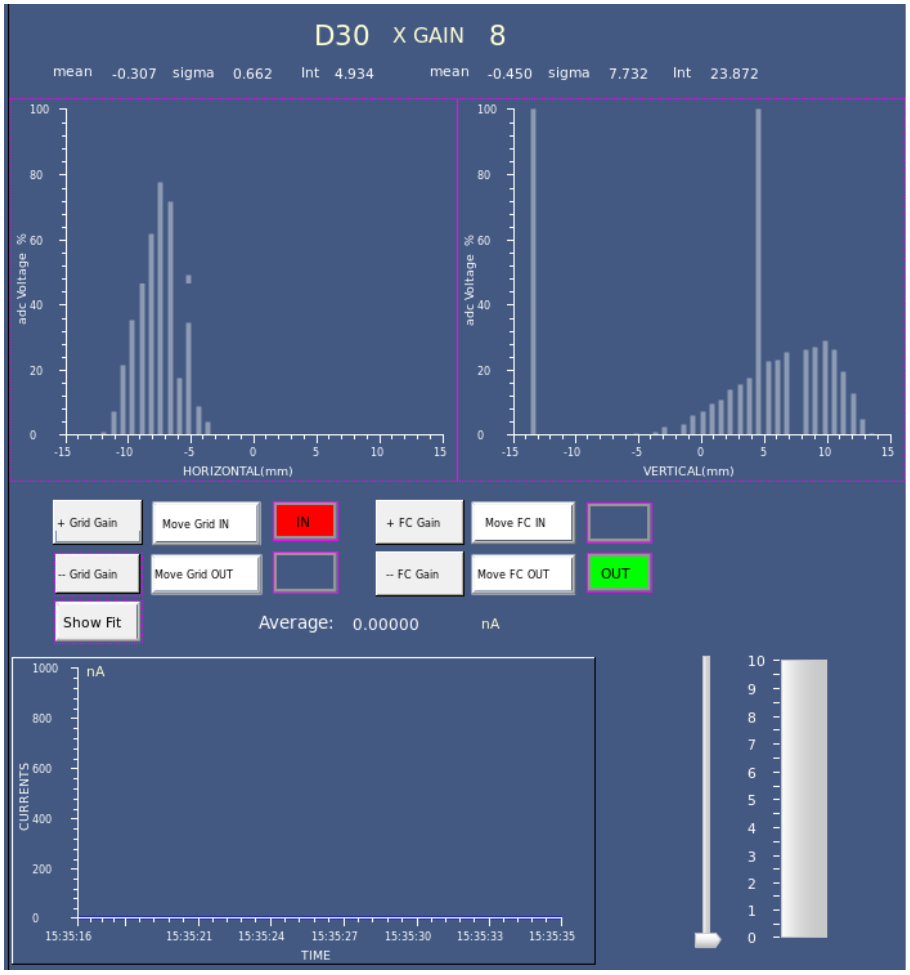


Figure 4.19. Beam diagnostic IOC prototype GUI (The user can insert and extract the detectors and see the corresponding acquire values).

4.5.2 The electrostatic beam focalization and beam extraction

Beam focalization devices are of paramount importance on an accelerator facility, used even more extensively that the beam diagnostics. Their goal, in conjunction with beam diagnostics, is to transport the beam around the accelerator, generally from the beam source into an experimental hall. On the SPES project, a large number of electrostatic beam focalization units will be install along the beam line, for transporting the low intensity beam into ALPI for reacceleration. Each one of these units is formed by a series of high voltage power supplies, used for setting an electrical potential to electrodes installed inside the beam transport line. Groups of these electrodes will generate electrical fields, hence inducing forces on the charged particles of the beam.

On the other hand, the beam extraction system, even though it has a completely different function, its control system implementation is very similar to the beam focalization system. It consist on a single high voltage power supply that set an electric potential between the ion source and the extraction electrode.

An IOC prototype was designed for controlling the beam extraction system and an electrostatic beam focalization unit. The focalization unit comprises a steerer and a quadrupole triplet groups. Both systems -extraction and focalization- are used for the initial transport of the RIB, and are present in the SPES Front-End apparatus, installed in the SPES off-line laboratory, as presented on Figure 4.20.

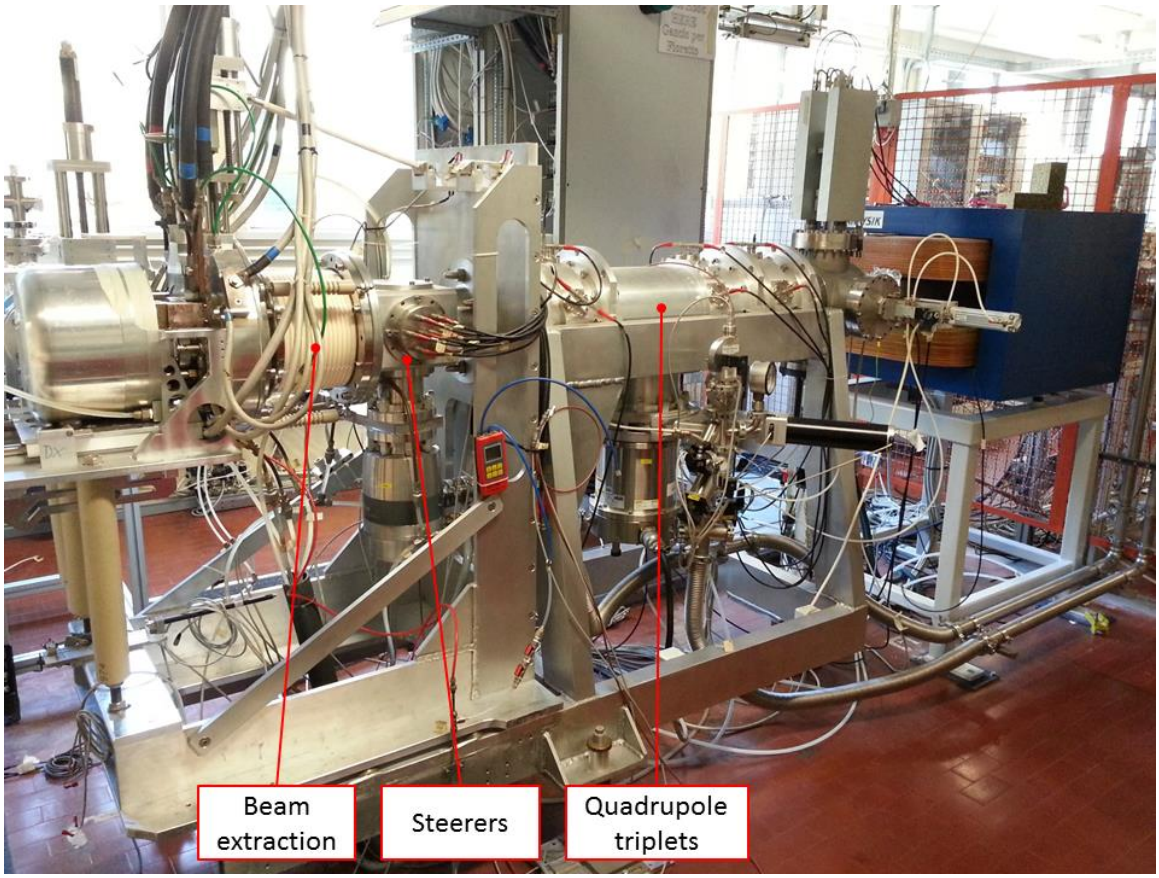


Figure 4.20. Beam extraction and focalization system at the off-line laboratory.

The IOC was installed in the existing EPICS network, form by the Raspberry Pi based IOCs (previously described on chapter 3.4) as well as VME IOCs [31] and Cosylab's microIOCs [42]. Figure 4.21 shows picture of the IOC installed along with the high voltage power supplies on the SPES off-line laboratory.

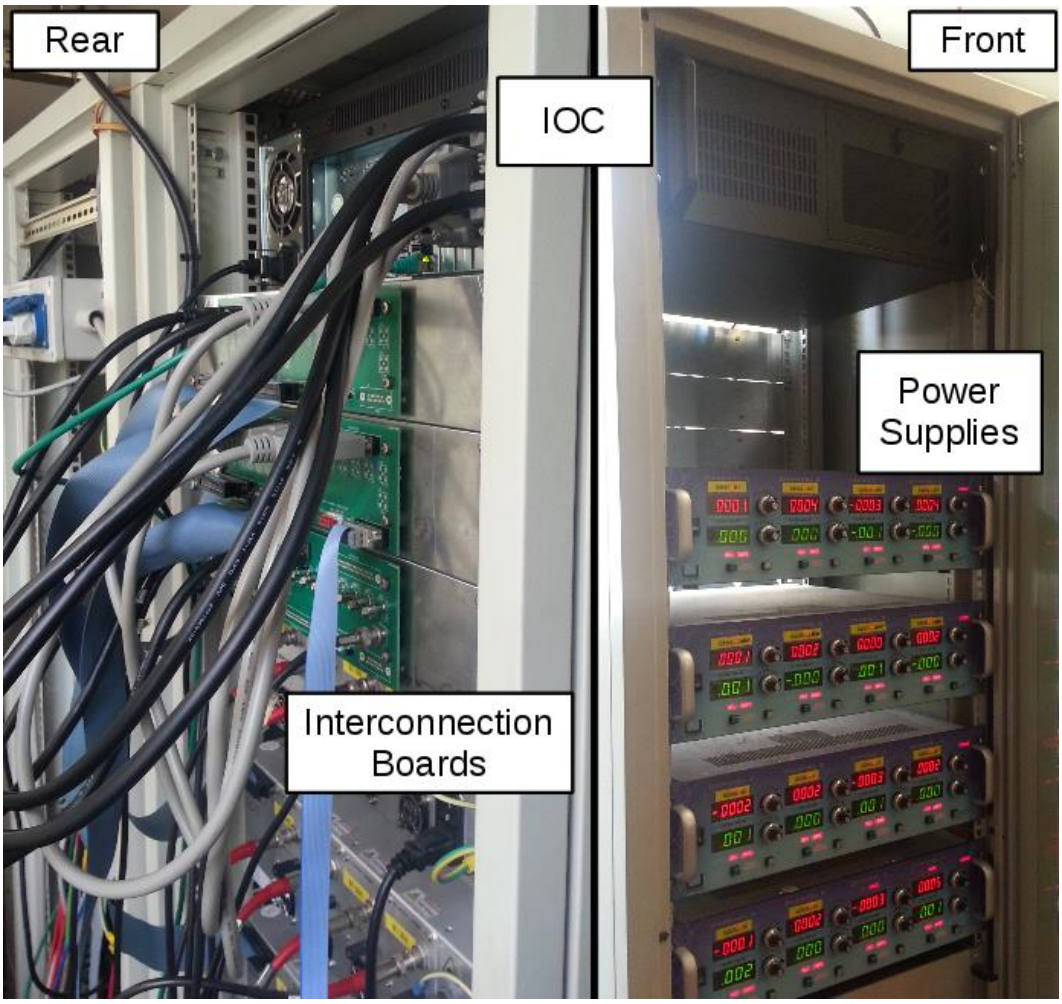


Figure 4.21. Electrostatic beam focalization IOC prototype(On top the prototype IOC, connected to the power supplies using custom interconnection boards).

On the steerer and quadrupole groups, there are pairs of electrodes driven using two power supplies, set symmetrically to the same voltage but with opposite polarity. Eight power supplies are used for controlling the steerers, and six for the quadrupoles. The beam extraction, as explained before, only uses one power supply.

For each power supply, both the output voltage and current can be set using four analog signals. An analog output and input are used for creating a closed control loop for each parameter. Additionally, a digital signal allows to switch on and off the power supply output.

On the IOC, two ADC channels, two DAC channels and a digital output are used to control each one of the power supplies. Inside the EPICS databases, Proportional-Integral-Derivative (PID) algorithms were

implemented in order to automatically control both the current and voltage, according to the set points imposed by the operator. Moreover, the database is implemented in such a way that a single set point is used to set the same voltage with opposite polarity into the two power supplies of each pair of electrodes. Figure 4.22 shows a schematic description of the system.

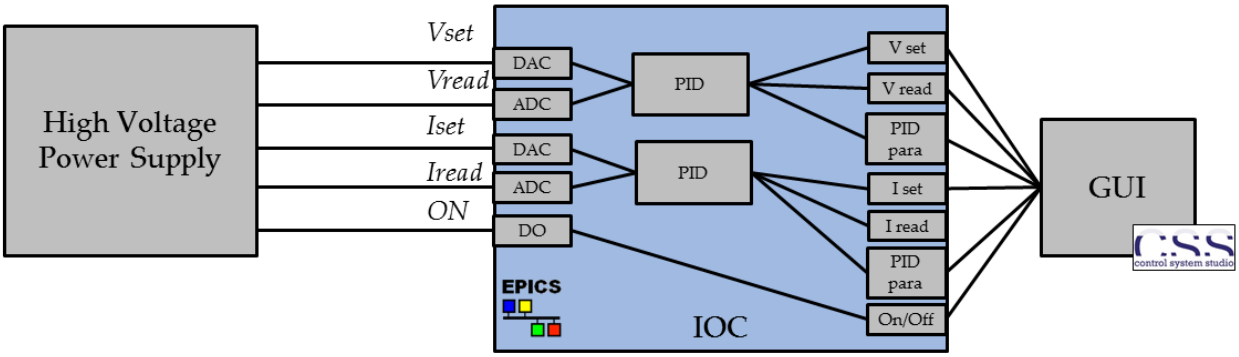


Figure 4.22. Electrostatic beam focalization system description. One of this block is used for controlling each one of the power supplies.

As the standard PID record found on the basic EPICS distribution not provide some necessary functionalities for this application, a custom enhanced PID algorithm was implemented. The implementation uses an aSub (Array Subroutine) EPICS record, which calls a user-defined C routine where the PID algorithm is implemented.

This enhanced PID algorithm, adds the following functionality to existing PID algorithm:

- A low-pass filter for the process variable, with settable time constant,
- A dead zone for the error value,
- Maximum and minimum limit to the output value,
- Selectable mode of operation, between manual and automatic. In automatic mode, the PID algorithm is perform for calculating the output value; while in manual mode, the output is set to an user-defined value,
- An enable/disable flag. When Enable, the PID algorithm is performed and the output value updates accordingly; while when disable, the algorithm is not perform and the output is keep at zero.

The input filter is a discrete-time implementation of a RC low-pass filter, i.e., an exponentially-weighted moving average filter, which is defined as [43],

$$y[n] = a \cdot x[n] + (1 - a) \cdot y[n-1] \quad (4.2)$$

Where *y[n]* is the filtered output signal, *x[n]* is the input signal (in this case the process value), and *a* is defined as,

$$a = \frac{T_S}{T_S + \tau} \quad (4.3)$$

Where T_S is the sampling period, and τ is the time constant of the low-pass filter, equal to *RC*.

The PID controller was implemented using the discrete-time form of the velocity algorithm, also known as the incremental algorithm, defined as

$$d[n] = k_1 \cdot e[n] + k_2 \cdot e[n-1] + k_3 \cdot e[n-2] \quad (4.4)$$

Where *d[n]* is the delta to be add the previous output value, *e[n]* is the error value calculated as the difference between the set point and the filtered process value, and the three constant are defines as [44],

$$k_1 = k_p + k_i \cdot T_S + \frac{k_d}{T_S} \quad (4.5)$$

$$k_2 = -k_p - 2 \cdot \frac{k_d}{T_S} \quad (4.6)$$

$$k_3 = \frac{k_d}{T_S} \quad (4.7)$$

Where k_p, k_i and k_d are respectively the proportional, integrative and derivative gain of the controller; and T_S is the sampling period.

A flowchart of the complete PID algorithm is presented on Figure 4.23, while the implemented C routine is reported on the appendixes.

The EPICS implementation of the closed control loop consist on an Analog Input (AI) record, which read the power supply output value using an ADC channel through the asynDriver interface driver. This value is pass to the aSub record where the PID algorithm is implemented. The resulting value is then written to an Analog Output (AO) record which control the power supply using a DAC channel through the asynDriver interface driver. To the aSub record are also passed of the configuration parameters (filter time constant, set point, dead zone, maximum and minimum output, PID constants, the enable/disable flag, and the

operation mode with eventual manual value to be set on the output). Figure 4.24 illustrate this control loop.

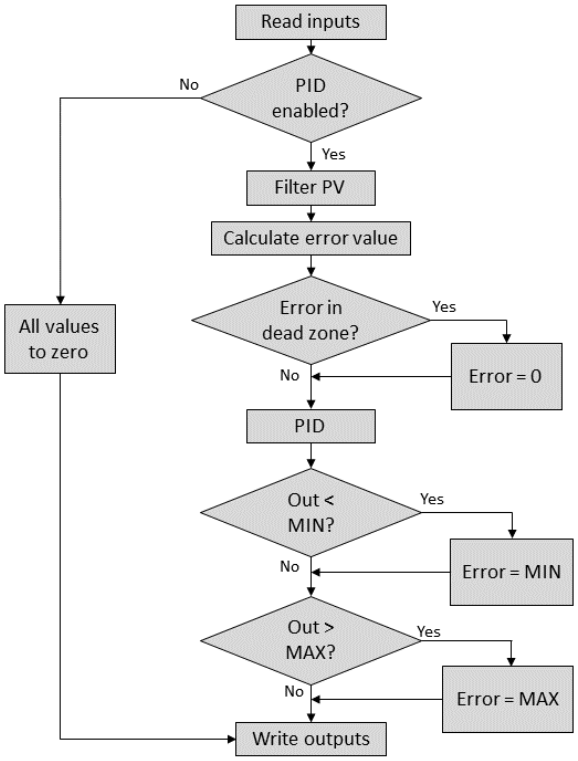


Figure 4.23. Implemented PID algorithm flowchart.

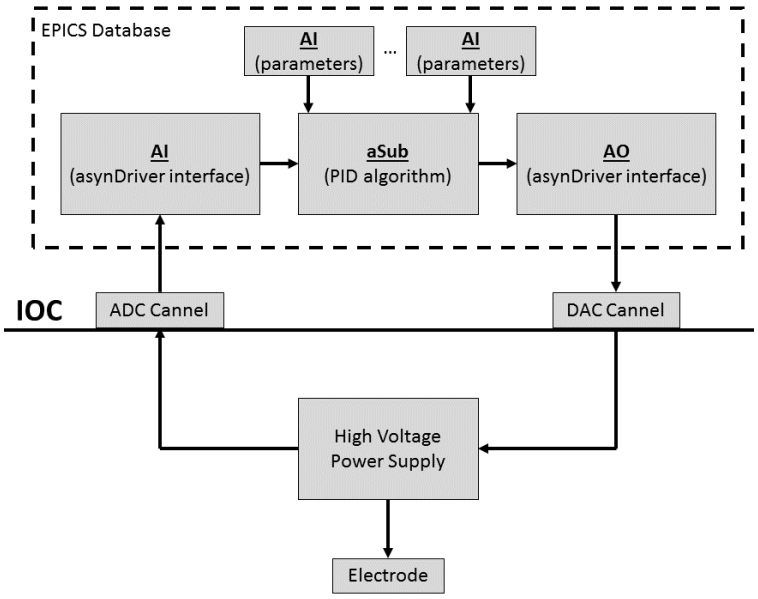


Figure 4.24. The EPIC PID control loop implementation for a high voltage power supply.

In order to measure the step response of a power supply, the corresponding PID channel was configured in manual mode, and the output voltsge was set to a 50% full-scale value. The obtained and normalized step response is presented on Figure 4.25. It can be seen an overshoot of almost a 33%, while the steady-state error is around the 0.86%. These values does not satisfy the operator request, which is an overshoot not greater than 10% and a steady-state error lower than 0.05%.

From the step response, it seems to be an underdamped second order system. From the plot we can calculate the overshoot (*OS*), damp ratio (ζ), natural frequency (ω_n), and gain (*K*) parameters as follow,

$$OS = \frac{V_{peak} - V_{steady-state}}{V_{steady-state}} \approx \frac{1.332975 - 1.008641}{1.008641} = 0.321556 \tag{4.8}$$

$$\zeta = \frac{-ln(OS)}{\sqrt{\pi^2 + ln^2(OS)}} = 0.318894 \tag{4.9}$$

$$\omega_n = \frac{\pi}{T_{peak}\sqrt{1-\zeta^2}} \approx \frac{\pi}{0.143\sqrt{1-0.318894^2}} 23.17921 \tag{4.10}$$

$$K = V_{steady-state} \approx 1.008641 \tag{4.11}$$

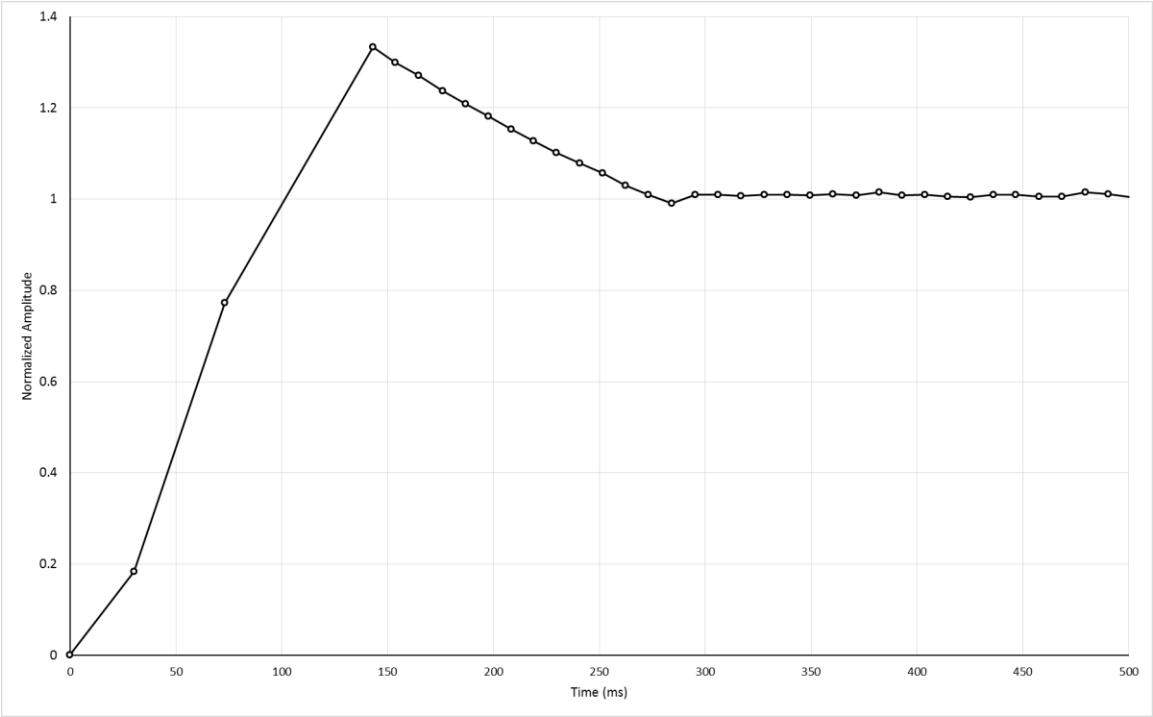


Figure 4.25. Step response of a power supply used on the beam focalization system.

Then, the estimated second order system transfer function is,

$$G(s) = K\frac{\omega_n^2}{s^2 + 2\cdot\zeta\cdot\omega_n\cdot s + \omega_n^2} = \frac{541.91808}{s^2 + 14.783438\cdot s + 537.27561} \qquad (4.12)$$

Using Scilab [29], the step response of the estimated system was obtain and it is reported on Figure 4.26 along with the real system response presented previously on Figure 4.25. It can be appreciated that the estimation fits closely the real system behavior.

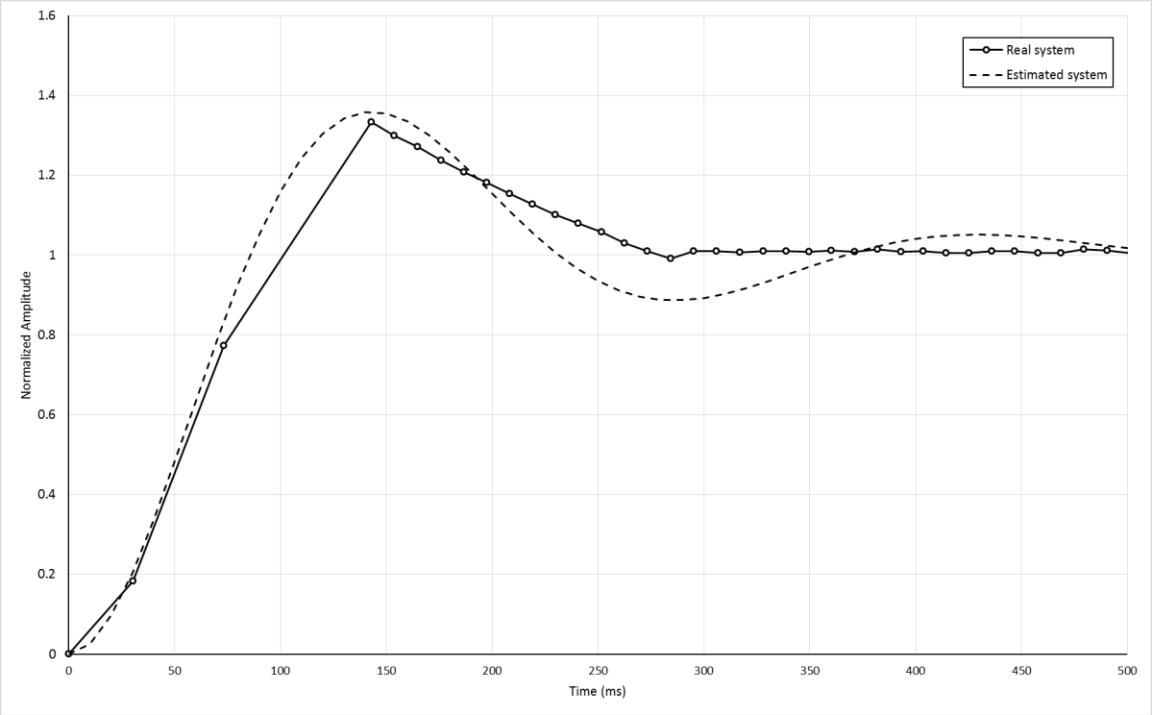


Figure 4.26. Estimated vs real system step responses.

The pole-zero diagram of the estimated system is presented on Figure 4.27. The system has only two poles as shown in the figure.

After the identification of the system, a PID tuning procedure was performed. The main goal was to reduce both, the overshoot and steady-state parameters, while no restriction was set for the settling time (as the system is used only on a static way). The PID channel was configured on automatic mode, and the following PID parameters were obtained using the Ziegler–Nichols heuristic method [45],

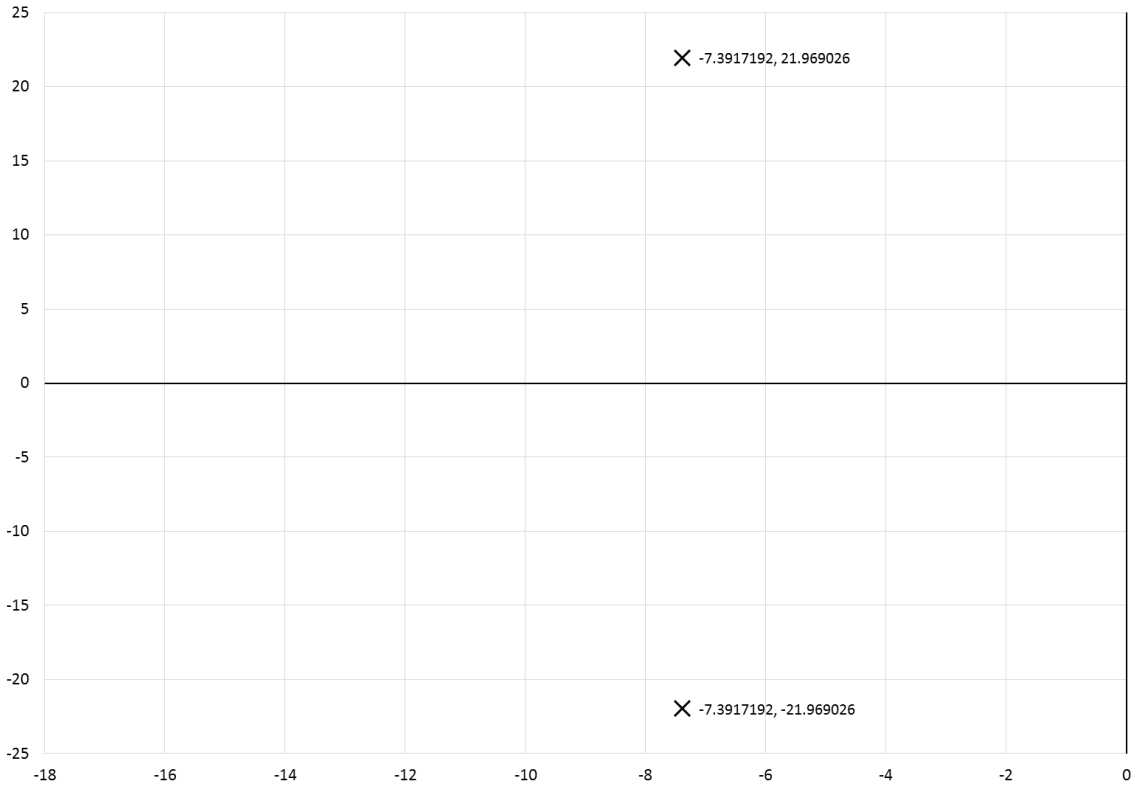


Figure 4.27. Pole-zero diagram of the estimated system.

$$\begin{matrix} k_p = 1.8 \\ k_i = 6 \\ k_p = 0.003 \end{matrix} \qquad (4.13)$$

The PID transfer function is then,

$$H(s) = \frac{k_d \cdot s^2 + k_p \cdot s + k_i}{s} = \frac{0.003 \cdot s^2 + 1.8 \cdot s + 6}{s} \qquad (4.14)$$

Using the estimated system transfer function from equation (4.12) in conjunction with the PID transfer function from equation (4.14), the estimated closed-loop system transfer function is,

$$H_C(s) = \frac{G(s)H(s)}{1 + G(s)H(s)} = \frac{6.0944999 \cdot s^2 + 3656.6999 \cdot s + 12189}{s^3 + 34.717625 \cdot s^2 + 5670.7966 \cdot s + 12189} \qquad (4.15)$$

The obtained system step response using the PID controller is presented on Figure 4.28, along with the estimated system one, calculated using Scilab using equation (4.15). Again, it can be seen that the estimation follows closely the response of the real system.

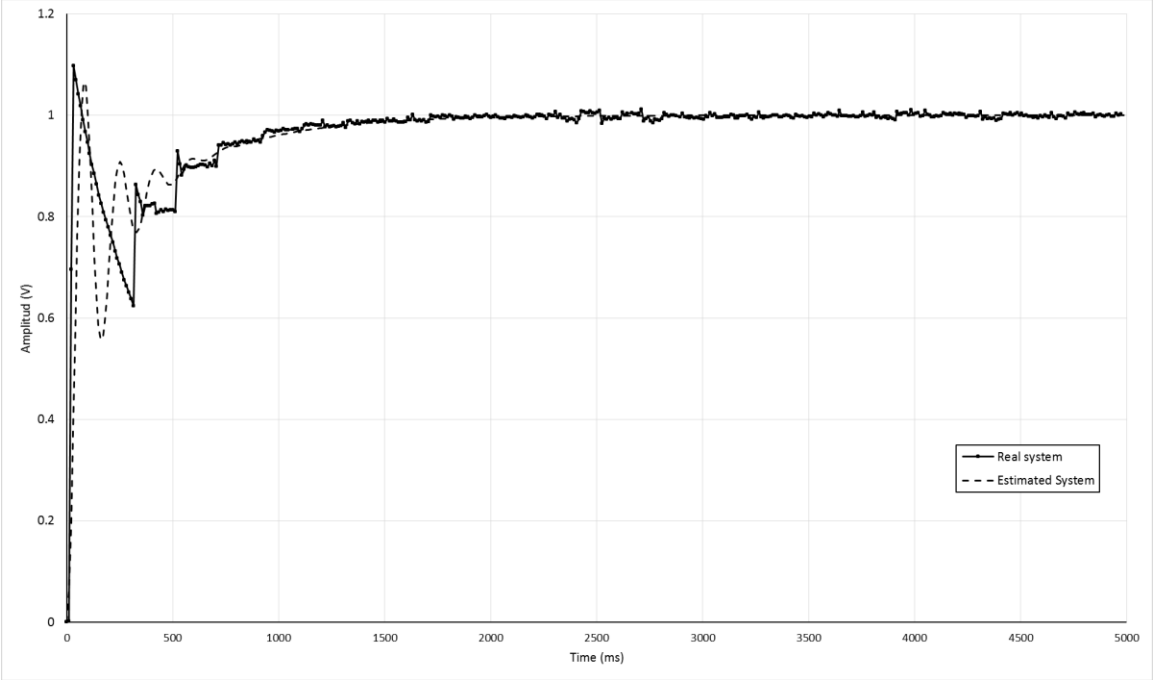


Figure 4.28. Estimated and real system step responses of the closed-loop system.

On the other hand, the pole-zero diagram is presented on Figure 4.29. The resulting system has three poles and two zeros, as showed on the figure. As all three poles lie within the left-half of the s-plane (i.e. all have negative real parts), the system is stable. The complex conjugate pole pair correspond to the decaying sinusoid component, while the pure real pole correspond to the exponentially decaying component; both components are appreciable on the system response on Figure 4.28.

Using this PID closed control loop, the overshoot has improved from a value of 33% to 9.7%, while the steady-state error has also improved from 0.86% to 0.03%. This satisfies the operator requested values of 10% and 0.05%, respectively. The downside is the settling time, which has increased from 0.2 s to 2 s; however, as explained before, in this application this parameters is not relevant.

For these systems, GUIs were developed using Control System Studio and integrated to the off-line general control panel [31]. The operator is presented with both the voltage and current set points and read back values, the PID parameters, and the power supply on/off control. Figure 4.30 shows a screenshot of the GUI screen.

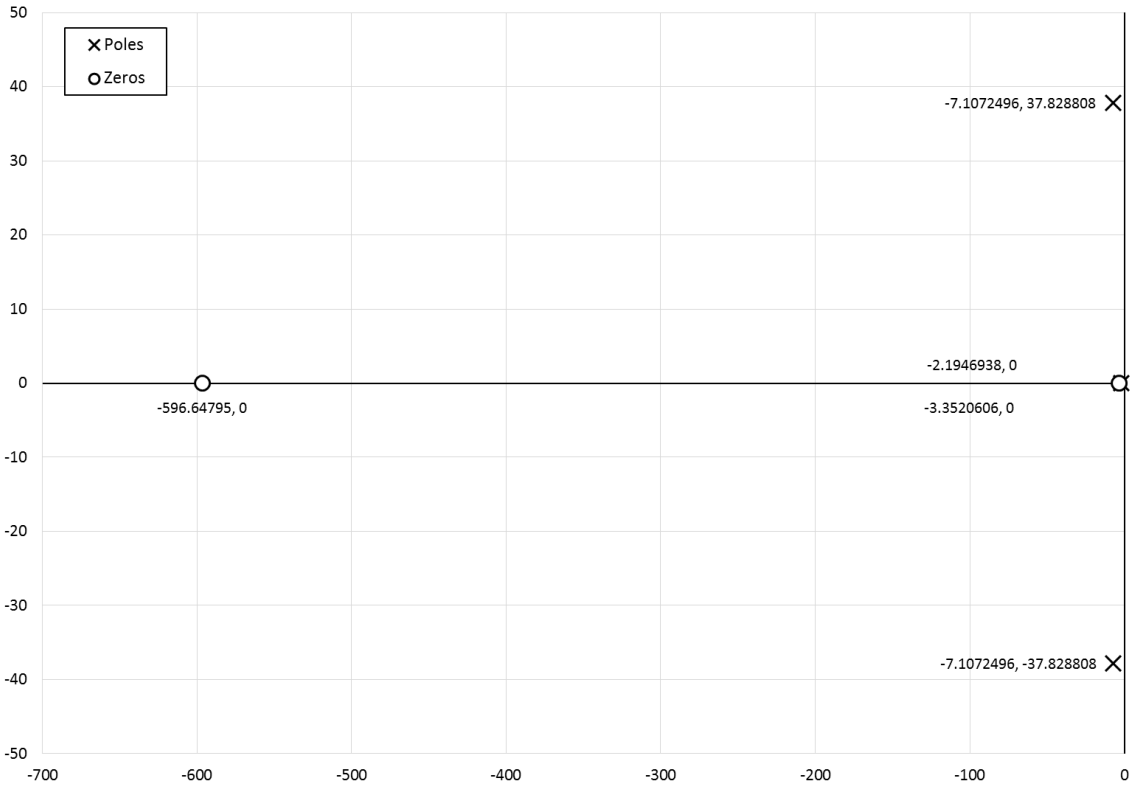


Figure 4.29. Pole-zero diagram of the estimated closed-loop system.

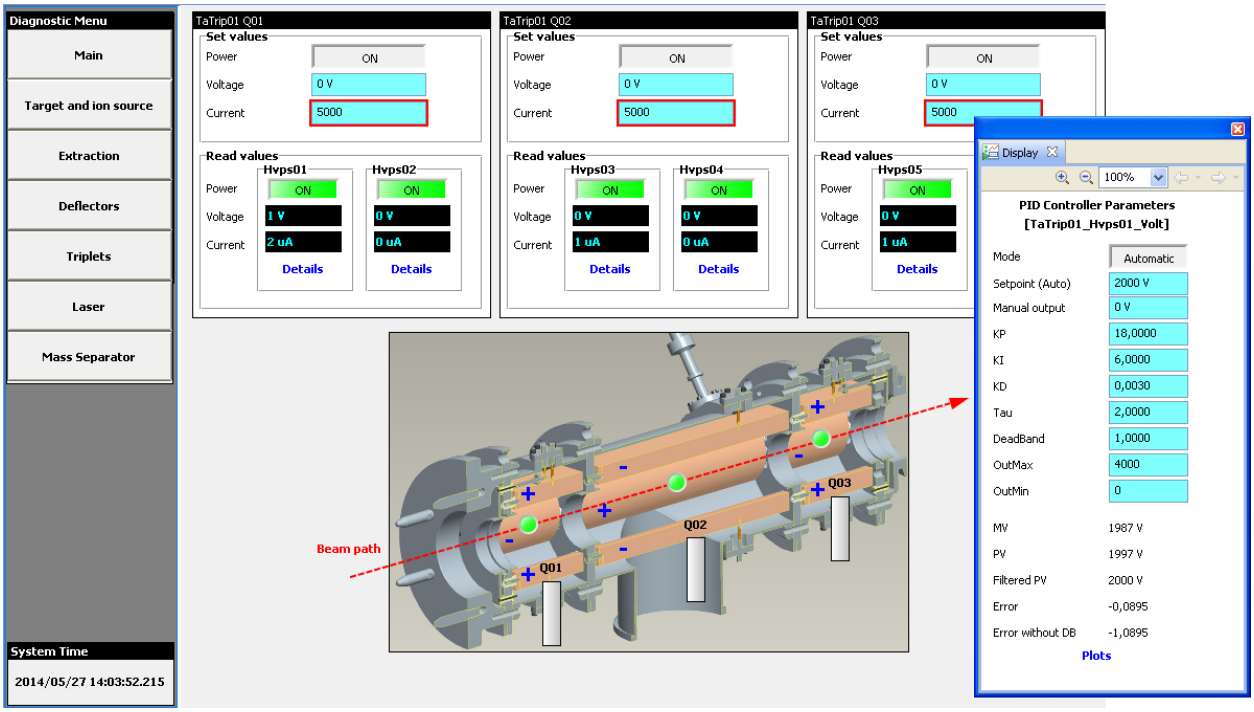


Figure 4.30. Electrostatic beam focalization IOC prototype GUI (The user can set and view the power supplies parameters).

4.5.3 The magnetic beam steerers

Magnetic steerers are part of the beam transport elements. They are widely used at LNL for the transport of high energy beams, presented after the reacceleration by means of the ALPI superconductive LINAC. For these high energy beams, magnetic focalization devices are more efficient that electrostatic ones, due the fact that the magnetic force, as opposed to the electric force, depends on the particle velocity.

A magnetic steerer unit is form by four coils, two vertical and two horizontals, installed on the beam line. They are connected on series of two, and piloted using two high current power supplies. Figure 4.31 shows one of the magnetic steerers installed at LNL.

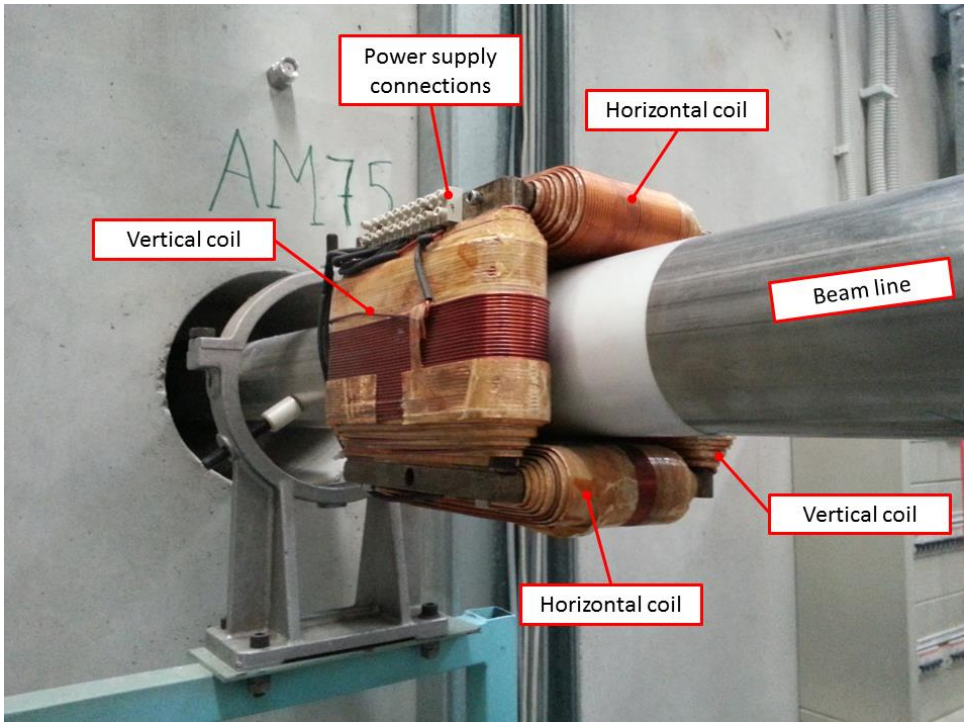


Figure 4.31. A magnetic steerer installed on a beam line at LNL.

For each power supplies, both the output voltage and current can be set using four analog signals. Additionally, a digital signal allows to switch on and off the power supply output. The IOC configuration is exactly the same respect to the electrostatic beam focalization described on the previous sections, that is, two ADC channels, two DAC channels and a digital output to control each one of the power supplies. As in the electrostatic beam focalization application, on the EPICS databases, Proportional-Integral-Derivative (PID) algorithms were implemented in order to automatically control both the current and

voltage, according to the set points imposed by the operator. It was used the same implantation explained on chapter 4.5.2. Figure 4.32 shows a schematic description of the system.

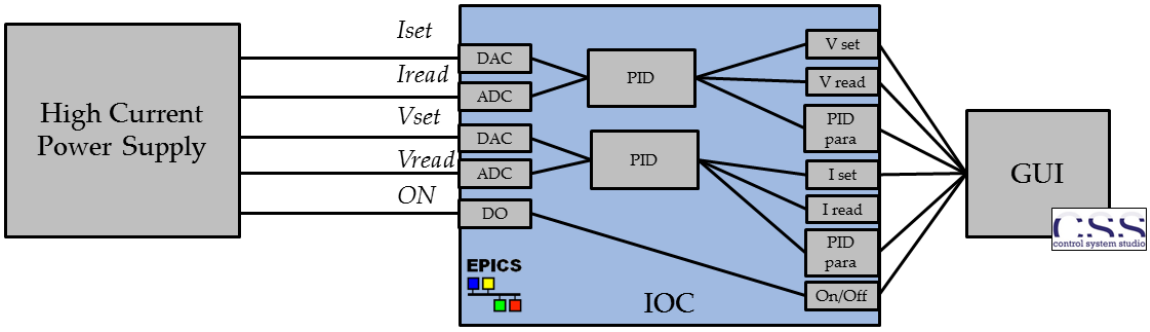


Figure 4.32. Magnetic beam steerer system description. *One of this block is used for controlling each one of the power supplies.*

The EPICS implementation of the control loop consist on an Analog Input (AI) record, which read the power supply output value using an ADC channel through the asynDriver interface driver. This value is pass to the aSub record where the PID algorithm is implemented. The resulting value is then written to an Analog Output (AO) record which control the power supply using a DAC channel through the asynDriver interface driver. To the aSub record are also passed of the configuration parameters (filter time constant, set point, dead zone, maximum and minimum output, PID constants, the enable/disable flag, and the operation mode with eventual manual value to be set on the output). Figure 4.33illustrate this control loop.

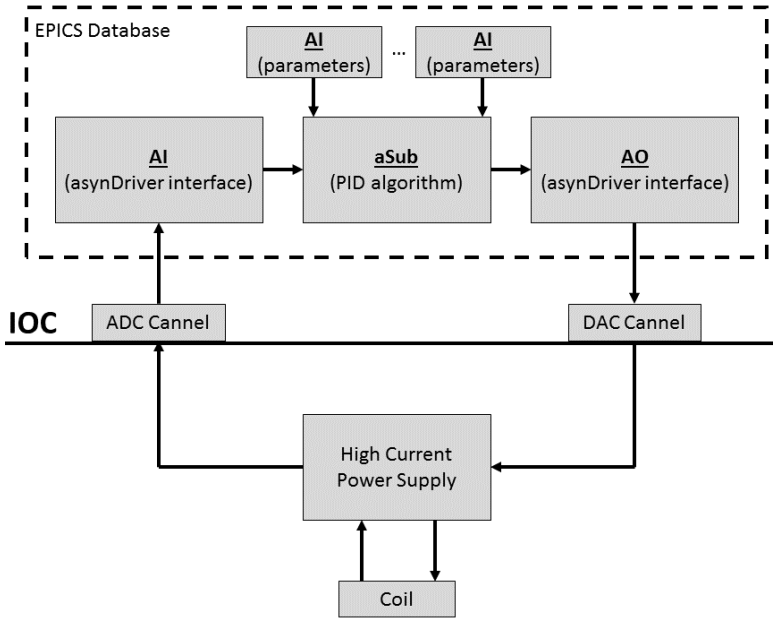


Figure 4.33. The EPIC PID control loop implementation for a high current power supply.

For this application, test IOC functionality was tested only on laboratory. The IOC implementation on a real application is still under development. In this case, both the IOC prototype and the high current power supplies are being developed side by side, as an integrated system. The result will be a four-channel power supply, able to control two magnetic steerers, with an integrated IOC.

A system analysis and PID tuning procedure will be carried out, using the same methodology explained on chapter 4.5.2.

4.5.4 The ECR (Electron Cyclotron Resonance) negative beam source

At LNL, one of the stable ion source present is an Electron Cyclotron Resonance (ECR) source [41] [46], showed on Figure 4.34. In an ECR ion source ions are produced in a magnetically confined plasma, which is heated by microwaves. In the plasma, long range Coulomb interactions take place; as a result, the high number of freely moving charges makes the plasma a very good conductor.

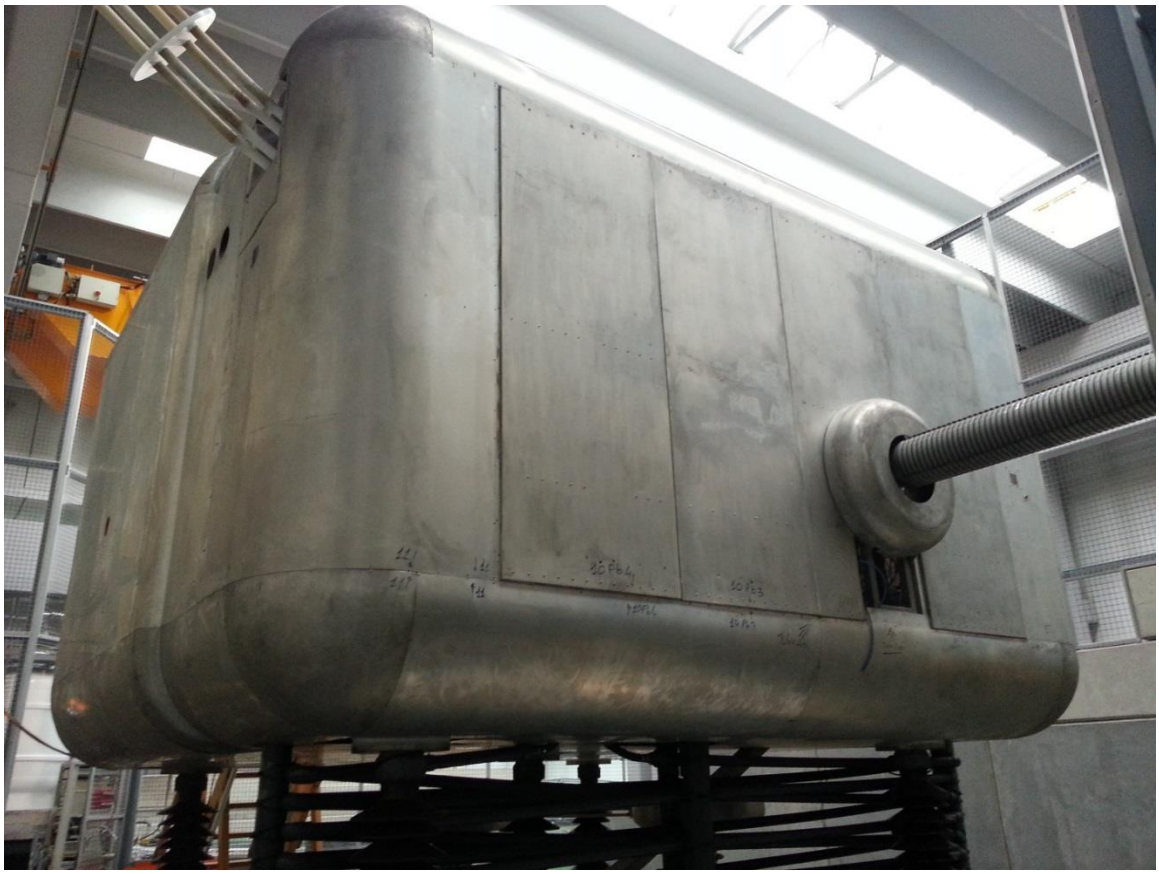


Figure 4.34. The LNL ECR ion source.

The ECR ion source in controlled by a large number of instruments, as for example, high voltage and high current power supplies, stepper motors, gas valves and vacuum instrumentation, among others, forms this system.

Furthermore, this system is very similar to the future Charge Breeder system, currently under development and to be installed on SPES. For the Charge Breeder, an EPICS control system will be developed at LNL and, for this reason, it has been decided to developed the ECR EPICS control system first, which later will be migrate to the Charge Breeder.

The control system of the ECR system is divided into two parts: all the instrumentation for the beam production and transport is controlled by an EPICS IOC; while a PLC in in charge of controlling the vacuum system. On the IOC, there is implemented a communication interface for reading the vacuum status from the PLC. The ECR ion source, and all its instrumentation, are installed on a high voltage platform, isolated from ground. Therefore, the communication with GUI installed on a control room is done using an optic fiber link. Figure 4.35 shows a description of the control system.

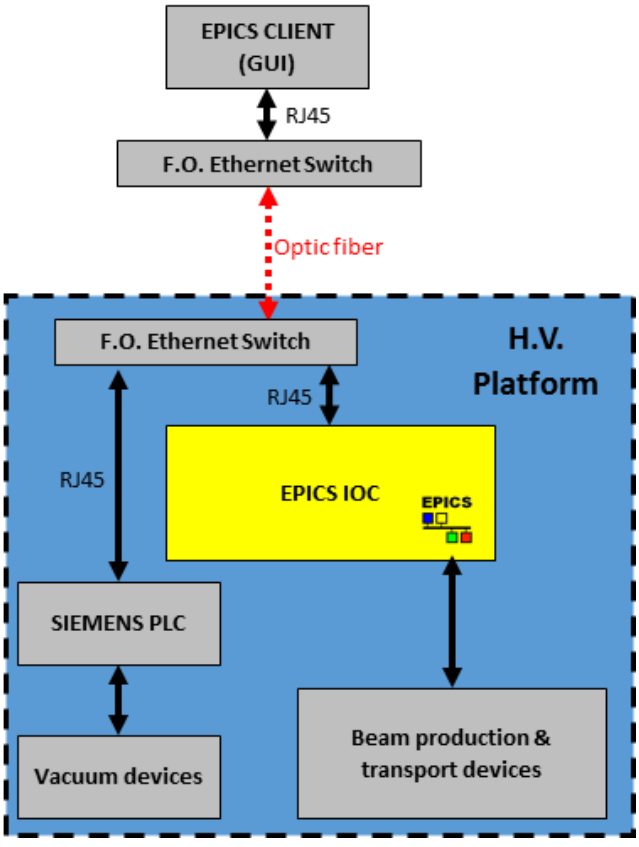


Figure 4.35. LNL ECR control system description.

The IOC developed is equipped with 16 DAC channels, 32 ADC channels, 96 digital IO and 8 serial ports. This number of interfaces were chosen in order to be able to control all the instrumentation of both, the LNL ECR ion source and the SPES Charge Breeder, with the same hardware configuration.

For the case of the ECR, the IOC controls nine power supplies with analog control, five power supplies and measure instrument with serial communication port (RS232), and one faraday cup for beam diagnostic. The communication with the PLC is done through Ethernet. Finally, the status of some external alarm are acquire using digital input channels.

Most of the software components were taken from the previously described prototypes, as for example, PIDs algorithms (using the same implantation explained on chapter 4.5.2) for the analog controlled power supplies and conversion algorithms for the beam diagnostics. On the other hand, for the instrumentation with serial communication, new algorithm were produce (as, in general, the communication protocol is different for each instrument).

The EPICS implementation of the acquisition system is identical to the one explained on chapter 4.4.1 and showed on Figure 4.18. In the same way, the EPICS implementation of the PID control loop for power supplies with analog controls is identical to the one explained on chapter 4.5.2 and shown on Figure 4.24.

Figure 4.36 shows a schematic description of the designed system. The final IOC is shown on Figure 4.37.

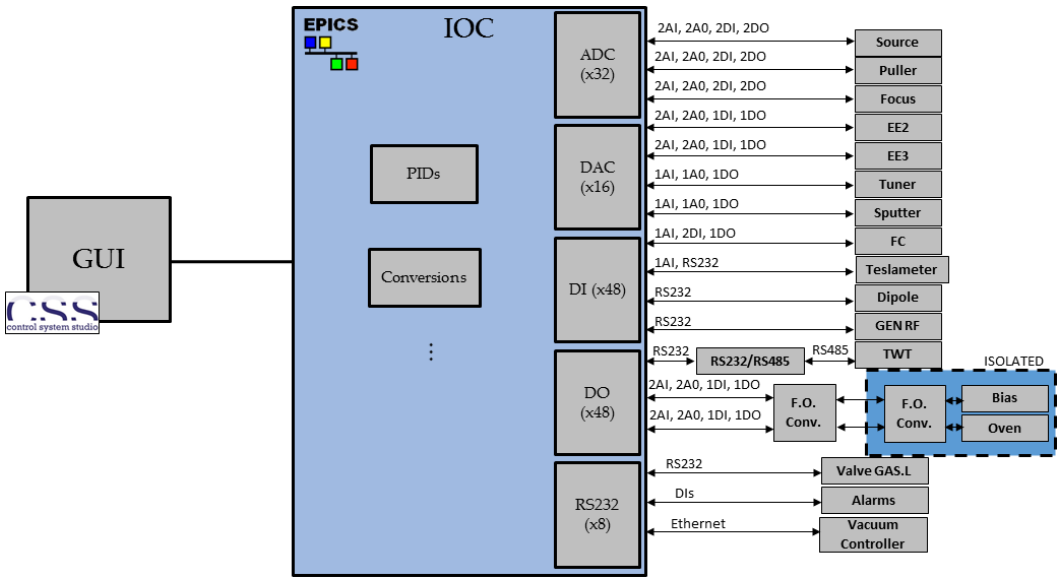


Figure 4.36. LNL ECR EPICS IOC description.

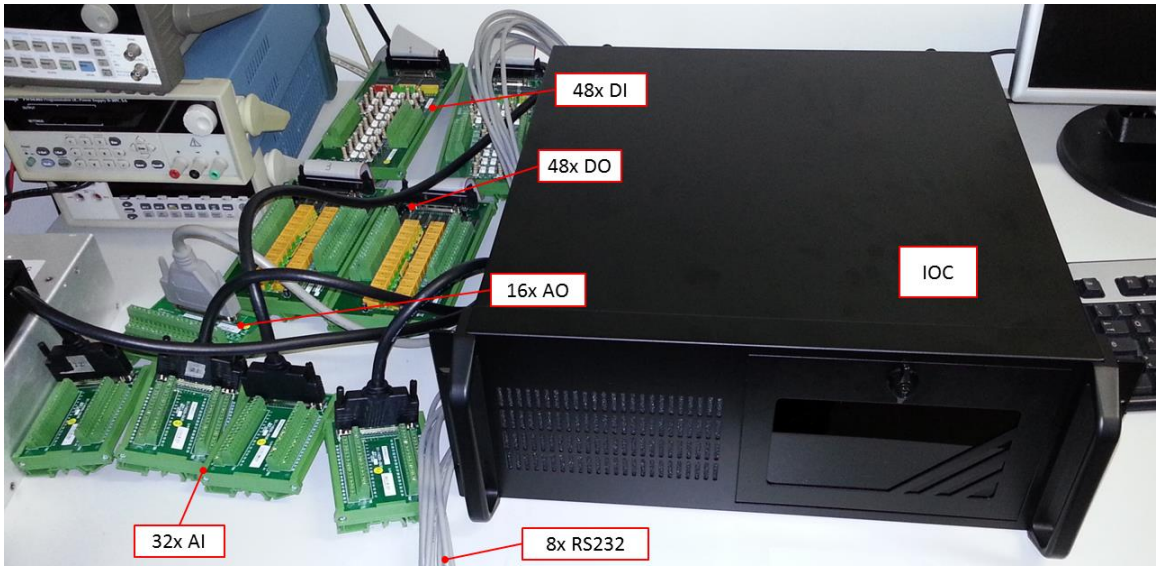


Figure 4.37. The IOC for the LNL ECR ion source.

The GUI was developed using CSS, and it is available on a remote control room. The operator can see the status of all the system as well as setting all the necessary parameter from this interface. Figure 4.38 presents a screenshot of this interface.

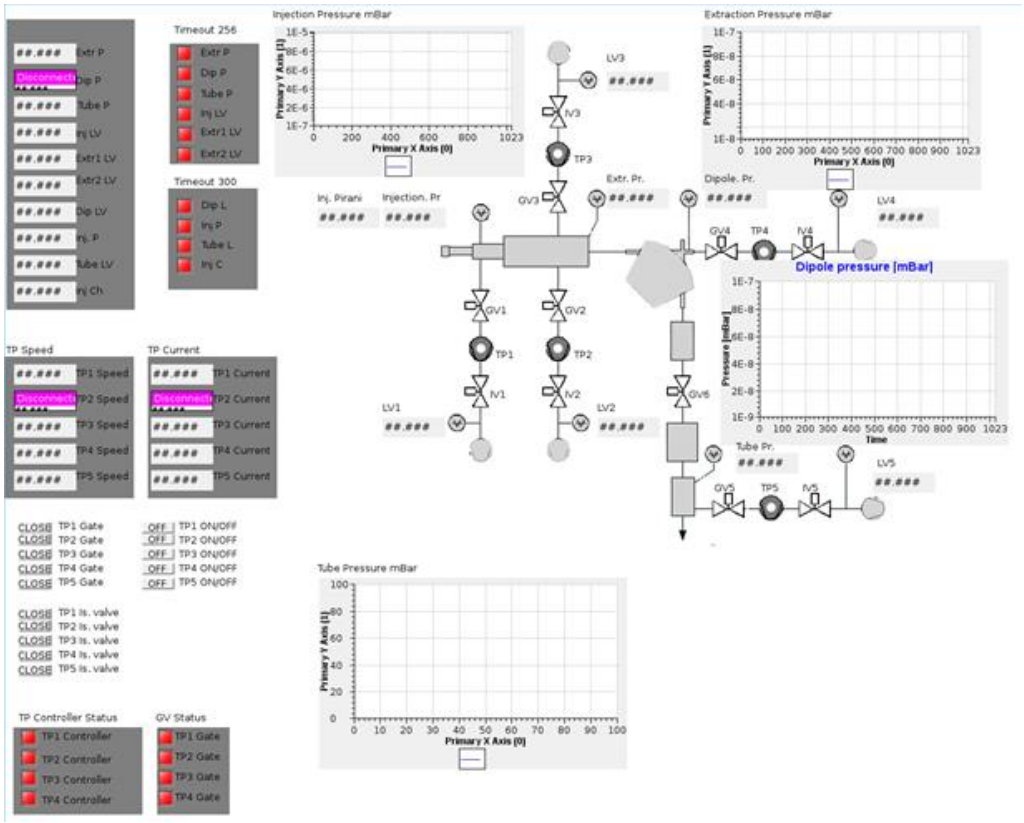


Figure 4.38. LNL ECR EPICS GUI.

4.6 Conclusions

Standardization of the control systems of the accelerator complex at LNL is being addressed with the development of the new EPICS IOC.

From the hardware point of view, it will be a basic construction block. It will be based on highly integrated Computer-on-Modules, installed on a custom tailored carrier board, equipped with all the necessary IO interfaces for satisfying the requirement of all the foreseen control systems. On the other hand, EPICS as a framework will bring a homogeneous software architecture with added benefits as total interoperability between the systems, and essential services as data archiving and logging.

Tests were carried out in order to determinate the performance of the IOC. It was shown that the ADC inputs could reach high sample rates, around 4.4 KSample/s at the cost of high CPU usage. Nevertheless, good tradeoffs are available; a good choice for the LNL applications is the rate of 900 Sample/s using only 18% of CPU. Furthermore, sample rates of tens of Samples/s could be easily reached, with negligible use of CPU.

High ENOB was obtained on the ADC inputs, due the effect of the oversampling signal-to-noise gain. Values of 19 bits were achieved. For the higher frequency that was study (400 Hz), good values of ENOB were also obtained, around 15.4 bits. Additionally, the obtained bias error was lower than 3 mV.

Similarly, for the DAC outputs, resolution of around 14.9 bits were estimated, at DC levels, with bias error lower than 30 mV.

In order to validate this hardware platform as suitable for developing the control systems at LNL, prototype IOCs were developed and implemented using commercial devices. These IOCs have been tested under real applications at LNL, with the implementation of four control systems: the beam diagnostic data acquisition, the Front-End beam extraction and focalization, the magnetic beam steerer and the ECR source systems. All the installed prototypes have shown that the hardware platform is suitable for the development of the LNL control systems using EPICS.

Although the performance is not optimal, the prototype IOCs have shown that they are suitable for controlling the selected systems, satisfying the operation requirements. However, much higher efficiency is expected from the custom IOC design.

Chapter 5

PLC based control system

5.1 Introduction

As previously stated, the control system for SPES have been divided into two main categories, that is, controls based on native EPICS IOCs or on PLCs. PLCs were selected for critical application, from a safety point of view, when a system with high affability is mandatory, which would be very difficult to guarantee using an IOC.

The sub-systems that will be controlled using PLCs are mainly those included on both, Machine and Personal Protection Systems (MPS and PPS).

In this chapter, a general overview of the PLC based control system for SPES will be presented. Also, the implementation of a new personal access control system (part of the PPS) for the LNL accelerator complex will be described, indicating also its integration to the overall control system architecture.

5.2 The SPES PLC based control system

For the SPES project, the systems that will be controlled using PLCs are the cyclotron, vacuum, room ventilation, gas recovering, target water cooling, target handling and storage, bunker gates, and the fire detection/extinguishing systems. For all the new control system for SPES, additionally, it have been decided to use exclusively PLC from Siemens, in order to have a homogenous PLC communication network.

For each system in this category, one (or more) PLC will be in charge of the controls of that system. Moreover, for most of the system on the SPES project, each one of this system will be in turn divided into two part, one related to the machine operation and the other to the safety functions. In general, the machine operation will be carried out by conventional PLCs, while for the safety functions it will be required the use of certificated PLCs, capable of reaching a Safety Integrity Levels (SIL) of 3, according to the IEC61508 international standard [47] [48]. As an example, Figure 5.1 present the architecture of the

control system for the cyclotron system, following the subdivision between machine operation and safety controls.

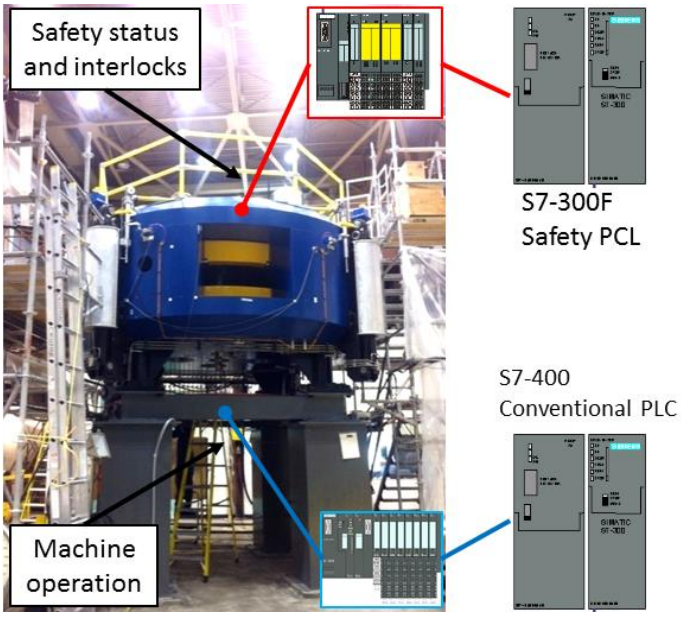


Figure 5.1. The SPES cyclotron PLC based control system. (A conventional PLC controls the operation of the machine, while a safety PLC is used for safety interlock functions).

For each system, its PLC will be in charge of the controls, in an autonomous way. However, EPICS will be used a main supervisor for all the systems. In this way, one (or more) interface IOC will be implemented in order to access the status of the system through the PLC. The IOC(s) will access the data from the PLC memory and then will make them available as EPICS PVs. The IOC will access the PLC data using the PROFINET [49] protocol, a native protocol on Siemens PLCs. If required, it would be possible for a PLC to know the status of another system; for this, the IOC will be in charge on writing the required information into a well define memory location inside the PLC, with the required status information. Figure 5.2 presents a diagram of the integration of the PLC network into EPICS.

On the other hand, for the safety part of the system, the architecture is shown on Figure 5.3. A main safety PLC will read the status of all the systems, communicating with the local safety PLCs. This main PLC will then calculate the overall system conditions, and give according interlock condition to each system, individually. For the communication, PROFISAFE [50] was chosen which it is certificated to reach a SIL3 level. The main safety PLC will also have a PROFINET interface from which an interface IOC will export the system status to EPICS.

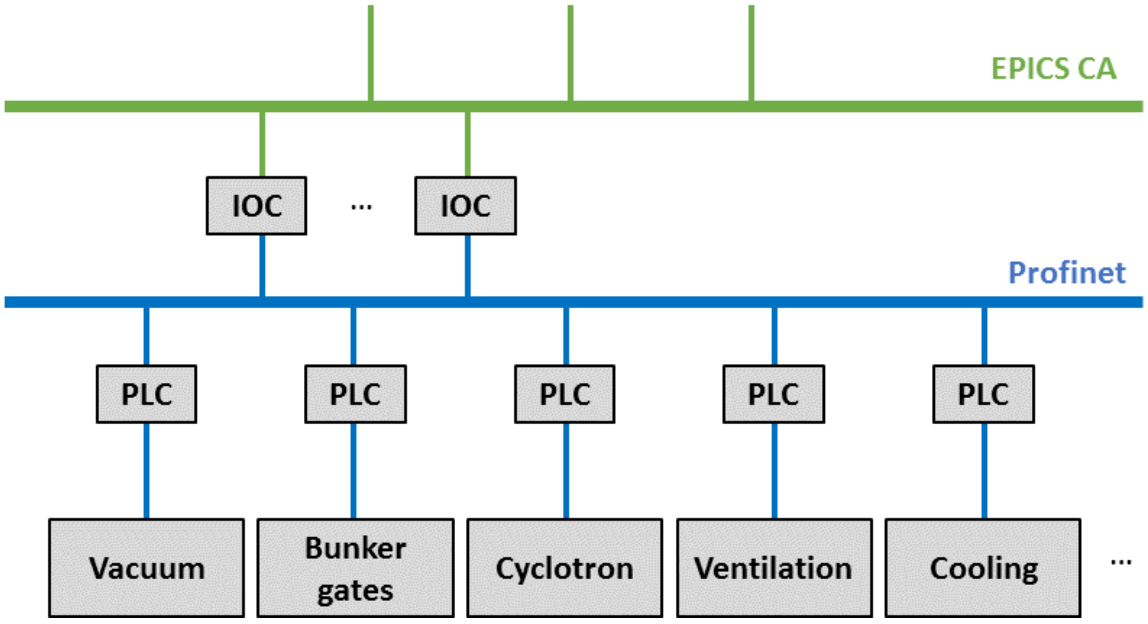


Figure 5.2. Integration of the PLC based control system to the EPICS network.

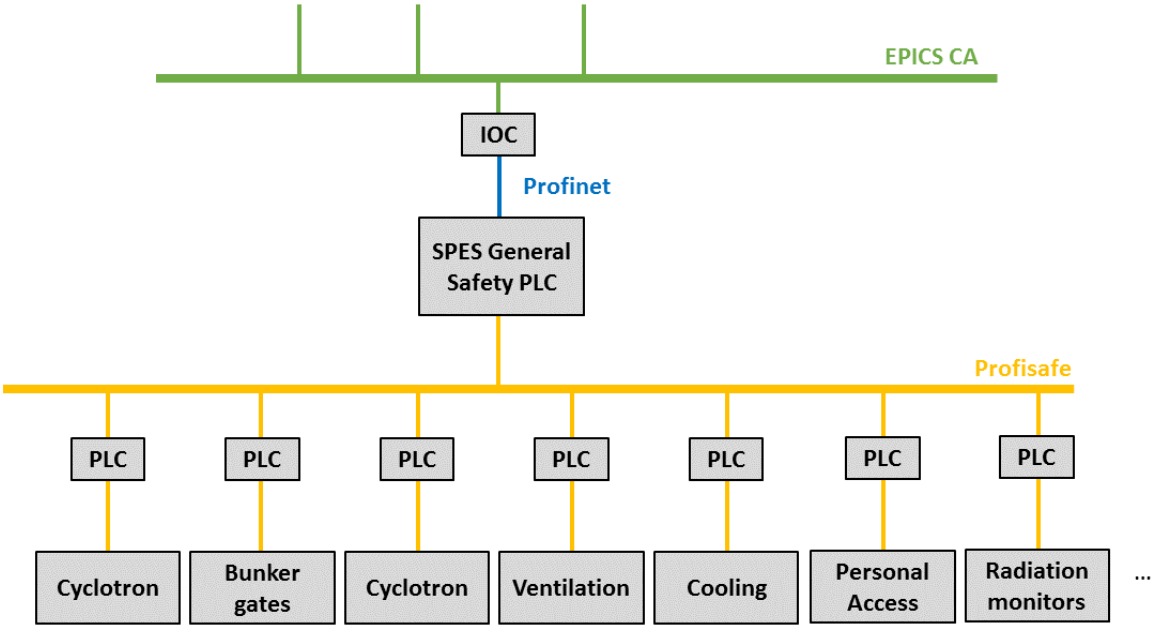


Figure 5.3. Safety control system architecture and it integration to EPICS.

Even though the described architecture have been designed for the SPES project, it will be ported to some of the system of the LNL accelerators. This will ease the final integration of both system, while also bringing homogeneity to the control systems at LNL. Two system have already been selected for their migration to this new architecture, that is, the vacuum and the personal access control system.

In the context of this thesis, the personal access control system of the LNL accelerators was chosen for a complete renewal, following the guidelines of SPES [51] [52]. This new implementation will be describe on the following chapter.

5.3 Implementation of the Personal Access Control system for the Accelerator Areas at LNL

The system, as its name indicates, is in charge of inhibit the access of persons to certain areas of the accelerator or experimental areas, depending on the beam or RF conditions. For this, patrol and lockout procedures must be performed by the operators. The control system verifies the correct sequence of all the procedures, enabling to operate the accelerator only when the personal absence is guarantee. Furthermore, the system is able to interlock the beam extraction or RF amplifiers, is a locked area is corrupted, or some specific alarm are detected.

All the accelerator facilities and experimental halls are divided into 20 areas approximately, with hundreds of emergency push button, search buttons, contacts and limit switches, gate's electro-locks, as well as visual and acoustic signaling devices. Moreover, a large number of combination of conditions must be continuously checked. All these factors made this a very complex control system.

The is currently completely install and operation at LNL. On the following chapter, it will be described.

5.3.1 Hardware implementation

For this personal protection system, as opposed to SPES, it was not required any specific SIL target. This is due the fact that the risk levels associate with this system are very low, unlike the ones associate to SPES.

The new control system is based on a fail-tolerant, redundant CPU, central PLC, connected to seven remote IO islands. This remote island are distributed on the accelerator facility, and are in charge of reading and writing the corresponding control elements.

The communication between the PLC and the IO modules is done using an optic fiber Ethernet network, in a redundant ring configuration. To accomplish the network redundancy, special Ethernet switches, which support the Rapid Spanning Tree Protocol (RSTP) were used. On the other hand, a second

redundant network was used for the supervisor system. This network is physically distinct to the IO communication one. All the communications use the Modbus/TCP protocol.

With this configuration, the system is tolerant to a single failure both, on the PLC CPU, and on the communication network elements (cable, optic fiber, or Ethernet switch). A general overview of the system is presented on Figure 5.4.

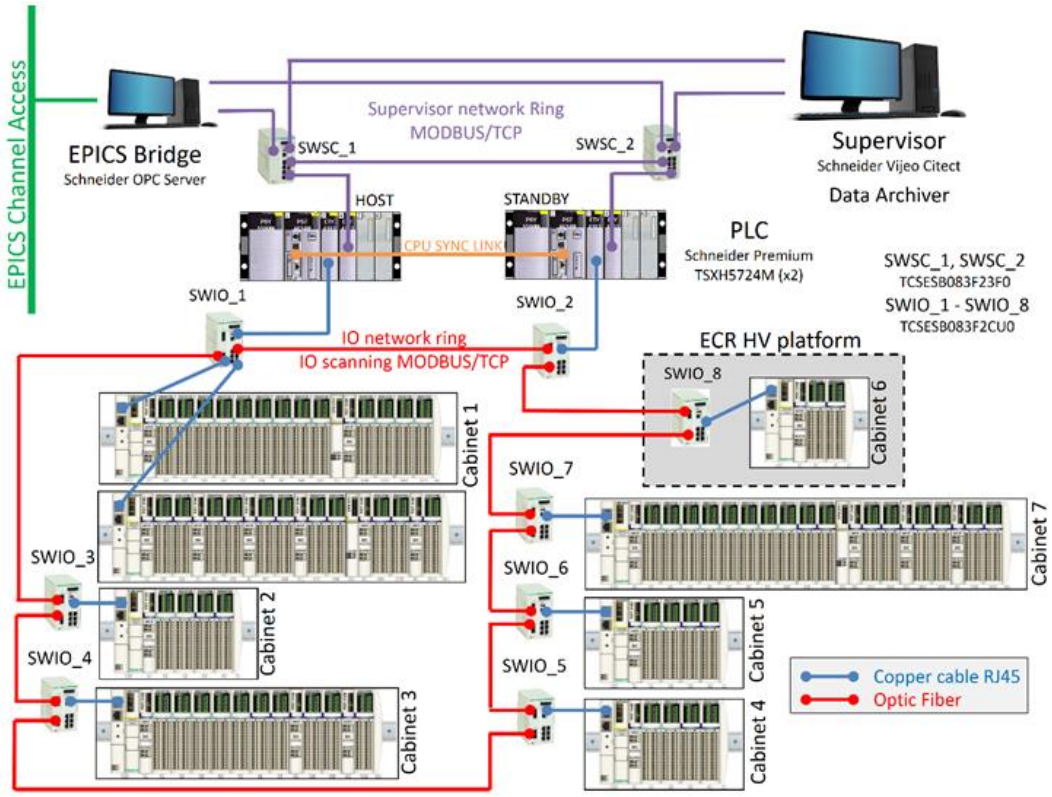


Figure 5.4. Overview of the LNL personal access control system.

The selected PLC is a TSXH5724M from the Schneider Electrics' Premium family. Two CPU work in HOST/STANDBY configuration, were the HOST CPU is normally in charge of the system; when a failure is detected on this CPU, the STANDBY takes control of the system.

The distributed IO islands use modules of the STB family from Schneider Electrics. Each island is formed by one communication module in conjunction with digital IO units. The sensors and actuators (door switches, emergency push buttons, signal lights, door locks, etc.) are connected to these IO channels.

On Figure 5.5 it is shown the main cabinet which hold the PLC system, while Figure 5.6 reports two of the seven remote IO island cabinets. These cabinets are currently installed on the LNL accelerator facility.

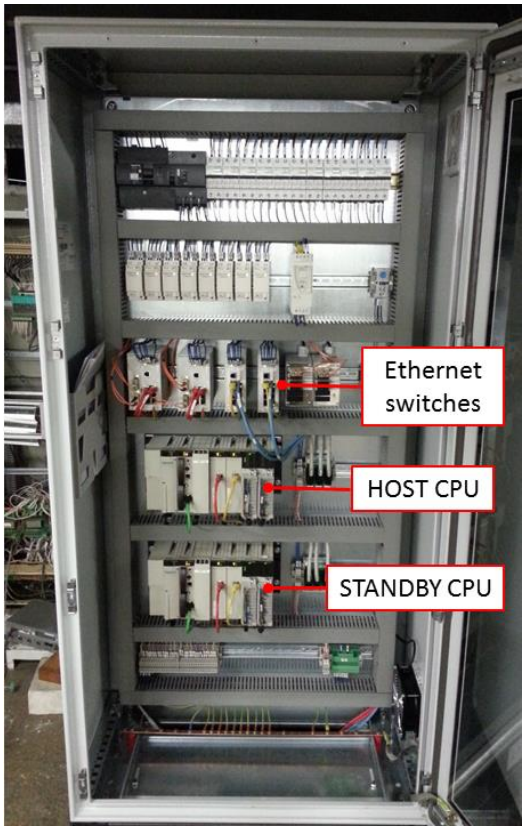


Figure 5.5. The PLC cabinet of the access control system.

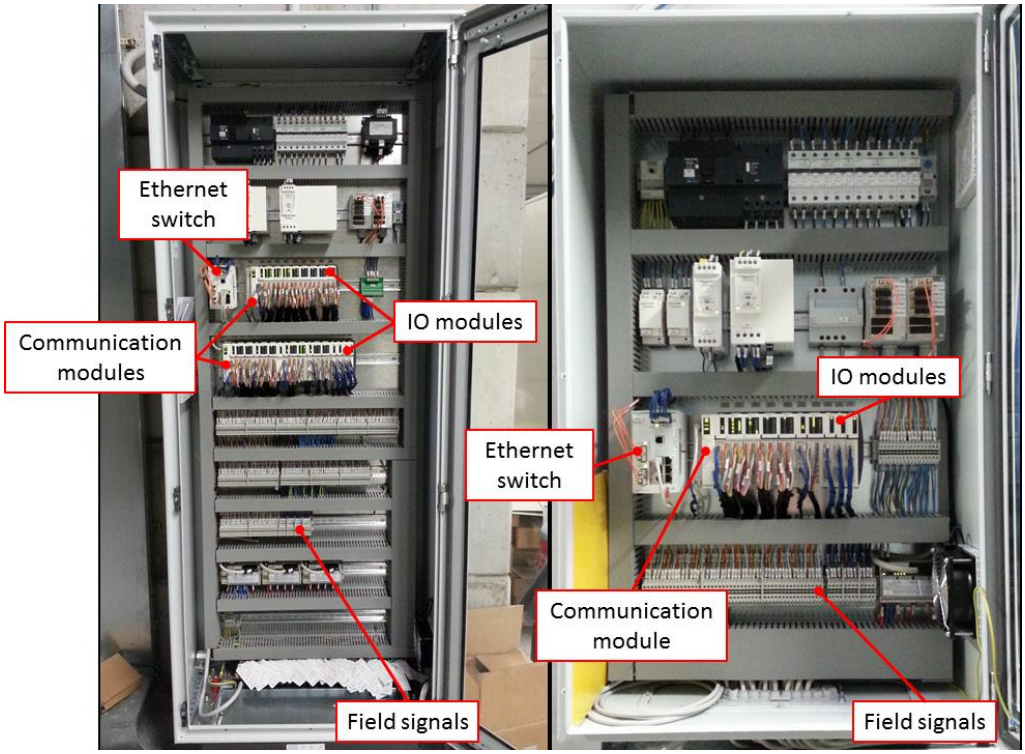


Figure 5.6. Two of the seven remote IO cabinets of the access control system.

5.3.2 Software Implementation

The PLC software was developed using the Schneider's Integrated Development Environment (IDE), called Unity Pro, using programming languages compliant with the IEC61131-3 international standard. On the other hand, the main user interface of the system was developed using the commercial Supervisory Control and Data Acquisition (SCADA) system, called Vijeo Citect from Schneider Electrics. Besides acting as a GUI, on this SCADA system basic services as authentication, logging and data archive were implemented.

The software was developed on a modular way. Functional control blocks were create in order to manage all the information (input/output signals, status, interlock, alarms, etc.) related to specific parts of the system. These blocks have input and output interfaces were the respective variables are attached to, while inside they are implemented all the logic algorithms. Then blocks are user for controlling all the existing devices, of the same type. The advantages of this approach are a fast addition of new elements on the system, reduce maintenance efforts, and homogenous control algorithms for all the elements.

On Figure 5.7 there are presented the implementation of two part of the control software using these functional blocks. It is shown the software part in charge of managing the patrol procedure of one of the area (left), and the part that verifies the status of all the emergency push-buttons of the same area (right).

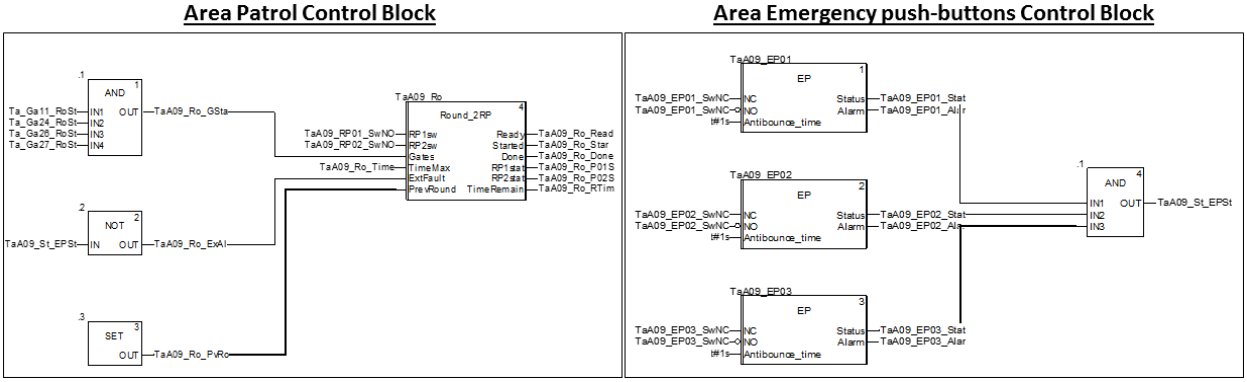


Figure 5.7. Functional blocks implemented on the PLC software. (On the left, it is presented the part of the software in charge of the patrol procedure of an area; while on the right, the part in charge of verifying the status of emergency push-button of an area).

This modularity design have been implemented on both, the PLC and SCADA software. For each PLC functional block, an analogous functional block exist on the SCADA side. Figure 5.8 presents a couple of examples, one for the gate (top) and the other for the faraday cup control blocks (bottom).

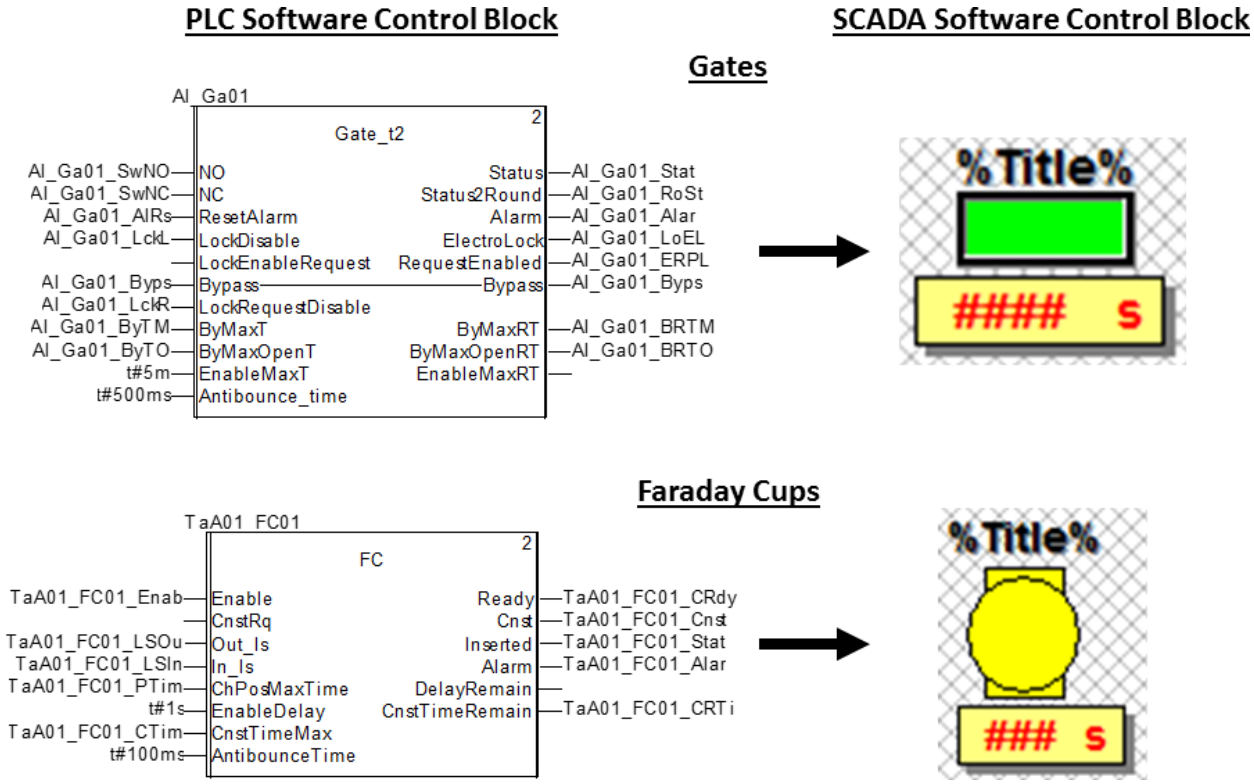


Figure 5.8. Functional bock, both of the PLC and SCADA side. (On top, it is presented the functional blocks for managing a gate; while on the bottom, the ones for managing a Faraday Cup).

In order to automatize the creation of the variables, both for the PLC and SCADA software, EXCEL tables were created. On these tables, the user introduces the device parameters (number of elements, functional area of the accelerator, description, etc.) and then the table creates the variables automatically. The PLC variables are exported to XML files, which are in turn importer into Unity Pro. On the other hand, the SCADA variables are introduce on the variable database, using an EXCEL plugin. This procedure is illustrated on Figure 5.9.

An EPICS IOC was developed for interfacing this system to the EPICS CA. The IOC was deployed on a PC with two Ethernet interfaces, one dedicated to the PLC communication, and the other to the EPICS CA. It communicates with the PLC using the EPICS driver support for Modbus [37], reading some defined variables from the memory of the PLC and writing them into EPICS record. The status are then available to the rest of IOC on the network as PVs. For this system, a GUI was developed using CSS.

Figure 5.10 shows the user interface developed using the SCADA software, while Figure 5.11 presents the user interface developed using CSS.

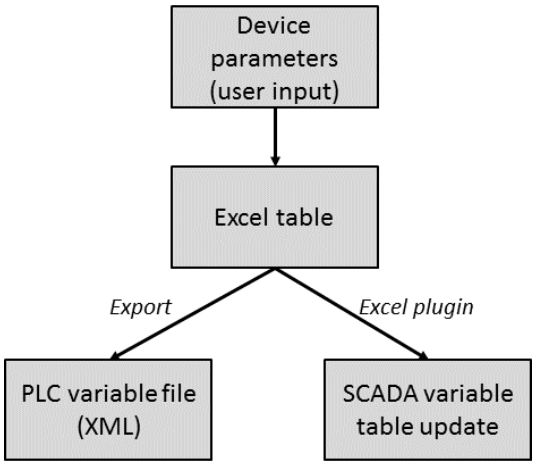


Figure 5.9. Automatic creation of variables procedure.

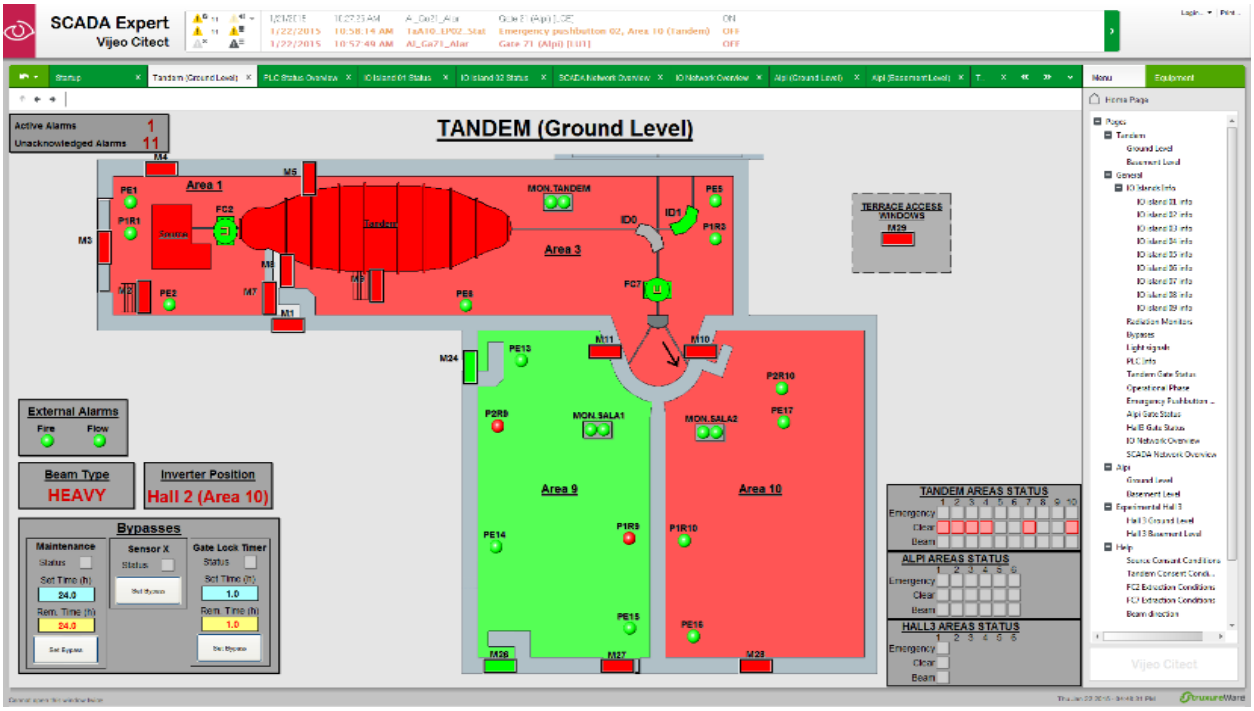


Figure 5.10. The main user interface of the access control system. It was developed using the Vijeo Citect commercial SCADA software.

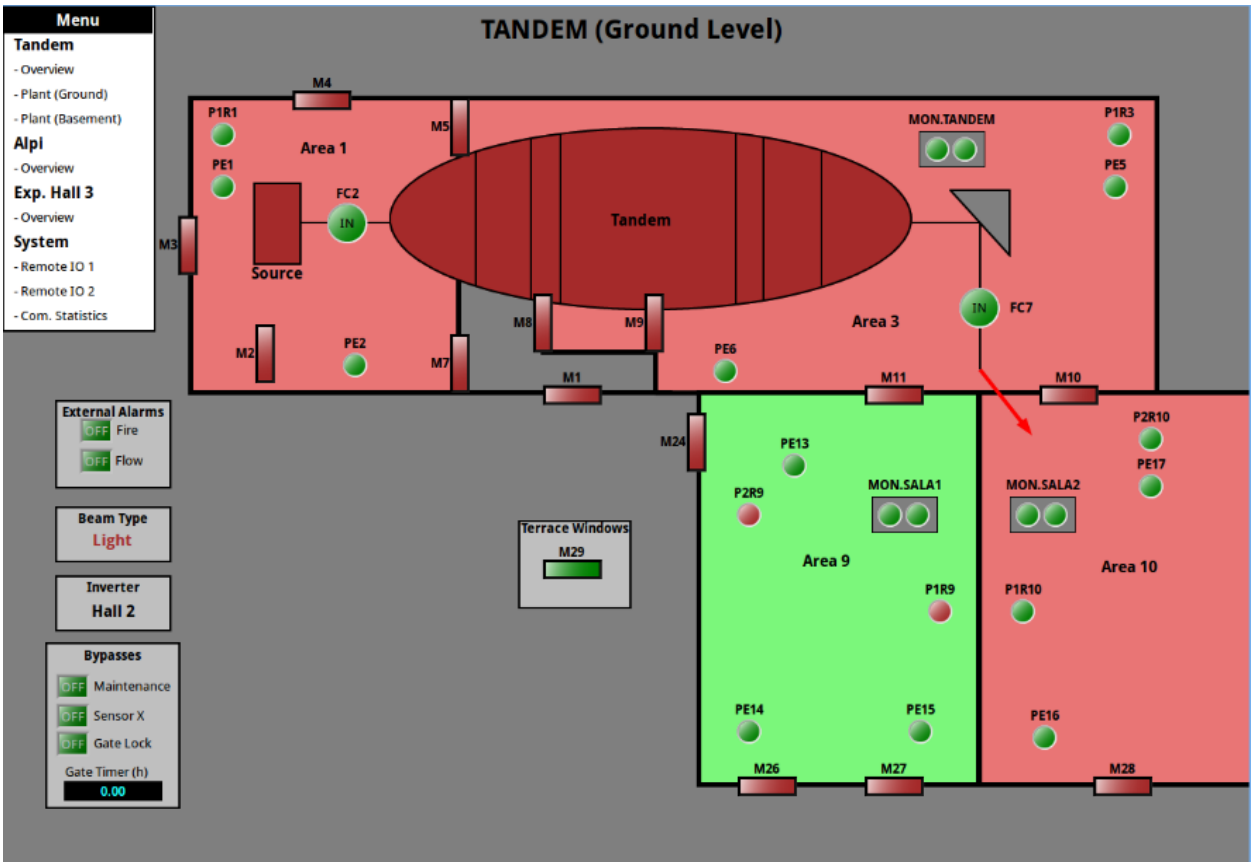


Figure 5.11. The EPICS GUI of the access control system. It was developed using the CSS EPICS client.

5.4 Conclusions

In this chapter, it was introduce the PLC based control system for SPES. There were indicated the motivation for this kind of implementation, as well as the systems that have been selected as part of this category. Moreover, it was showed how this system is, in turn, sub-divided into the control part dedicated to the machine operation, and the part related to the safety functions. An overview of both sub-system were presented and described. In addition, it was illustrated the system integration to the general control system for SPES, based on EPICS.

In addition, it was explained how this architecture is being ported also to the LNL accelerator facilities. One of the system selected for this migration is the personal access control system. In that sense, a new control system was designed and implemented on the context of this thesis. Details about its implantation were presented, as well as its integration to the overall control system.

Conclusions

SPES it is currently one of the major INFN project, aiming to the production of RIBs and radioisotopes for nuclear and medicine applications. It is being constructed at the LNL in Legnaro, Italy. This kind of facility have brought many technology challenges on many fields. In this thesis, challenges on the control system were addresses. The project was introduced on chapter 1.

EPICS was chosen as the general framework for the control system. In general terms, the system is divided into two main groups; on one part there is the safety related control system which will be based on PLC, while on the other hand there are the rest of the accelerator instrumentation which is based on native EPICS IOCs. The EPICS CA will servers as a main communication trunk among all the systems, where essential services such as data archiving, logging and remote GUIs, in conjunction with basic networking services will be deployed. An introduction to these aspects was presented on chapter 2. In this thesis, developments of both part s of the control system were presented.

The arrival of the SPES project to the LNL has also trigger an upgrade campaign for the existing control system of the accelerator complex. Thus, standardization is a key factor on the current control system developments. This aspect was the main topic of this thesis, addressed by the developing custom EPICS IOCs, which will be used as basic construction blocks for building the entire control system.

On Chapter 3, it was presented the first version of a custom EPICS IOC developed at LNL. It is based on the computer board Raspberry Pi in conjunction with standard USB converters and custom IO expansion boards, for adding the IO interfaces necessary for controlling the accelerator instrumentation.

Four types of expansion boards where developed for the applications found on the off-line laboratory: a 1-channel, current-signal acquisition board; a 40-channel, current-signal acquisition board; a stepper motor controller board; and a general-purpose IO board. Using these boards, three different IOCs were developed and used for the implementation of four different control systems, namely, the beam diagnostic data acquisition system, the mass separator control system, the vacuum measurement acquisition system, and the PLC-to-EPICS interface.

The experimental results demonstrated that this IOC is able to acquired signal at a maximum rate of 84 Samples/s, with an ENOB of almost 15 bits, and bias error lower than 60 mV. For current signals, the system capable of acquiring current from 100pA until 3.5mA, with less than 2% of bias error for most cases. On the other hand, The IOC can generate analog signals with an estimate resolution of almost 15bits, and bias errors lower than 16 mV.

While a low-cost solution is completely valid for the off-line laboratory, SPES and the LNL accelerators require a more robust and standard solution, which can operated under industrial environments and constructed with hardware components that can be available on the marked for the major part of the life cycle of the accelerators. This motivation caused the evolution of the first version of the IOC, into the version presented on chapter 4. This second version is based on highly integrated Computer-on-Modules. All the necessary IO interfaces for satisfying the requirements of all the foreseen control system will be integrated on a tailored carrier board for the COM. In order to validate the choice of the hardware platform, there were developed prototypes of the IOC using commercial development boards.

The experimental results showed that the IOC is capable of acquiring analog signals at a maximum rate of 4.4 KSample/s, with ENOB from values that goes from the 19 bits (for input signals of some Hz) until the 15 bits (for input signals of 400 Hz), and bias error lower than 3 mV. On the other hand, the generation of analog values were possible reaching an estimated resolution of 15 bits, and bias errors lower than 30 mV.

Four application at LNL were targeted for the implementation of a control system using these prototypes: the beam diagnostic data acquisition, the Front-End beam extraction and focalization, the magnetic beam steerer and the ECR source systems. All the installed prototypes have validated that the hardware platform is suitable for the development of the LNL control systems using EPICS.

Finally, developments on the context of the PLC based control system were presented on Chapter 5. It was presented the design and implementation of a new personnel access control system for the LNL accelerator complex. This renewal is a mandatory step for the system integration between the existing accelerator facility at LNL and SPES. Additionally, it was showed the EPICS interface implemented for this system, a fundamental element on its intergartion with the general EPICS control system. This approach will be applied to all future PLC control systems for the SPES project.

The development and implementation of a control system for a project as SPES is a colossal task. With the result obtained during this thesis period, an important step towards the standardization of the control system was given.

In the near future, a final version of the custom IOC hardware platform will be in production; consequently, development on the software layer of the system will be necessary. This new version of IOC will be widely used at LNL. The results obtained using prototypes presented on this thesis, will serve as a guide for these developments.

On the other hand, developments on the PLC control system will be necessary for the critical system on the SPES project. A safety control network will be build, while EPICS will acts a major supervisor of the system. The results presented on this thesis will outline the path for the future developments on this area.

The standardization of all control system at LNL will ultimately help to considerably reduce costs and maintenance efforts, not only during the implementation stage, but also during the whole project life spam, while bringing, at the same time, homogeneity to the whole system.

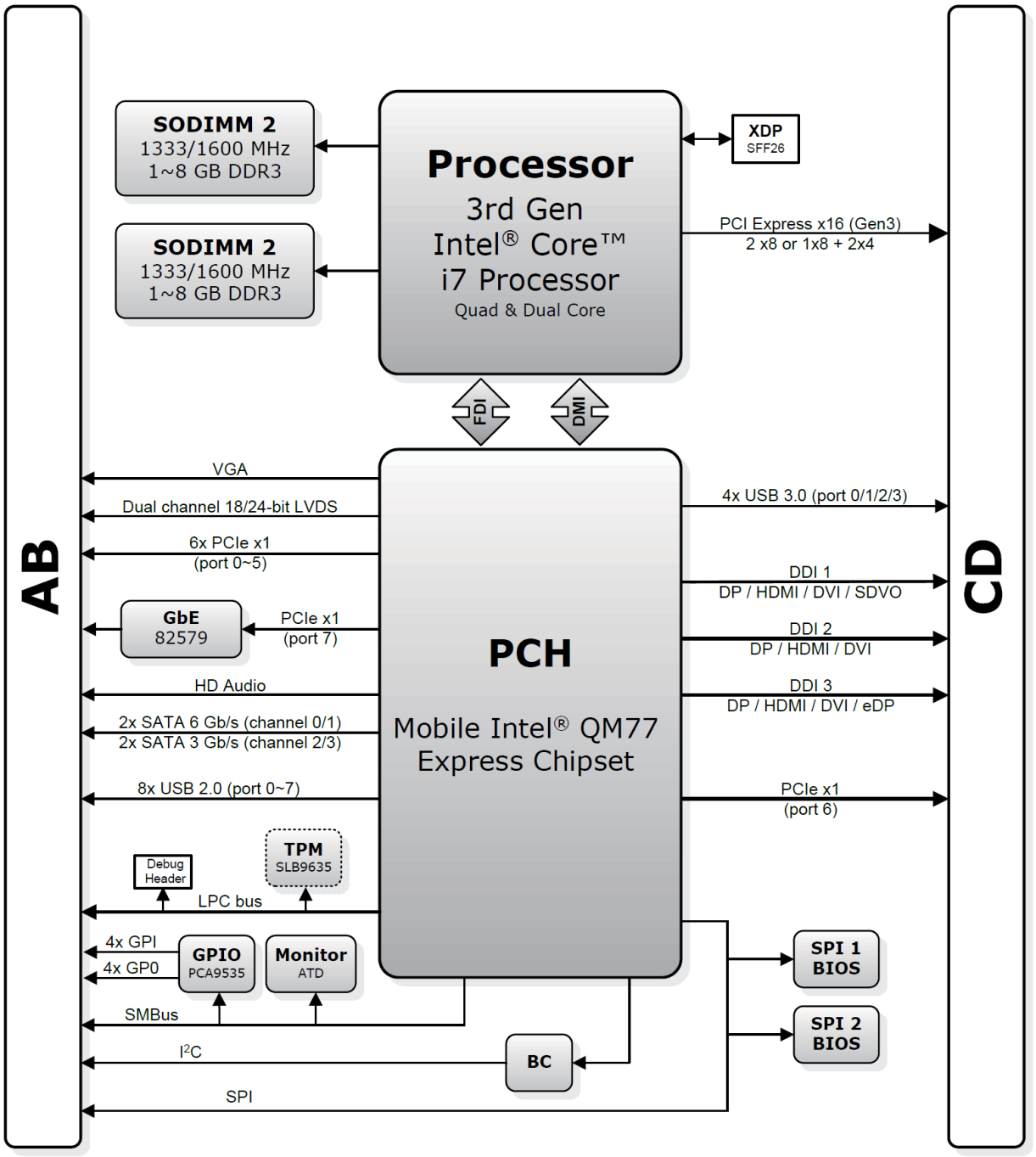
Appendix A: Adlink's boards technical specifications [38] [39] [40]

COM Express-IB-i3-3120ME

Core System	
CPU	3rd Generation Intel® Core™ i7/i5/i3, 22nm process, BGA type Intel® Core™ i3-3120ME 2.4GHz, 3MB L3 cache, 35W (2C)
Memory	Dual channel 1333/1600 MHz DDR3 non-ECC memory up to 16GB in dual stacked SODIMM socket
Chipset	Mobile Intel® QM77 Express Chipset
L3 Cache	6MB (i7-3615QE and i7-3612QE), 4MB (i7-3555LE and i7-3517UE), 3MB (i5-3610ME, i3-3120ME and i3-3217UE)
BIOS	AMI EFI with CMOS backup in 16 Mbit SPI flash
Hardware Monitor	Supply voltages and CPU temperature
Debug Interface	XDP SFF-26 extension for ICE debug
Watchdog Timer	Programmable timer range to generate RESET
Expansion Busses	PCI Express x16 (Gen3) bus for discrete graphics solution or general purpose PCI Express (2 x8 or 1 x8 with 2 x4)
	8 PCI Express x1: Lanes 0/1/2/3/4/5/6 are free, lane 7 is occupied by GbE
	LPC bus, SMBus (system) , I2 C (user)
Video	
Integrated in Processor	ntel® HD Graphics 4000 at 650–1200 MHz (depending on processor)
Integrated Video	DirectX 11, OpenGL 3.1, OpenCL 1.1
Feature Support	Intel® Clear Video HD Technology Advanced Scheduler 2.0, 1.0, XPDM support DirectX Video Acceleration (DXVA) support for full AVC/VC1/MPEG2 hardware decode
VGA Interface	Analog VGA support with 300 MHz DAC Analog monitor support up to QXGA (2048 x 1536) and VGA hot plug
LVDS Interface	Dual channel 18/24-bit LVDS
Digital Display Interface	Three DDI ports supporting HDMI/DVI/DisplayPort or SDVO
Audio	
Chipset	Integrated in Mobile Intel® QM77 PCH
Audio Codec	On Express-BASE6 carrier (ALC886)
Ethernet	
Chipset	Intel® Gigabit Ethernet PHY WG82579LM
Interface	10/100/1000 Mbps Ethernet
I/O Interfaces	
Chipset	Integrated in Mobile Intel® QM77 PCH

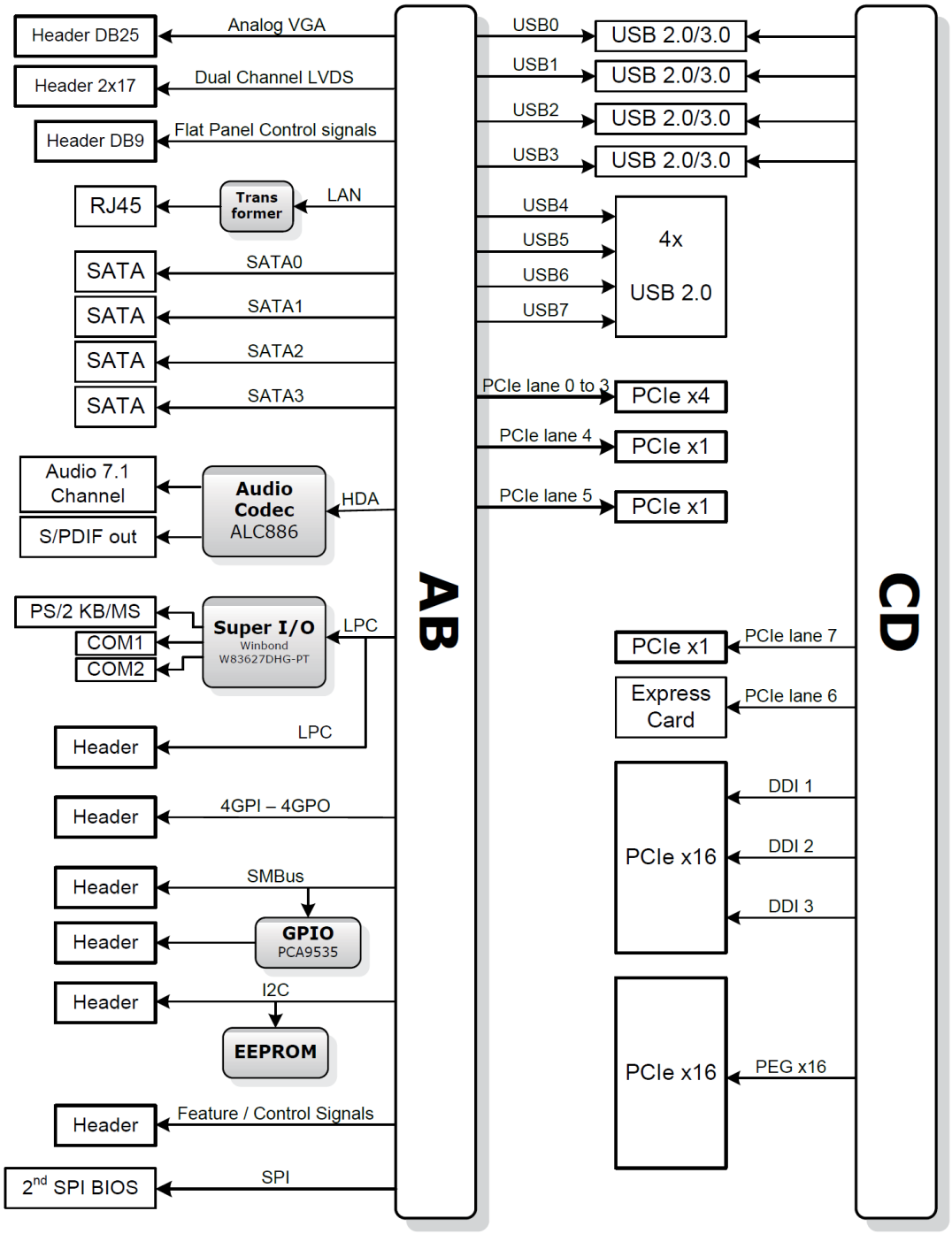
USB	4 ports USB 3.0 (USB0~3) and 4 ports USB 2.0 (USB4~7)
SATA	Supports two SATA ports at 6 Gb/s and two ports at 3 Gb/s with support for RAID 0,1,5,10
Super I/O	
	Connected to LPC bus on carrier if needed
Power	
Input Power	8.5~20V only (AT), 8.5~20V and 5Vsb (ATX)
Power States	Supports S0, S1, S3, S4, S5
Smart Battery Support	Yes
Power Consumption	40W with i7-3612QE and 4GB memory typcia
Mechanical and Environmental	
Form Factor	PICMG COM.0: Rev 2.1 Type 6
Dimension	Basic size: 125 mm x 95 mm
Operating Temperature	Standard: 0°C to +60°C
Storage Temperature	-20°C to +80°C
Humidity	90% at +60°C
Shock	15G peak-to-peak, 11ms duration, non-operating
Vibration	Non-operating: 1.88Grms, 5-500Hz, each axis Operating: 0.5Grms, 5-500Hz, each axis
Compatibility	COM Express Type 6, Basic form factor 125mm x 95mm
Certification	CE, FCC, HALT

Functional Diagram



Express-BASE6 reference carrier board

Functional Diagram



DAQe-2214

Analog Input	
Resolution	16 bits, no missing codes
Number of channels	16 single-ended or 8 differential (software selectable per channel)
Channel gain queue size	512
Maximum update rate	250 kS/s
Programmable gain	1, 2, 4, 8
Bipolar input ranges	±10 V, ±5 V, ±2.5 V, ±1.25 V
Unipolar input ranges	0-10 V, 0-5 V, 0-2.5 V, 0-1.25 V
Offset error	±1 mV
Gain error	±0.06% of FSR
Input coupling	DC
Overvoltage protection	Power on: Continuous ±30 V, Power off: Continuous ±15 V
Input impedance	1 GΩ /100 pF
CMRR (gain = 1)	83 dB
Settling time	4 µs to 0.01% error
-3 dB small signal bandwidth	600 kHz (@Bipolar +/-10V Gain=1)
(@Bipolar +/-10V Gain=1)	
Trigger sources	Software, external digital/analog trigger, SSI bus
Trigger modes	Pre-trigger, post-trigger, middle-trigger, delay-trigger, and repeated trigger
FIFO buffer size	1 k samples
Data transfers	Polling, scatter-gather DMA
Analog Output	
Number of channels	2 voltage outputs
Resolution	- 12 bits
Output ranges	- 0-10 V, ±10 V, 0-AOEXTREF, ±AOEXTREF
Maximum update rate	- 1 µs
Slew rate	- 20 V / µs
Settling time	- 3 µs to ±0.5 LSB accuracy
Offset error	- ±2 mV
Gain error	- ±0.04% of max. output
Driving capacity	- ±5 mA
Stability	- Any passive load, up to 1500 pF
Trigger sources	- Software, external digital/analog trigger, SSI bus
Trigger modes	- Post-trigger, delay-trigger, and repeated trigger
FIFO buffer size	- 1 k samples
Data transfers	- Programmed I/O, scatter-gather DMA
Digital I/O	
Number of channels	24-CH 8255 programmable input/output
Compatibility	5 V/TTL
Data transfers	Programmed I/O
General-Purpose Timer/Counter	

Number of channels	2
Compatibility	16 bits
Data transfers	5 V/TTL
Number of channels	40 MHz, external clock up to 10 MHz
Auto Calibration	
Onboard reference	+5 V
Temperature drift	±2 ppm/˚C
Stability	±6 ppm/1000 Hrs
General Specifications	
Dimensions	175 mm x 107 mm (not including connectors) (DAQ-2213/2214) 168 mm x 107 mm (not including connectors) (DAQe-2213/2214)
Connector	68-pin VHDCI female x 2
Operating temperature	0 to 55˚C
Storage temperature	-20 to 70˚C
Humidity	5 to 95%, non-condensing
Power requirements	+5 V 1.2 A typical

PCIe-6216

Voltage Output	
Number of channels	16
Resolution	16 Bit
Monotonicity	15 Bit typical
Output ranges	±10 V
Slew rate	26 V/µs typical
Settling time	130 µs typical (20 V step)
Gain Error	±0.2% maximum
DNL	±1 LSB typical
Output driving capacity	±5 mA maximum
Output initial status	0 V
Data transfer	programmed I/O
Digital I/O	
Number of channels	4 inputs and 4 outputs
Compatibility	5 V/TTL
Data transfers	programmed I/O
General Specifications	
I/O connector	One 37-pin D-sub female
Operating temperature	0˚C to 50˚C (32˚F to 122˚F)
Storage temperature	-20˚C to 80˚C (-4˚F to 176˚F)
Relative humidity	5% to 95%, non-condensing
Power requirements	+5V (1.2 A typical) +12V (280 mA typica)

Appendix B: Software source code

DAQe-2214 ADC board asynDriver interface driver

```
/*
 * drvDAQE2214 : Driver for ADLINK DAQe-2214 acquisition board using Mark Rivers' AsynPortDriver
 *
 * Copyright © 2013 Jesus Vasquez
 *
 * This file is part of drvDAQE2214.
 *
 * drvDAQE2214 is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * drvDAQE2214 is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with drvDAQE2214.  If not, see <http://www.gnu.org/licenses/>.
 *
 * Version 2.0
 * This version read analog and digital inputs by a contiuos polling thread
 * respectively at a fixed period and then
 * (so, the digital input records must specify "I/O Intr" on their SCAN field)
 *
 * Author:          Jesus Vasquez
 * Created on:      Dic 06, 2013
 * Contact:         jesus.vasquez@lnl.infn.it
 *
 */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <algorithm>


#include <epicsTypes.h>
#include <epicsTime.h>
#include <epicsThread.h>
#include <epicsString.h>
#include <epicsTimer.h>
#include <epicsMutex.h>
#include <epicsEvent.h>
#include <iocsh.h>

#include "drvDAQE2214_v2.h"
#include "asynPortDriver.h"
#include <epicsExport.h>

#include "d2kdask.h"

int DAQE2214::cardId_ = -1;
const char * DAQE2214::driverName="DAQE2214_v2";

DAQE2214::DAQE2214( const char *portName, int boardNum, int ai_polltime, int di_polltime,
          int ch0_size, int ch1_size, int ch2_size, int ch3_size, int ch4_size, int ch5_size, int ch6_size, int ch7_size,
          int ch8_size, int ch9_size, int ch10_size, int ch11_size, int ch12_size, int ch13_size, int ch14_size, int ch15_size)
          : asynPortDriver(   portName,
                              MAX_SIGNALS,
                              NUM_PARAMS,
                              asynInt32Mask | asynDrvUserMask | asynFloat64ArrayMask | asynUInt32DigitalMask, // Interface Mask
                              asynFloat64ArrayMask | asynUInt32DigitalMask | asynInt32Mask,         // Interrupt Mask
                              ASYN_MULTIDEVICE | ASYN_CANBLOCK,                                     // asynFlags
                              1,                                                                    // Autoconnect
                              0,                                                                    //Default priority
                              0),                                                                   // Default stack size
            boardNum_(boardNum),
            forceCallback_(1)
{
          asynStatus status;
          int i;
          const char *functionName = "DAQE2214";

          for (i=0; i<NUM_ANALOG_IN; i++)
                    pData_[i] = (epicsFloat64 *)calloc(BUFFER_SIZE_MAX, sizeof(epicsFloat64));

          if (ai_polltime >= MIN_AI_POOL_TIME)
                    aiPollTime_ = ((double)ai_polltime)/1000.0;
          else
                    aiPollTime_ = ((double)MIN_AI_POOL_TIME)/1000.0;
```

```
        if (di_polltime >= MIN_DI_POOL_TIME)
                diPollTime_ = ((double)di_polltime)/1000.0;
        else
                diPollTime_ = ((double)MIN_DI_POOL_TIME)/1000.0;


        ch_size[0] =        ch0_size;
        ch_size[1] =        ch1_size;
        ch_size[2] =        ch2_size;
        ch_size[3] =        ch3_size;
        ch_size[4] =        ch4_size;
        ch_size[5] =        ch5_size;
        ch_size[6] =        ch6_size;
        ch_size[7] =        ch7_size;
        ch_size[8] =        ch8_size;
        ch_size[9] =        ch9_size;
        ch_size[10] =       ch10_size;
        ch_size[11] =       ch11_size;
        ch_size[12] =       ch12_size;
        ch_size[13] =       ch13_size;
        ch_size[14] =       ch14_size;
        ch_size[15] =       ch15_size;

        // Analog input parameters
        createParam(analogInValueString,        asynParamInt32,               &analogInValue_);
        createParam(analogInRangeString,        asynParamInt32,               &analogInRange_);
        createParam(analogInWaveformString,     asynParamFloat64Array,        &analogInWaveform_);

        // Analog output parameters
        createParam(analogOutValueString,       asynParamInt32,               &analogOutValue_);
        createParam(analogOutRangeString,       asynParamInt32,               &analogOutRange_);

        // Digital IO parameters
        createParam(digitalDirectionString,     asynParamInt32,               &digitalDirection_);
        createParam(digitalInputString,         asynParamUInt32Digital,       &digitalInValue_);
        createParam(digitalOutputString,        asynParamUInt32Digital,       &digitalOutvalue_);

        status =(asynStatus)(epicsThreadCreate("DIPoller", epicsThreadPriorityLow,
                        epicsThreadGetStackSize(epicsThreadStackMedium), (EPICSTHREADFUNC)pollerThreadC, this) == NULL);

        if (status)
        {
                printf("%s:%s: epicsThreadCreate failure\n", driverName, functionName);
                return;
        }

        status =(asynStatus)(epicsThreadCreate("AIPoller", epicsThreadPriorityLow,
                        epicsThreadGetStackSize(epicsThreadStackMedium), (EPICSTHREADFUNC)::scanTask, this) == NULL);

        if (status)
        {
                printf("%s:%s: epicsThreadCreate failure\n", driverName, functionName);
                return;
        }
}

////////////////////////////////
// + ADLINK-dipendent functions //
////////////////////////////////
int DAQE2214::DAQE2214Init(int boardNum)
{
        I16 err;

        printf("DAQe-2214 board initialization:\n");
        cardId_ = D2K_Register_Card((I16)CARD_TYPE, (I16)boardNum);
        if (cardId_ < 0)
        {
                printf("Error registering card %d, code= %d\n", (I16)boardNum, cardId_);
                exit(-1);
        }
        else
                printf("card number %d registered with ID %d\n", (I16)boardNum, cardId_);

        err = D2K_AI_Config(cardId_, 0, 0, 0, 0, 0, (BOOLEAN)TRUE);
        printf("D2K_AI_Config(card, 0, 0, 0, 0, 0, %d) = %d\n", (BOOLEAN)TRUE, err);
        if (err)
                return -1;

        err = D2K_AO_Config(cardId_, 0, 0, 0, 0, 0, (BOOLEAN)TRUE);
        printf("D2K_AO_Config(card, 0, 0, 0, 0, 0, %d) = %d\n", (BOOLEAN)TRUE, err);
        if (err)
                return -2;

        err = D2K_DIO_PortConfig(cardId_, CLK_OUTPUT_PORT, OUTPUT_PORT);
        printf("D2K_DIO_PortConfig(%d, %d, %d) = %d\n", cardId_, CLK_OUTPUT_PORT, OUTPUT_PORT, err);
        if (err)
                return -3;

        err = D2K_DIO_PortConfig(cardId_, DO_PORT, OUTPUT_PORT);
        printf("D2K_DIO_PortConfig(%d, %d, %d) = %d\n", cardId_, DO_PORT, OUTPUT_PORT, err);
        if (err)
                return -4;

        err = D2K_DIO_PortConfig(cardId_, DI_PORT, INPUT_PORT);
        printf("D2K_DIO_PortConfig(%d, %d, %d) = %d\n", cardId_, DI_PORT, INPUT_PORT, err);
```

```
        if (err)
                return -5;

        return 0;
}

int DAQE2214::readDigitalPort(int cardid, int addr, int *value)
{
        I16 Status;
        U16 CardNumber;
        U16 Port;
        U32 Val;

        CardNumber = (U16)cardid;
        Port = (U16)addr;

        Status = D2K_DI_ReadPort(CardNumber, Port, &Val);

        *value = (int)Val;

        return (int)Status;
}

int DAQE2214::writeDigitalPort(int cardid, int addr, int value)
{
        I16 Status;
        U16 CardNumber;
        U16 Port;
        U32 Val;

        CardNumber = (U16)cardid;
        Port = (U16)addr;
        Val = (U32)value;

        Status = D2K_DO_WritePort(CardNumber, Port, Val);

        return (int)Status;
}

int DAQE2214::readADC(int cardid, int addr, int *value)
{
        I16 Status;
        U16 CardNumber;
        U16 Channel;
        U16 Val;

        CardNumber = (U16)cardid;
        Channel = (U16)addr;

        Status = D2K_AI_ReadChannel(CardNumber, Channel, &Val);

        *value = (int)((I16)Val);

        return (int)Status;
}

int DAQE2214::readADCArray(int cardid, int addr, int *value, size_t n_request, size_t *n_read)
{
        I16 Status;
        U16 CardNumber;
        U16 Channel;
        U16 Val;
        int i, n;

        CardNumber = (U16)cardid;
        Channel = (U16)addr;

        if (n_request <= BUFFER_SIZE_MAX)
                n = n_request;
        else
                n = BUFFER_SIZE_MAX;

        for (i=0; i<n; i++)
        {
                Status = D2K_AI_ReadChannel(CardNumber, Channel, &Val);

                if (Status != 0)
                        break;

                value[i] = (int)((I16)Val);

                clock_port_value_ |= 0b00000001;
                D2K_DO_WritePort(CardNumber, CLK_OUTPUT_PORT, clock_port_value_);
                clock_port_value_ &= 0b11111110;
                D2K_DO_WritePort(CardNumber, CLK_OUTPUT_PORT, clock_port_value_);

                usleep(1);          // wait analog switch transient
        }
        usleep(4000);       // wait 4ms before a new scan

        *n_read = i+1;

        return (int)Status;
}
```

```
int DAQE2214::writeDAC(int cardid, int addr, int value)
{
          I16 Status;
          U16 CardNumber;
          U16 Channel;
          U16 Val;

          CardNumber = (U16)cardid;
          Channel = (U16)addr;
          Val = (U16)value;

          Status = D2K_AO_WriteChannel(CardNumber, Channel, Val);

          return (int)Status;

}

int DAQE2214::changeADCRange(int cardid, int addr, int range)
{
          I16 Status;
          U16 CardNumber;
          U16 Channel;
          U16 AdRange;

          CardNumber = (U16)cardid;
          Channel = (U16)addr;
          AdRange = ((U16)range)|ADC_MODE;

          Status = D2K_AI_CH_Config(CardNumber, Channel, AdRange);

          return (int)Status;
}

int DAQE2214::changeDACRange(int cardid, int addr, int range)
{
          I16 Status;
          U16 CardNumber;
          U16 Channel;
          U16 OutputPolarity;
          U16 IntOrExtRef;
          F64 refVoltage;

          CardNumber = (U16)cardid;
          Channel = (U16)addr;
          IntOrExtRef = DAC_REF_SOURCE;
          refVoltage = DAC_V_REF;

          OutputPolarity = (U16)range;

          Status = D2K_AO_CH_Config(CardNumber, Channel, OutputPolarity, IntOrExtRef, refVoltage);

          return (int)Status;
}

int DAQE2214::changeDigitalPortDirection(int cardid, int addr, int direction)
{
          I16 Status;
          U16 CardNumber;
          U16 Port;
          U16 Direction;

          CardNumber = (U16)cardid;
          Port = (U16)addr;
          Direction = (U16)direction;

          Status = D2K_DIO_PortConfig(CardNumber, Port, Direction);

          return (int)Status;
}


////////////////////////////////////
// - ADLINK-dipendent functions //
////////////////////////////////////

/////////////////////////////////////////
// + Digital inputs scanning routines //
/////////////////////////////////////////
static void pollerThreadC(void * pPvt)
{
          DAQE2214 *pDAQE2214 = (DAQE2214 *)pPvt;
          pDAQE2214->pollerThread();
}

void DAQE2214::pollerThread()
{
          static const char *functionName = "pollerThread";
          epicsUInt32  biVal, newValue, changedBits, prevInput=0;
          int i, addr, direction, status, val;

          while(1)
          {
                    lock();

                    for (addr = 0; addr < NUM_DIO_PORTS; addr++)
```

```
                {
                        getIntegerParam(addr, digitalDirection_, &direction);

                        if (direction == INPUT_PORT)
                        {
                                status = DAQE2214::readDigitalPort(cardId_, addr, &val);
                                biVal = (epicsUInt32)val;

                                if (status)
                                        asynPrint(pasynUserSelf, ASYN_TRACE_ERROR, "%s:%s: ERROR calling
D2K_DI_ReadPort, status=%d\n", driverName, functionName, status);

                                newValue = biVal;
                                changedBits = newValue ^ prevInput;

                                if (forceCallback_ || (changedBits != 0))
                                {
                                        asynPrint(pasynUserSelf, ASYN_TRACEIO_DRIVER, "%s:%s: Old vlaue =0x%x -> New
value=0x%x read from port=%d \n", driverName, functionName, prevInput, newValue, addr);
                                        prevInput = newValue;
                                        forceCallback_ = 0;
                                        setUIntDigitalParam(addr, digitalInValue_, newValue, 0xFFFFFFFF);
                                }

                        }

                }

                for (i=0; i<MAX_SIGNALS; i++)
                        callParamCallbacks(i);

                unlock();

                epicsThreadSleep(diPollTime_);
        }
}
/////////////////////////////////////////
// - Digital inputs scanning routines //
/////////////////////////////////////////

/////////////////////////////////////////
// + Analog inputs scanning routines //
/////////////////////////////////////////
void scanTask(void * pPvt)
{
        DAQE2214 *pDAQE2214 = (DAQE2214 *)pPvt;
        pDAQE2214->scanTask();
}

void DAQE2214::scanTask(void)
{
        int i, status, val;
        int buffer[BUFFER_SIZE_MAX];
        size_t n_request, n_read;
        static const char *functionName = "scanTask";

        while(1)
        {
                lock();

                for (i = 0; i < NUM_ANALOG_IN; i++)
                {
                        n_request = ch_size[i];

                        if (n_request != 0)
                        {

                                if (n_request == 1)
                                {
                                        status = DAQE2214::readADC(cardId_, i, &val);
                                        setIntegerParam(i, analogInValue_, (epicsInt32)val);
                                }
                                else
                                {
                                        status = DAQE2214::readADCArray(cardId_, i, buffer, n_request, &n_read);
                                        std::copy(buffer, buffer+n_read, pData_[i]);
                                        doCallbacksFloat64Array(pData_[i], n_read, analogInWaveform_, i);
                                }
                        }
                }

                for (i=0; i<MAX_SIGNALS; i++)
                        callParamCallbacks(i);

                unlock();

                epicsThreadSleep(aiPollTime_);
        }
}
/////////////////////////////////////////
// + Analog inputs scanning routines //
/////////////////////////////////////////

///////////////////////////////////////////////
// + Methods overrided from asynPortDriver //
```

```
//////////////////////////////////////////
asynStatus DAQE2214::getBounds(asynUser *pasynUser, epicsInt32 *low, epicsInt32 *high)
{
          int function = pasynUser->reason;

          if ((function == analogInValue_) || (function == analogInWaveform_))
          {
                    *low = ADC_LOW_WORD;
                    *high = ADC_HIGH_WORD;
                    return(asynSuccess);
          }
          else if (function == analogOutValue_)
          {
                    *low = DAC_LOW_WORD;
                    *high = DAC_HIGH_WORD;
                    return(asynSuccess);
          }
          else

          return(asynError);
}

asynStatus DAQE2214::writeInt32(asynUser *pasynUser, epicsInt32 value)
{
          int addr;
          int function = pasynUser->reason;
          int status=0;
          int range, direction, val;
          static const char *functionName = "writeInt32";

          this->getAddress(pasynUser, &addr);
          setIntegerParam(addr, function, value);

          if (function == analogOutValue_)
          {
                    val = (int)value;

                    status = DAQE2214::writeDAC(cardId_, addr, val);

                    if (status == 0)
                              asynPrint(pasynUser, ASYN_TRACEIO_DRIVER, "%s:%s(%d), port %s, wrote %d to card %d, address %d\n",
                                                                  driverName, functionName, function, this->portName, value, cardId_,
addr);
                    else
                              asynPrint(pasynUser, ASYN_TRACE_ERROR, "%s:%s(%d), port %s, ERROR writing %d to card %d, address
%d, status=%d\n", driverName, functionName, function, this->portName, value, cardId_, addr, status);
          }
          else if (function == analogInRange_)
          {
                    getIntegerParam(addr, analogInRange_, &range);
                    status = DAQE2214::changeADCRange(cardId_, addr, range);

                    if (status == 0)
                              asynPrint(pasynUser, ASYN_TRACEIO_DRIVER, "%s:%s(%d), port %s, changed range to %d on address
%d\n", driverName, functionName, function, this->portName, range, addr);
                    else
                              asynPrint(pasynUser, ASYN_TRACE_ERROR, "%s:%s(%d), port %s, ERROR changing range to %d on address
%d, status=%d\n", driverName, functionName, function, this->portName, range, addr, status);
          }
          else if (function == analogOutRange_)
          {
                    getIntegerParam(addr, analogOutRange_, &range);
                    status = DAQE2214::changeDACRange(cardId_, addr, range);

                    if (status == 0)
                              asynPrint(pasynUser, ASYN_TRACEIO_DRIVER, "%s:%s(%d), port %s, changed range to %d on address
%d\n", driverName, functionName, function, this->portName, range, addr);
                    else
                              asynPrint(pasynUser, ASYN_TRACE_ERROR, "%s:%s(%d), port %s, ERROR changing range to %d on address
%d, status=%d\n", driverName, functionName, function, this->portName, range, addr, status);

          }
          else if (function == digitalDirection_)
          {
                    getIntegerParam(addr, digitalDirection_, &direction);
                    status = DAQE2214::changeDigitalPortDirection(cardId_, addr, direction);

                    if (status == 0)
                              asynPrint(pasynUser, ASYN_TRACEIO_DRIVER, "%s:%s(%d), port %s, changed direction to %d on address
%d\n", driverName, functionName, function, this->portName, direction, addr);
                    else
                              asynPrint(pasynUser, ASYN_TRACE_ERROR, "%s:%s(%d), port %s, ERROR changing direction to %d on
address %d, status=%d\n", driverName, functionName, function, this->portName, direction, addr, status);
          }
          else
          {
                    status = asynPortDriver::writeInt32(pasynUser, value);
          }

          callParamCallbacks(addr);

          return (status==0) ? asynSuccess : asynError;
}

asynStatus DAQE2214::readInt32(asynUser *pasynUser, epicsInt32 *value)
```

```
{
          int addr;
          int function = pasynUser->reason;
          int status=0;
          int val;

          static const char *functionName = "readInt32";

          this->getAddress(pasynUser, &addr);

          if (function == analogInValue_)
                    *value = (epicsInt32)pData_[addr][0];
          else
                    status = asynPortDriver::readInt32(pasynUser, value);

          return (status==0) ? asynSuccess : asynError;
}

asynStatus DAQE2214::readFloat64Array(asynUser *pasynUser, epicsFloat64 *value, size_t nElements, size_t *nIn)
{
          int addr;
          int function = pasynUser->reason;
          int status=0;
          int i;
          size_t n;

          static const char *functionName = "readFloat64Array";

          this->getAddress(pasynUser, &addr);


          if (function == analogInWaveform_)
          {

                    if (nElements <= BUFFER_SIZE_MAX)
                              n = nElements;
                    else
                              n = BUFFER_SIZE_MAX;

                    memcpy(value, pData_[addr], n*sizeof(epicsFloat64));
                    *nIn = n;
          }
          else
                    status = asynPortDriver::readFloat64Array(pasynUser, value, nElements, nIn);

          return (status==0) ? asynSuccess : asynError;
}

asynStatus DAQE2214::writeUInt32Digital(asynUser *pasynUser, epicsUInt32 value, epicsUInt32 mask)
{
          int addr;
          int function = pasynUser -> reason;
          int status = 0;
          static const char *functionName = "writeUInt32Digital";

          this->getAddress(pasynUser, &addr);

          if (function == digitalOutvalue_)
          {
                    setUIntDigitalParam(addr, digitalOutvalue_, value, mask);


                    port_value_ &= ~mask;
                    port_value_ |= value;

                    status = DAQE2214::writeDigitalPort(cardId_, addr, port_value_);

                    if (status == 0)
                              asynPrint(pasynUser, ASYN_TRACEIO_DRIVER, "%s:%s(%d), port %s, wrote value=0x%x, mask=0x%x on
address %d\n", driverName, functionName, function, this->portName, value, mask, addr);
                    else
                              asynPrint(pasynUser, ASYN_TRACE_ERROR, "%s:%s(%d), port %s, ERROR writing value=0x%x, mask=0x%x on
address %d\n", driverName, functionName, function, this->portName, value, mask, addr);

          }

          callParamCallbacks(addr);

          return (status==0) ? asynSuccess : asynError;
}

asynStatus DAQE2214::asynDisconnect(void *drvPvt, asynUser *pasynUser)
{
          int function = pasynUser->reason;
          static const char *functionName = "asynDisconnect";

    D2K_Release_Card(cardId_);

    pasynManager->exceptionDisconnect(pasynUser);

          asynPrint(pasynUser, ASYN_TRACEIO_DRIVER, "%s:%s(%d), port %s, Disconnected!\n",
                                                              driverName, functionName, function, this->portName);

          return asynSuccess;
}
```

```
////////////////////////////////////////////
// - Methods overrided from asynPortDriver //
////////////////////////////////////////////

extern "C" int DAQE2214Config(const char *portName, int boardNum, int ai_polltime, int di_polltime,
          int ch0_size, int ch1_size, int ch2_size, int ch3_size, int ch4_size, int ch5_size, int ch6_size, int ch7_size,
          int ch8_size, int ch9_size, int ch10_size, int ch11_size, int ch12_size, int ch13_size, int ch14_size, int ch15_size)
{
          int status;

          DAQE2214 *pDAQE2214 = new DAQE2214(portName, boardNum, ai_polltime, di_polltime,
                                       ch0_size, ch1_size, ch2_size, ch3_size, ch4_size, ch5_size, ch6_size, ch7_size,
                                       ch8_size, ch9_size, ch10_size, ch11_size, ch12_size, ch13_size, ch14_size, ch15_size);
          pDAQE2214 = NULL;

          status = DAQE2214::DAQE2214Init(boardNum);

          return (status==0) ? asynSuccess : asynError;
}

static const iocshArg confArg0 =        { "portName",               iocshArgString};
static const iocshArg confArg1 =        { "Board number",           iocshArgInt};
static const iocshArg confArg2 =        { "AI poll time",           iocshArgInt};
static const iocshArg confArg3 =        { "DI poll time",           iocshArgInt};
static const iocshArg confArg4 =        { "Ch0_size",               iocshArgInt};
static const iocshArg confArg5 =        { "Ch1_size",               iocshArgInt};
static const iocshArg confArg6 =        { "Ch2_size",               iocshArgInt};
static const iocshArg confArg7 =        { "Ch3_size",               iocshArgInt};
static const iocshArg confArg8 =        { "Ch4_size",               iocshArgInt};
static const iocshArg confArg9 =        { "Ch5_size",               iocshArgInt};
static const iocshArg confArg10 =       { "Ch6_size",               iocshArgInt};
static const iocshArg confArg11 =       { "Ch7_size",               iocshArgInt};
static const iocshArg confArg12 =       { "Ch8_size",               iocshArgInt};
static const iocshArg confArg13 =       { "Ch9_size",               iocshArgInt};
static const iocshArg confArg14 =       { "Ch10_size",              iocshArgInt};
static const iocshArg confArg15 =       { "Ch11_size",              iocshArgInt};
static const iocshArg confArg16 =       { "Ch12_size",              iocshArgInt};
static const iocshArg confArg17 =       { "Ch13_size",              iocshArgInt};
static const iocshArg confArg18 =       { "Ch14_size",              iocshArgInt};
static const iocshArg confArg19 =       { "Ch15_size",              iocshArgInt};

static const iocshArg * const confArgs[] = {
          &confArg0,
          &confArg1,
          &confArg2,
          &confArg3,
          &confArg4,
          &confArg5,
          &confArg6,
          &confArg7,
          &confArg8,
          &confArg9,
          &confArg10,
          &confArg11,
          &confArg12,
          &confArg13,
          &confArg14,
          &confArg15,
          &confArg16,
          &confArg17,
          &confArg18,
          &confArg19
};

static const iocshFuncDef configFuncDef = {"DAQE2214Config",20,confArgs};

static void configCallFunc(const iocshArgBuf *args)
{
    DAQE2214Config(args[0].sval, args[1].ival, args[2].ival, args[3].ival,
      args[4].ival, args[5].ival, args[6].ival, args[7].ival, args[8].ival, args[9].ival, args[10].ival, args[11].ival,
      args[12].ival, args[13].ival, args[14].ival, args[15].ival, args[16].ival, args[17].ival, args[18].ival, args[19].ival);
}

void drvDAQE2214Register(void)
{
    iocshRegister(&configFuncDef,configCallFunc);
}

extern "C" {
          epicsExportRegistrar(drvDAQE2214Register);
}
```

PCIe-6216 DAC board asynDriver interface driver

```
/*
 * drvPCIE6216 : Driver for ADLINK PCIe-6216V acquisition board using Mark Rivers' AsynPortDriver
 *
 * Copyright © 2014 Jesus Vasquez
 *
 * This file is part of drvPCIE6216.
 *
 * drvPCIE6216 is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * drvPCIE6216 is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with drvPCIE6216.  If not, see <http://www.gnu.org/licenses/>.
 *
 * Version 1.0
 *
 * Author:       Jesus Vasquez
 * Created on:   Apr 28, 2014
 * Contact:      jesus.vasquez@lnl.infn.it
 *
 */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <algorithm>


#include <epicsTypes.h>
#include <epicsTime.h>
#include <epicsThread.h>
#include <epicsString.h>
#include <epicsTimer.h>
#include <epicsMutex.h>
#include <epicsEvent.h>
#include <iocsh.h>

#include "drvPCIE6216_v1.h"
#include "asynPortDriver.h"
#include <epicsExport.h>

#include "dask.h"

int PCIE6216::cardId_ = -1;
const char * PCIE6216::driverName="PCIE6216_v1";

PCIE6216::PCIE6216 (const char *portName, int boardNum)
        : asynPortDriver(portName,
                                 MAX_SIGNALS,
                                 NUM_PARAMS,
                                 asynInt32Mask | asynDrvUserMask,           // Interface Mask
                                 0,                                         // Interrupt Mask
                                 ASYN_MULTIDEVICE | ASYN_CANBLOCK,          // asynFlags
                                 1,                                         // Autoconnect
                                 0,                                         // Default priority
                                 0),                                        // Default stack size
          boardNum_(boardNum)
{
        // Analog output parameters
        createParam(analogOutValueString, asynParamInt32,          &analogOutValue_);
}

////////////////////////////////////
// + ADLINK-dipendent functions //
////////////////////////////////////
int PCIE6216::PCIE6216Init(int boardNum)
{
        printf("PCIe-6216 board initialization:\n");
        cardId_ = (int)(Register_Card((I16)CARD_TYPE, (I16)boardNum));
```

```
        if (cardId_ < 0)
        {
                printf("Error registering card %d, code= %d\n", boardNum, cardId_);
                exit(-1);
        }
        else
                printf("card number %d registered with ID %d\n", boardNum, cardId_);

        return 0;
}

int PCIE6216::writeDAC(int cardid, int addr, int value)
{
        I16 Status;
        U16 CardNumber;
        U16 Channel;
        U16 Val;

        CardNumber = (U16)cardid;
        Channel = (U16)addr;
        Val = (U16)value;

        Status = AO_WriteChannel(CardNumber, Channel, Val);

        return (int)Status;

}

/////////////////////////////////
// - ADLINK-dipendent functions //
/////////////////////////////////

/////////////////////////////////////////////
// + Methods overrided from asynPortDriver //
/////////////////////////////////////////////
asynStatus PCIE6216::getBounds(asynUser *pasynUser, epicsInt32 *low, epicsInt32 *high)
{
        int function = pasynUser->reason;

        if (function == analogOutValue_)
        {
                *low = DAC_LOW_WORD;
                *high = DAC_HIGH_WORD;
                return(asynSuccess);
        }
        else
                return(asynError);
}

asynStatus PCIE6216::writeInt32(asynUser *pasynUser, epicsInt32 value)
{
        int addr;
        int function = pasynUser->reason;
        int status=0;
        int val;
        static const char *functionName = "writeInt32";

        this->getAddress(pasynUser, &addr);
        setIntegerParam(addr, function, value);

        if (function == analogOutValue_)
        {
                val = (int)value;

                status = PCIE6216::writeDAC(cardId_, addr, val);

                if (status == 0)
                        asynPrint(pasynUser, ASYN_TRACEIO_DRIVER, "%s:%s(%d), port %s, wrote %d to card %d,
address %d\n", driverName, functionName, function, this->portName, value, cardId_, addr);
                else
                        asynPrint(pasynUser, ASYN_TRACE_ERROR, "%s:%s(%d), port %s, ERROR writing %d to card
%d, address %d, status=%d\n", driverName, functionName, function, this->portName, value, cardId_, addr,
status);
        }
        else
        {
                status = asynPortDriver::writeInt32(pasynUser, value);
        }

        callParamCallbacks(addr);
```

```
        return (status==0) ? asynSuccess : asynError;
}

asynStatus PCIE6216::asynDisconnect(void *drvPvt, asynUser *pasynUser)
{
        int function = pasynUser->reason;
        static const char *functionName = "asynDisconnect";

    Release_Card(cardId_);

    pasynManager->exceptionDisconnect(pasynUser);

        asynPrint(pasynUser, ASYN_TRACEIO_DRIVER, "%s:%s(%d), port %s, Disconnected!\n",
                                        driverName, functionName, function, this->portName);

        return asynSuccess;
}

/////////////////////////////////////////
// - Methods overrided from asynPortDriver //
/////////////////////////////////////////

extern "C" int PCIE6216Config(const char *portName, int boardNum)
{
        int status;

        PCIE6216 *pPCIE6216 = new PCIE6216(portName, boardNum);
        pPCIE6216 = NULL;

        status = PCIE6216::PCIE6216Init(boardNum);

        return (status==0) ? asynSuccess : asynError;
}

static const iocshArg confArg0 =  { "portName",            iocshArgString};
static const iocshArg confArg1 =  { "Board number",iocshArgInt};
static const iocshArg * const confArgs[] = {
        &confArg0,
        &confArg1
};

static const iocshFuncDef configFuncDef = {"PCIE6216Config",2,confArgs};

static void configCallFunc(const iocshArgBuf *args)
{
    PCIE6216Config(args[0].sval, args[1].ival);
}

void drvPCIE6216Register(void)
{
    iocshRegister(&configFuncDef,configCallFunc);
}

extern "C" {
        epicsExportRegistrar(drvPCIE6216Register);
}
```

Raspberry Pi IOC beam diagnostic data acquisition interface driver

```
#include <bcm2835.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <cadef.h>
#define epicsAlarmGLOBAL
#include <alarm.h>
#include <epicsEvent.h>
#include <epicsMutex.h>

////////////////////////////////////////////
///////////////// DEFINITIONS /////////////////
////////////////////////////////////////////
#define ADC_CS                      RPI_V2_GPIO_P1_12
#define ADC_RC                      RPI_V2_GPIO_P1_13
#define ADC_BUSY                    RPI_V2_GPIO_P1_15

#define MUX_MAIN_A0                 RPI_V2_GPIO_P1_03
#define MUX_MAIN_A1                 RPI_V2_GPIO_P1_05
#define MUX_MAIN_A2                 RPI_V2_GPIO_P1_07
#define MUX_SEC_A0                  RPI_V2_GPIO_P1_08
#define MUX_SEC_A1                  RPI_V2_GPIO_P1_10
#define MUX_SEC_A2                  RPI_V2_GPIO_P1_11

#define SPI MOSI                    RPI V2 GPIO P1 19
#define SPI MISO                    RPI V2 GPIO P1 21
#define SPI_SCLK                    RPI_V2_GPIO_P1_23

#define MOT1_DIR                    RPI_V2_GPIO_P1_18
#define MOT1_STEP                   RPI_V2_GPIO_P1_16
#define MOT1 LS CW                  RPI V2 GPIO P5 03
#define MOT1_LS_CCW                 RPI_V2_GPIO_P5_04
#define MOT2_DIR                    RPI_V2_GPIO_P1_26
#define MOT2_STEP                   RPI_V2_GPIO_P1_24
#define MOT2_LS_CW                  RPI_V2_GPIO_P5_05
#define MOT2_LS_CCW                 RPI_V2_GPIO_P5_06


#define ANTIBOUNCE LS MAX           1000                    // in microseconds
#define MOTOR_NUMBER                2
#define MOTOR_PULSE_DELAY           800                     // in microseconds
#define ST_BASE                     (0x20003000)            // Hardware timer address
#define TIMER_OFFSET                (4)                     // Hardware timer address

#define  FALSE    0
#define  TRUE     1

/* Strings describing the connection status of a channel */
const char *channel_state_str[4] = {
    "not found",
    "connection lost",
    "connected",
    "closed"
};

/* Define a "process variable" (PV) */
typedef struct {
    chid channel;
    int status;
    struct dbr_ctrl_double info;
    struct dbr_sts_double data;
} epicsDoublePV;

struct args_motor_struct {
        uint8_t motor_index;
};


////////////////////////////////////////////
/////////////// GLOBAL VARIABLES /////////////
////////////////////////////////////////////
```

```
epicsEventId monitorEvent = NULL;
epicsMutexId accessMutex = NULL;
uint8_t data2write = FALSE;
chid data2write_ch = 0;
int32_t data2write_val = 0;

int32_t motor1_pos[MOTOR_NUMBER];
int32_t motor1_pos_target[MOTOR_NUMBER];
int32_t motor1_pos_req[MOTOR_NUMBER];
uint8_t motor1_pos_reach[MOTOR_NUMBER];
int8_t motor1_pos_step[MOTOR_NUMBER];
uint8_t motor_ls_cw[MOTOR_NUMBER];
uint8_t motor_ls_ccw[MOTOR_NUMBER];
uint8_t motor_error[MOTOR_NUMBER];

uint8_t motor_step_pin[MOTOR_NUMBER] = {MOT1_STEP, MOT2_STEP};
uint8_t motor_dir_pin[MOTOR_NUMBER] = {MOT1_DIR, MOT2_DIR};
uint8_t motor_ls_cw_pin[MOTOR_NUMBER] = {MOT1_LS_CW, MOT2_LS_CW};
uint8_t motor_ls_ccw_pin[MOTOR_NUMBER] = {MOT1_LS_CCW, MOT2_LS_CCW};

////////////////////////////////////////////
///////////////// MACROS ///////////////////
////////////////////////////////////////////
/* swap function */
#define SWAP(x)    (((x << 8) & 0xff00) | ((x >> 8) & 0x00ff))

/* get full information about a PV */
#define cainfo(pv, type) SEVCHK(\
    (pv).status = (ca_state((pv).channel) != cs_conn ? ECA_DISCONN : \
    ca_get(dbf_type_to_DBR_CTRL(type), (pv).channel, &(pv).info)), ca_name((pv).channel))

/* get only usually changing information about a PV */
#define caget(pv, type) SEVCHK(\
    (pv).status = (ca_state((pv).channel) != cs_conn ? ECA_DISCONN : \
    ca_get(dbf_type_to_DBR_STS(type), (pv).channel, &(pv).data)), ca_name((pv).channel))

////////////////////////////////////////////
///////////////// FUNCTIONS ////////////////
////////////////////////////////////////////
/* set multiplexer address */
void mux_set_addr(uint8_t addr)
{
        uint8_t addr_low, addr_high;

        addr_high = addr / 8;
        addr_low = addr % 8;

        switch(addr_high)
        {
                case 0:
                        bcm2835_gpio_write(MUX_MAIN_A0, LOW);
                        bcm2835_gpio_write(MUX_MAIN_A1, LOW);
                        bcm2835_gpio_write(MUX_MAIN_A2, LOW);
                        break;
                case 1:
                        bcm2835_gpio_write(MUX_MAIN_A0, HIGH);
                        bcm2835_gpio_write(MUX_MAIN_A1, LOW);
                        bcm2835_gpio_write(MUX_MAIN_A2, LOW);
                        break;
                case 2:
                        bcm2835_gpio_write(MUX_MAIN_A0, LOW);
                        bcm2835_gpio_write(MUX_MAIN_A1, HIGH);
                        bcm2835_gpio_write(MUX_MAIN_A2, LOW);
                        break;
                case 3:
                        bcm2835_gpio_write(MUX_MAIN_A0, HIGH);
                        bcm2835_gpio_write(MUX_MAIN_A1, HIGH);
                        bcm2835_gpio_write(MUX_MAIN_A2, LOW);
                        break;
                case 4:
                        bcm2835_gpio_write(MUX_MAIN_A0, LOW);
                        bcm2835_gpio_write(MUX_MAIN_A1, LOW);
                        bcm2835_gpio_write(MUX_MAIN_A2, HIGH);
                        break;
                case 5:
                        bcm2835_gpio_write(MUX_MAIN_A0, HIGH);
                        bcm2835_gpio_write(MUX_MAIN_A1, LOW);
                        bcm2835_gpio_write(MUX_MAIN_A2, HIGH);
                        break;
                case 6:
                        bcm2835_gpio_write(MUX_MAIN_A0, LOW);
```

```
                        bcm2835_gpio_write(MUX_MAIN_A1, HIGH);
                        bcm2835_gpio_write(MUX_MAIN_A2, HIGH);
                        break;
                case 7:
                        bcm2835_gpio_write(MUX_MAIN_A0, HIGH);
                        bcm2835_gpio_write(MUX_MAIN_A1, HIGH);
                        bcm2835_gpio_write(MUX_MAIN_A2, HIGH);
                        break;
        }

        switch(addr_low)
        {
                case 0:
                        bcm2835_gpio_write(MUX_SEC_A0, LOW);
                        bcm2835_gpio_write(MUX_SEC_A1, LOW);
                        bcm2835_gpio_write(MUX_SEC_A2, LOW);
                        break;
                case 1:
                        bcm2835_gpio_write(MUX_SEC_A0, HIGH);
                        bcm2835_gpio_write(MUX_SEC_A1, LOW);
                        bcm2835_gpio_write(MUX_SEC_A2, LOW);
                        break;
                case 2:
                        bcm2835_gpio_write(MUX_SEC_A0, LOW);
                        bcm2835_gpio_write(MUX_SEC_A1, HIGH);
                        bcm2835_gpio_write(MUX_SEC_A2, LOW);
                        break;
                case 3:
                        bcm2835_gpio_write(MUX_SEC_A0, HIGH);
                        bcm2835_gpio_write(MUX_SEC_A1, HIGH);
                        bcm2835_gpio_write(MUX_SEC_A2, LOW);
                        break;
                case 4:
                        bcm2835_gpio_write(MUX_SEC_A0, LOW);
                        bcm2835_gpio_write(MUX_SEC_A1, LOW);
                        bcm2835_gpio_write(MUX_SEC_A2, HIGH);
                        break;
                case 5:
                        bcm2835_gpio_write(MUX_SEC_A0, HIGH);
                        bcm2835_gpio_write(MUX_SEC_A1, LOW);
                        bcm2835_gpio_write(MUX_SEC_A2, HIGH);
                        break;
                case 6:
                        bcm2835_gpio_write(MUX_SEC_A0, LOW);
                        bcm2835_gpio_write(MUX_SEC_A1, HIGH);
                        bcm2835_gpio_write(MUX_SEC_A2, HIGH);
                        break;
                case 7:
                        bcm2835_gpio_write(MUX_SEC_A0, HIGH);
                        bcm2835_gpio_write(MUX_SEC_A1, HIGH);
                        bcm2835_gpio_write(MUX_SEC_A2, HIGH);
                        break;
        }
}

/* read adc value */
uint64_t read_adc(void) {

        uint64_t buffer = 0;
        uint8_t i, j;


        bcm2835_gpio_write(SPI_SCLK, LOW);
        bcm2835_gpio_write(ADC_CS, LOW);
        bcm2835_gpio_write(ADC_RC, LOW);

        while(!bcm2835_gpio_lev(ADC_BUSY));                // 10 ns delay

        bcm2835_delayMicroseconds(10);

        bcm2835_gpio_write(ADC_RC, HIGH);

        for (j = 0; j < 3; j++)
        {
                for (i = 0; i < 16; i++) {
                        buffer <<= 1;
                        bcm2835_gpio_write(SPI_SCLK, HIGH);
                        bcm2835_gpio_write(SPI_SCLK, LOW);
                        if (bcm2835_gpio_lev(SPI_MISO))
                                buffer++;
                }
```

```
                bcm2835_gpio_write(SPI_SCLK, HIGH);
                bcm2835_gpio_write(SPI_SCLK, LOW);
        }

        bcm2835_gpio_write(ADC_CS, HIGH);

        return buffer;
}
/* This is a user-defined callback function.
   Whenever a channel has a new value, this function is called. */
static void monitor(struct event_handler_args args)
{
        //uint16_t aux_word;

    if (args.status != ECA_NORMAL)
    {
        /* Something went wrong. */
        SEVCHK(args.status, "monitor");
        return;
    }

        const struct dbr_sts_double* data = args.dbr;

    data2write = TRUE;
         data2write_ch = args.chid;
    data2write_val = (int32_t)(data->value);
}

/* print the contents of our PV */
void printDoublePV(const epicsDoublePV* pv)
{
    if (ca_state(pv->channel) != cs_conn)
    {
        printf("%s: <%s>\n",
            ca_name(pv->channel), channel_state_str[ca_state(pv->channel)]);
        return;
    }
    /* Print channel name, native channel type,
       value, units, severity, range and setrange */
    printf("%s (%s as DOUBLE) = %#.*f %s  %s  range:[%#.*f ... %#.*f]  setrange:[%#.*f ... %#.*f]\n",

        /* Get name and native type from channel ID */
        ca_name(pv->channel), dbf_type_to_text(ca_field_type(pv->channel)),

        /* Get static info 'precision' and 'units', and dynamic data 'value' */
        pv->info.precision, pv->data.value, pv->info.units,

        /* Get dynamic data 'severity' */
        epicsAlarmSeverityStrings[pv->data.severity],

        /* Get more static infos */
        pv->info.precision, pv->info.lower_disp_limit,
        pv->info.precision, pv->info.upper_disp_limit,
        pv->info.precision, pv->info.lower_ctrl_limit,
        pv->info.precision, pv->info.upper_ctrl_limit);
}

int help(char *command)
{
        printf("usage: %s <FILTER SIZE> <EPICS VAR FILE>\n Where:\n  <FILTER_SIZE> is the filter size for
analog inputs (>2).\n <EPICS VAR FILE> is the file name that contains the EPICS variable names.\n", command);
        return 1;
}

void* controlMotor1(void *arguments) {

        struct args_motor_struct *args = arguments;
        uint8_t motor_index;
        struct timespec sleeper, dummy;
        long long int t, prev, *timer; // 64 bit timer
    int fd;
    void *st_base; // byte ptr to simplify offset math
        volatile unsigned int dummy2;
        uint32_t motor_mov;
        int32_t motor_pos_cpy;
        uint32_t req_time;

        motor_index = args->motor_index;
        motor1_pos_target[motor_index] = motor1_pos[motor_index];

        // + HARDWARE TIMER ACCESS
```

```
        // get access to system core memory
    if (-1 == (fd = open("/dev/mem", O_RDONLY))) {
        fprintf(stderr, "open() failed.\n");
        return 255;
    }

    // map a specific page into process's address space
    if (MAP_FAILED == (st_base = mmap(NULL, 4096, PROT_READ, MAP_SHARED, fd, ST_BASE))) {
        fprintf(stderr, "mmap() failed.\n");
        return 254;
    }

    // set up pointer, based on mapped page
    timer = (long long int *)((char *)st_base + TIMER_OFFSET);
    // - HARDWARE TIMER ACCESS

        sleeper.tv_sec  = 0;
        sleeper.tv_nsec = (long)(1);

        bcm2835_gpio_write(motor_step_pin[motor_index], LOW);

        while(1)
        {
                if (!motor1_pos_reach[motor_index])
                {
                        if (((motor1_pos_step[motor_index] > 0) & (!motor_ls_cw[motor_index])) |
                        ((motor1_pos_step[motor_index] < 0) & (!motor_ls_ccw[motor_index])))
                        {
                                req_time = abs(motor1_pos_req[motor_index]) * 103;
                                prev = *timer;
                                bcm2835_gpio_write(motor_step_pin[motor_index], HIGH);
                        }
                        else
                        {
                                motor1_pos_target[motor_index] = motor1_pos[motor_index];
                                motor_error[motor_index] = 1;
                        }

                        motor_pos_cpy = motor1_pos[motor_index];

                        do
                        {
                                nanosleep (&sleeper, &dummy);

                                if (((motor1_pos_step[motor_index] > 0) & (motor_ls_cw[motor_index])) |
                                ((motor1_pos_step[motor_index] < 0) & (motor_ls_ccw[motor_index])))
                                {
                                        motor1_pos_target[motor_index] = motor1_pos[motor_index];
                                        motor_error[motor_index] = 1;
                                        break;
                                }

                                t = *timer;
                                motor_mov = (t - prev) / 103;
                                motor1_pos[motor_index] = motor_pos_cpy + motor1_pos_step[motor_index] *
motor_mov;

                                t = *timer;
                        }
                        while((t - prev) < req_time);

                        bcm2835_gpio_write(motor_step_pin[motor_index], LOW);
                        t = *timer;
                        motor_mov = (t - prev) / 103;
                        motor1_pos[motor_index] = motor_pos_cpy + motor1_pos_step[motor_index] * motor_mov;

                        motor1_pos_reach[motor_index] = TRUE;
                }
                else
                        sleep(1);
        }
}

void* readMotorLS(void *arguments) {

        struct args_motor_struct *args = arguments;
        uint8_t motor_index;
        uint16_t auntibounce1_count = 0;
        uint16_t auntibounce2_count = 0;

        motor_index = args->motor_index;
```

```
        while(1)
        {
                // CW Limit switch
                if (!motor_ls_cw[motor_index])
                {
                        if (bcm2835_gpio_lev(motor_ls_cw_pin[motor_index]))
                        {
                                if ((auntibounce1_count++) == ANTIBOUNCE_LS_MAX)
                                {
                                        auntibounce1_count = 0;
                                        motor_ls_cw[motor_index] = 1;
                                }
                        }
                        else
                        {
                                auntibounce1_count = 0;
                        }
                }
                else
                {
                        if (!bcm2835_gpio_lev(motor_ls_cw_pin[motor_index]))
                        {
                                if ((auntibounce1_count++) == ANTIBOUNCE_LS_MAX)
                                {
                                        auntibounce1_count = 0;
                                        motor_ls_cw[motor_index] = 0;
                                }
                        }
                        else
                        {
                                auntibounce1_count = 0;
                        }
                }

                // CCW Limit switch
                if (!motor_ls_ccw[motor_index])
                {
                        if (bcm2835_gpio_lev(motor_ls_ccw_pin[motor_index]))
                        {
                                if ((auntibounce2_count++) == ANTIBOUNCE_LS_MAX)
                                {
                                        auntibounce2_count = 0;
                                        motor_ls_ccw[motor_index] = 1;
                                }
                        }
                        else
                        {
                                auntibounce2_count = 0;
                        }
                }
                else
                {
                        if (!bcm2835_gpio_lev(motor_ls_ccw_pin[motor_index]))
                        {
                                if ((auntibounce2_count++) == ANTIBOUNCE_LS_MAX)
                                {
                                        auntibounce2_count = 0;
                                        motor_ls_ccw[motor_index] = 0;
                                }
                        }
                        else
                        {
                                auntibounce2_count = 0;
                        }
                }
        }
}

////////////////////////////////////////////
////////////////// MAIN ////////////////////
////////////////////////////////////////////
int main(int argc, char **argv)
{
        uint8_t mux_addr;
        uint8_t i;
        char epics_varname_fc[30], epics_varname_vgrid[30], epics_varname_hgrid[30];
        char epics_varname_motor1_pos[MOTOR_NUMBER][30], epics_varname_motor1_req[MOTOR_NUMBER][30],
                epics_varname_motor1_go[MOTOR_NUMBER][30];
        char epics_varname_motor1_ls_cw[MOTOR_NUMBER][30], epics_varname_motor1_ls_ccw[MOTOR_NUMBER][30],
                epics_varname_motor1_error[MOTOR_NUMBER][30];
```

```
char epics_varname_motor_reset[MOTOR_NUMBER][30];
char* aux_ptr;
double voltage, currentFC, currentsV[40], currentsH[40];
uint32_t currentFC_acum, currentsV_acum, currentsH_acum;
uint16_t currentFC_max, currentsV_max, currentsH_max;
uint16_t currentFC_min, currentsV_min, currentsH_min;
uint16_t subval;
uint64_t val;
uint16_t filter_size;
double aux_double_var;
pthread_t *threads;
int err;
struct args_motor_struct *args_motor;
int32_t motor1_pos_cpy[MOTOR_NUMBER];
uint8_t motor_ls_cw_cpy[MOTOR_NUMBER], motor_ls_ccw_cpy[MOTOR_NUMBER], motor_error_cpy[MOTOR_NUMBER],
motor_pos_reach_cpy[MOTOR_NUMBER];
FILE *fr = NULL;

epicsDoublePV epicsvar_fc, epicsvar_vgrid, epicsvar_hgrid;
epicsDoublePV epics_var_motor1_pos[MOTOR_NUMBER], epics_var_motor1_req[MOTOR_NUMBER],
                epics_var_motor1_go[MOTOR_NUMBER];
epicsDoublePV epics_var_motor1_ls_cw[MOTOR_NUMBER], epics_var_motor1_ls_ccw[MOTOR_NUMBER],
                epics_var_motor1_error[MOTOR_NUMBER];
epicsDoublePV epics_var_motor_reset[MOTOR_NUMBER];

double search_timeout = 5.0; /* seconds */
double put_timeout = 1.0; /* seconds */
 int status;
 CA_SYNC_GID gid; /* required for blocking put */

setlinebuf(stdout);

if (argc != 3)
        return help(argv[0]);

filter_size = atoi(argv[1]);

if (filter_size < 3)
        return help(argv[0]);

// Create epics variable name
printf("Opening epics var file %s\n", argv[1]);
fr = fopen(argv[2], "rt");

if (fr == NULL)
{
        printf("Error opening %s\n", argv[1]);
        exit(0);
}
else
{
        printf("Reading file...\n");
        fscanf(fr, "%s", epics_varname_fc);
        fscanf(fr, "%s", epics_varname_hgrid);
        fscanf(fr, "%s", epics_varname_vgrid);
        for (i = 0; i < MOTOR_NUMBER; i++)
        {
                fscanf(fr, "%s", &epics_varname_motor1_req[i][0]);
                fscanf(fr, "%s", &epics_varname_motor1_pos[i][0]);
                fscanf(fr, "%s", &epics_varname_motor1_go[i][0]);
                fscanf(fr, "%s", &epics_varname_motor1_ls_cw[i][0]);
                fscanf(fr, "%s", &epics_varname_motor1_ls_ccw[i][0]);
                fscanf(fr, "%s", &epics_varname_motor1_error[i][0]);
                fscanf(fr, "%s", &epics_varname_motor_reset[i][0]);
        }
        printf("Closing file...\n");
        fclose(fr);
}

if (!bcm2835_init())
{
        printf("Error during BCM2835_INIT() function\n");
        return 1;
}


bcm2835_gpio_fsel(MUX_MAIN_A0, BCM2835_GPIO_FSEL_OUTP);             // Set the pin to be an output
bcm2835_gpio_fsel(MUX_MAIN_A1, BCM2835_GPIO_FSEL_OUTP);             // Set the pin to be an output
bcm2835_gpio_fsel(MUX_MAIN_A2, BCM2835_GPIO_FSEL_OUTP);             // Set the pin to be an output
bcm2835_gpio_fsel(MUX_SEC_A0, BCM2835_GPIO_FSEL_OUTP);              // Set the pin to be an output
bcm2835_gpio_fsel(MUX_SEC_A1, BCM2835_GPIO_FSEL_OUTP);              // Set the pin to be an output
```

```
bcm2835_gpio_fsel(MUX_SEC_A2, BCM2835_GPIO_FSEL_OUTP);              // Set the pin to be an output
bcm2835_gpio_fsel(ADC_CS, BCM2835_GPIO_FSEL_OUTP);                  // Set the pin to be an output
bcm2835_gpio_fsel(ADC_RC, BCM2835_GPIO_FSEL_OUTP);                  // Set the pin to be an output
bcm2835_gpio_fsel(ADC_BUSY, BCM2835_GPIO_FSEL_INPT);                // Set the pin to be an input
bcm2835_gpio_fsel(SPI_MISO, BCM2835_GPIO_FSEL_INPT);                // Set the pin to be an output
bcm2835_gpio_fsel(SPI_MOSI, BCM2835_GPIO_FSEL_OUTP);                // Set the pin to be an output
bcm2835_gpio_fsel(SPI_SCLK, BCM2835_GPIO_FSEL_OUTP);                // Set the pin to be an output
for (i = 0; i < MOTOR_NUMBER; i++)
{
        bcm2835_gpio_fsel(motor_step_pin[i], BCM2835_GPIO_FSEL_OUTP);   // Set the pin to be an output
        bcm2835_gpio_fsel(motor_dir_pin[i], BCM2835_GPIO_FSEL_OUTP);    // Set the pin to be an output
        bcm2835_gpio_fsel(motor_ls_cw_pin[i], BCM2835_GPIO_FSEL_INPT);  // Set the pin to be an input
        bcm2835_gpio_fsel(motor_ls_ccw_pin[i], BCM2835_GPIO_FSEL_INPT); // Set the pin to be an input
}

// PUT CS HIGH
bcm2835_gpio_write(ADC_CS, HIGH);
bcm2835_delay(1);
bcm2835_gpio_write(ADC_RC, HIGH);
bcm2835_delay(1);

bcm2835_gpio_write(MOT1_DIR, LOW);
bcm2835_gpio_write(MOT1_STEP, LOW);

mux_addr = 0;

for (i = 0; i < MOTOR_NUMBER; i++)
{
        motor1_pos[i] = 0;
        motor1_pos_cpy[i] = 0;
        motor1_pos_target[i] = 0;
        motor1_pos_reach[i] = TRUE;
        motor_pos_reach_cpy[i] = TRUE;
        motor1_pos_req[i] = 0;
        motor_ls_cw[i] = 0;
        motor_ls_cw_cpy[i] = 0;
        motor_ls_ccw[i] = 0;
        motor_ls_ccw_cpy[i] = 0;
        motor_error[i] = 0;
        motor_error_cpy[i] = 0;

}

/* Start EPICS multi-threaded */
ca_context_create(ca_enable_preemptive_callback);

aux_ptr = &epics_varname_fc;
puts(aux_ptr);
ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT, &epicsvar_fc.channel);

aux_ptr = &epics_varname_vgrid;
puts(aux_ptr);
ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT, &epicsvar_vgrid.channel);

aux_ptr = &epics_varname_hgrid;
puts(aux_ptr);
ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT, &epicsvar_hgrid.channel);

for (i = 0; i < MOTOR_NUMBER; i++)
{
        aux_ptr = &epics_varname_motor1_pos[i];
        puts(aux_ptr);
        ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT, &epics_var_motor1_pos[i].channel);

        aux_ptr = &epics_varname_motor1_req[i];
        puts(aux_ptr);
        ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT, &epics_var_motor1_req[i].channel);

        aux_ptr = &epics_varname_motor1_go[i];
        puts(aux_ptr);
        ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT, &epics_var_motor1_go[i].channel);

        aux_ptr = &epics_varname_motor1_ls_cw[i];
        puts(aux_ptr);
        ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT,
                                &epics_var_motor1_ls_cw[i].channel);

        aux_ptr = &epics_varname_motor1_ls_ccw[i];
        puts(aux_ptr);
        ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT,
                                &epics_var_motor1_ls_ccw[i].channel);
```

```
        aux_ptr = &epics_varname_motor1_error[i];
        puts(aux_ptr);
        ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT,
                          &epics_var_motor1_error[i].channel);

        aux_ptr = &epics_varname_motor_reset[i];
        puts(aux_ptr);
        ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT,
                          &epics_var_motor_reset[i].channel);
}

SEVCHK(status = ca_pend_io(search_timeout), "searching channels");
if (status != ECA_NORMAL) goto end;

monitorEvent = epicsEventCreate(epicsEventEmpty);

/* Setup a mutex semaphore to make PV access thread-safe */
accessMutex = epicsMutexCreate();
ca_get(DBR_CTRL_DOUBLE, epicsvar_fc.channel, &epicsvar_fc.info);
ca_get(DBR_CTRL_DOUBLE, epicsvar_vgrid.channel, &epicsvar_vgrid.info);
ca_get(DBR_CTRL_DOUBLE, epicsvar_hgrid.channel, &epicsvar_hgrid.info);
for (i = 0; i < MOTOR_NUMBER; i++)
{
        ca_get(DBR_CTRL_DOUBLE, epics_var_motor1_pos[i].channel, &epics_var_motor1_pos[i].info);
        ca_get(DBR_CTRL_DOUBLE, epics_var_motor1_req[i].channel, &epics_var_motor1_req[i].info);
        ca_get(DBR_CTRL_DOUBLE, epics_var_motor1_go[i].channel, &epics_var_motor1_go[i].info);
        ca_get(DBR_CTRL_DOUBLE, epics_var_motor1_ls_cw[i].channel, &epics_var_motor1_ls_cw[i].info);
        ca_get(DBR_CTRL_DOUBLE, epics_var_motor1_ls_ccw[i].channel, &epics_var_motor1_ls_ccw[i].info);
        ca_get(DBR_CTRL_DOUBLE, epics_var_motor1_error[i].channel, &epics_var_motor1_error[i].info);
        ca_get(DBR_CTRL_DOUBLE, epics_var_motor_reset[i].channel, &epics_var_motor_reset[i].info);

        ca_create_subscription(DBR_STS_DOUBLE, 1, epics_var_motor1_go[i].channel, DBE_VALUE|DBE_ALARM,
                          monitor, NULL, NULL);
        ca_create_subscription(DBR_STS_DOUBLE, 1, epics_var_motor1_req[i].channel,
                          DBE_VALUE|DBE_ALARM, monitor, NULL, NULL);
        ca_create_subscription(DBR_STS_DOUBLE, 1, epics_var_motor_reset[i].channel,
                          DBE_VALUE|DBE_ALARM, monitor, NULL, NULL);
}

SEVCHK(status = ca_pend_io(search_timeout), "initializing channels");
if (status != ECA_NORMAL) goto end;

/* Create the "synchronous group id" (gid) used later for put. */
SEVCHK(status = ca_sg_create(&gid), "creating synchronous group");
if (status != ECA_NORMAL) goto end;

/* Initialize motor variables */
aux_double_var = 0;
for (i = 0; i < MOTOR_NUMBER; i++)
{
        ca_sg_put(gid, DBR_DOUBLE, epics_var_motor1_pos[i].channel, &aux_double_var);
        SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epics_var_motor1_pos[i].channel));
        ca_sg_put(gid, DBR_DOUBLE, epics_var_motor1_req[i].channel, &aux_double_var);
        SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epics_var_motor1_req[i].channel));
        ca_sg_put(gid, DBR_DOUBLE, epics_var_motor1_go[i].channel, &aux_double_var);
        SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epics_var_motor1_go[i].channel));
        ca_sg_put(gid, DBR_DOUBLE, epics_var_motor1_ls_cw[i].channel, &aux_double_var);
        SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epics_var_motor1_ls_cw[i].channel));
        ca_sg_put(gid, DBR_DOUBLE, epics_var_motor1_ls_ccw[i].channel, &aux_double_var);
        SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epics_var_motor1_ls_ccw[i].channel));
        ca_sg_put(gid, DBR_DOUBLE, epics_var_motor1_error[i].channel, &aux_double_var);
        SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epics_var_motor1_error[i].channel));
        ca_sg_put(gid, DBR_DOUBLE, epics_var_motor_reset[i].channel, &aux_double_var);
        SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epics_var_motor_reset[i].channel));
}

threads = malloc(2 * MOTOR_NUMBER * sizeof(*threads));
for (i = 0; i < MOTOR_NUMBER; i++)
{
        // Motor control thread
        if((args_motor = malloc(sizeof(*args_motor))) == NULL){
                fprintf(stderr,"MALLOC THREAD_PARAM ERROR");
                return (-1);
        }

        args_motor->motor_index = i;
        err = pthread_create(&(threads), NULL, &controlMotor1, (void *)args_motor);
        if (err != 0)
                printf("\ncan't create thread :[%s]", strerror(err));
        else
```

```
                printf("\n Motor %d thread created successfully\n", i);

        // Motor Limit sitches read thread
        if((args_motor = malloc(sizeof(*args_motor))) == NULL){
                fprintf(stderr,"MALLOC THREAD_PARAM ERROR");
                return (-1);
        }

        args_motor->motor_index = i;
        err = pthread_create(&(threads), NULL, &readMotorLS, (void *)args_motor);
        if (err != 0)
                printf("\ncan't create thread :[%s]", strerror(err));
        else
                printf("\n Motor %d LS thread created successfully\n", i);

}

while(1)
{

        //Read analog inputs

        currentFC_acum = 0;
        currentFC_max = 0;
        currentFC_min = 65535;

        for (mux_addr=0; mux_addr<40; mux_addr++)
        {
                mux_set_addr(mux_addr);
                bcm2835_delayMicroseconds(1);

                currentsV_acum = 0;
                currentsH_acum = 0;
                currentsV_max = 0;
                currentsH_max = 0;
                currentsV_min = 65535;
                currentsH_min = 65535;
                for (i = 0 ; i < filter_size ; i++)
                {
                        // Read all data
                        val = read_adc();

                        // FC data extraction
                        subval = (uint16_t)((val & 0x0000ffff00000000) >> 32);
                        currentFC_acum += subval;
                        if (subval > currentFC_max)
                                currentFC_max = subval;
                        if (subval < currentFC_min)
                                currentFC_min = subval;


                        // BP1 data extraction
                        subval = (uint16_t)((val & 0x00000000ffff0000) >> 16);
                        currentsV_acum += subval;
                        if (subval > currentsV_max)
                                currentsV_max = subval;
                        if (subval < currentsV_min)
                                currentsV_min = subval;

                        //BP2 data extraction
                        subval = (uint16_t)(val & 0x000000000000ffff);
                        currentsH_acum += subval;
                        if (subval > currentsH_max)
                                currentsH_max = subval;
                        if (subval < currentsH_min)
                                currentsH_min = subval;
                }

                currentsV_acum -= currentsV_max;
                currentsV_acum -= currentsV_min;
                voltage = (((double)currentsV_acum) / (65536 * (filter_size - 2))) * 20 - 10;
                currentsV[mux_addr] = 124 * exp(voltage * 1.12321224);
                currentsV[mux_addr] = round(currentsV[mux_addr]*100)/100;

                currentsH_acum -= currentsH_max;
                currentsH_acum -= currentsH_min;
                voltage = (((double)currentsH_acum) / (65536 * (filter_size - 2))) * 20 - 10;
                currentsH[mux_addr] = 124 * exp(voltage * 1.12321224);
                currentsH[mux_addr] = round(currentsH[mux_addr]*100)/100;
        }
```

```
currentFC_acum -= currentFC_max;
currentFC_acum -= currentFC_min;
voltage = (((double)currentFC_acum) / (65536 * (40 * filter_size - 2))) * 20 - 10;
currentFC = 124*exp(voltage*1.12321224);
currentFC = round(currentFC*100)/100;

ca_sg_put(gid, DBR_DOUBLE, epicsvar_fc.channel, &currentFC);
SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epicsvar_fc.channel));
ca_sg_array_put(gid, DBR_DOUBLE, 40, epicsvar_vgrid.channel, &currentsV);
SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epicsvar_vgrid.channel));
ca_sg_array_put(gid, DBR_DOUBLE, 40, epicsvar_hgrid.channel, &currentsH);
SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epicsvar_hgrid.channel));

if (data2write)
{
        for (i = 0; i < MOTOR_NUMBER; i++)
        {
                if (data2write_ch == epics_var_motor1_req[i].channel)
                {
                        motor1_pos_req[i] = data2write_val;
                        data2write = FALSE;
                }

                if (data2write_ch == epics_var_motor1_go[i].channel)
                {
                        if(data2write_val)
                        {
                                if (motor1_pos_req[i] >= 0)
                                {
                                        bcm2835_gpio_write(motor_dir_pin[i], HIGH);
                                        motor1_pos_step[i] = 1;
                                }
                                else
                                {
                                        bcm2835_gpio_write(motor_dir_pin[i], LOW);
                                        motor1_pos_step[i] = -1;
                                }

                                motor1_pos_target[i] += motor1_pos_req[i];
                                motor1_pos_reach[i] = FALSE;
                        }
                        data2write = FALSE;
                }

                if (data2write_ch == epics_var_motor_reset[i].channel)
                {
                        if(data2write_val)
                        {
                                motor1_pos[i] = 0;
                                motor1_pos_target[i] = 0;
                                data2write = FALSE;
                                aux_double_var = 0;
                                ca_sg_put(gid, DBR_DOUBLE,
                                        epics_var_motor_reset[i].channel, &aux_double_var);
                                SEVCHK(ca_sg_block(gid, put_timeout),
                                        ca_name(epics_var_motor_reset[i].channel));
                        }
                }
        }
}

for (i = 0; i < MOTOR_NUMBER; i++)
{
        // Update position
        if (motor1_pos_cpy[i] != motor1_pos[i])
        {
                motor1_pos_cpy[i] = motor1_pos[i];
                aux_double_var = (double)motor1_pos[i];
                ca_sg_put(gid, DBR_DOUBLE, epics_var_motor1_pos[i].channel, &aux_double_var);
                SEVCHK(ca_sg_block(gid, put_timeout),
                        ca_name(epics_var_motor1_pos[i].channel));
        }

        // Update position reach
        if (motor1_pos_reach[i] != motor_pos_reach_cpy[i])
        {
                //motor1_pos_reach[i] = TRUE;
                motor_pos_reach_cpy[i] = motor1_pos_reach[i];
                aux_double_var = 0;
                ca_sg_put(gid, DBR_DOUBLE, epics_var_motor1_go[i].channel, &aux_double_var);
                SEVCHK(ca_sg_block(gid, put_timeout),
```

```
                                ca_name(epics_var_motor1_go[i].channel));
                        }

                        // Update CW limit switch
                        if (motor_ls_cw[i] != motor_ls_cw_cpy[i])
                        {
                                motor_ls_cw_cpy[i] = motor_ls_cw[i];
                                aux_double_var = (double)motor_ls_cw[i];
                                ca_sg_put(gid, DBR_DOUBLE, epics_var_motor1_ls_cw[i].channel,
                                        &aux_double_var);
                                SEVCHK(ca_sg_block(gid, put_timeout),
                                        ca_name(epics_var_motor1_ls_cw[i].channel));
                        }

                        // Update CCW limit switch
                        if (motor_ls_ccw[i] != motor_ls_ccw_cpy[i])
                        {
                                motor_ls_ccw_cpy[i] = motor_ls_ccw[i];
                                aux_double_var = (double)motor_ls_ccw[i];
                                ca_sg_put(gid, DBR_DOUBLE, epics_var_motor1_ls_ccw[i].channel,
                                        &aux_double_var);
                                SEVCHK(ca_sg_block(gid, put_timeout),
                                        ca_name(epics_var_motor1_ls_ccw[i].channel));
                        }

                        // Update Error signal
                        if (motor_error[i] != motor_error_cpy[i])
                        {
                                motor_error_cpy[i] = motor_error[i];
                                aux_double_var = (double)motor_error[i];
                                ca_sg_put(gid, DBR_DOUBLE, epics_var_motor1_error[i].channel,
                                        &aux_double_var);
                                SEVCHK(ca_sg_block(gid, put_timeout),
                                        ca_name(epics_var_motor1_error[i].channel));
                        }
                }
        }

        ca_sg_delete(gid);
end:
    ca_context_destroy();
    epicsMutexDestroy(accessMutex);
    epicsEventDestroy(monitorEvent);

    bcm2835_close();
    return 0;
}
```

Raspberry Pi IOC general purpose IO board interface driver

```
#include <bcm2835.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <cadef.h>
#define epicsAlarmGLOBAL
#include <alarm.h>
#include <epicsEvent.h>
#include <epicsMutex.h>

////////////////////////////////////////////
//////////////// DEFINITIONS ///////////////
////////////////////////////////////////////
#define ADC_CS            RPI_V2_GPIO_P1_03
#define ADC_RC            RPI_V2_GPIO_P1_05
#define ADC_BUSY          RPI_V2_GPIO_P1_07
#define MUX_A0            RPI_V2_GPIO_P1_11
#define MUX_A1            RPI_V2_GPIO_P1_12
#define MUX_A2            RPI_V2_GPIO_P1_13
#define IOE1_CS           RPI_V2_GPIO_P1_15
#define IOE2_CS           RPI_V2_GPIO_P1_16
#define DAC_CS            RPI_V2_GPIO_P1_18
#define SPI_MOSI          RPI_V2_GPIO_P1_19
#define SPI_MISO          RPI_V2_GPIO_P1_21
#define SPI_SCLK          RPI_V2_GPIO_P1_23

#define NUM_VARANAIN_MAX          (8ul)
#define NUM_VARDIGIN_MAX          (1ul)
#define NUM_VARDIGOUT_MAX         (1ul)
#define NUM_VARANAOUT_MAX         (8ul)

#define DATA2WRITE_BUFFER_SIZE    (NUM_VARDIGOUT_MAX + NUM_VARANAOUT_MAX)


#define TRUE     1
#define FALSE    0

/* Strings describing the connection status of a channel */
const char *channel_state_str[4] = {
    "not found",
    "connection lost",
    "connected",
    "closed"
};

/* Define a "process variable" (PV) */
typedef struct {
    chid channel;
    int status;
    struct dbr_ctrl_double info;
    struct dbr_sts_double data;
} epicsDoublePV;

typedef enum {
        set_output = 1,
        set_zero = 2,
        set_gain = 3
} DAC_OPERATION_TYPE;

////////////////////////////////////////////
////////////// GLOBAL VARIABLES ////////////
////////////////////////////////////////////
epicsEventId monitorEvent = NULL;
epicsMutexId accessMutex = NULL;
chid data2write_ch[DATA2WRITE_BUFFER_SIZE];
uint16_t data2write_data[DATA2WRITE_BUFFER_SIZE];
uint8_t data2write_index;


////////////////////////////////////////////
///////////////// MACROS ///////////////////
////////////////////////////////////////////
/* swap function */
#define SWAP(x)    (((x << 8) & 0xff00) | ((x >> 8) & 0x00ff))

/* get full information about a PV */
```

```
#define cainfo(pv, type) SEVCHK(\
    (pv).status = (ca_state((pv).channel) != cs_conn ? ECA_DISCONN : \
    ca_get(dbf_type_to_DBR_CTRL(type), (pv).channel, &(pv).info)), ca_name((pv).channel))

/* get only usually changing information about a PV */
#define caget(pv, type) SEVCHK(\
    (pv).status = (ca_state((pv).channel) != cs_conn ? ECA_DISCONN : \
    ca_get(dbf_type_to_DBR_STS(type), (pv).channel, &(pv).data)), ca_name((pv).channel))

////////////////////////////////////////////
///////////////// FUNCTIONS ////////////////
////////////////////////////////////////////
/* Initialized I/O expansion device MCP23S17 as INPUT */
void IOE1_init(uint8_t addr)
{
        uint8_t i;
        uint8_t IOE_cont;
        uint32_t buffer = 0;

        bcm2835_gpio_write(SPI_SCLK, LOW);

        IOE_cont = 0x40 | ((addr << 1) & 0x0e);
        buffer = ((uint32_t)IOE_cont << 16) | (0x0a08);     // IOCON register -> Hardware address ON

        bcm2835_gpio_write(IOE1_CS, LOW);
        for(i = 0; i < 24; i++) {
                if (buffer & 0x800000)
                        bcm2835_gpio_write(SPI_MOSI, HIGH);
                else
                        bcm2835_gpio_write(SPI_MOSI, LOW);

                bcm2835_gpio_write(SPI_SCLK, LOW);
                bcm2835_gpio_write(SPI_SCLK, HIGH);

                buffer <<= 1;
        }
        bcm2835_gpio_write(IOE1_CS, HIGH);

        buffer = ((uint32_t)IOE_cont << 24) | (0x00ffff);   // IODIRA register (IO direction) -> PORTA & PORTB
                                                            as input

        bcm2835_gpio_write(IOE1_CS, LOW);
        for(i = 0; i < 32; i++) {
                if (buffer & 0x80000000)
                        bcm2835_gpio_write(SPI_MOSI, HIGH);
                else
                        bcm2835_gpio_write(SPI_MOSI, LOW);

                bcm2835_gpio_write(SPI_SCLK, LOW);
                bcm2835_gpio_write(SPI_SCLK, HIGH);

                buffer <<= 1;
        }
        bcm2835_gpio_write(IOE1_CS, HIGH);

        buffer = ((uint32_t)IOE_cont << 24) | (0x0c0000);   // GPPUA register (interenal pull-up R
                                                            configuration) -> PORTA & PORTB int. pullup R OFF

        bcm2835_gpio_write(IOE1_CS, LOW);
        for(i = 0; i < 32; i++) {
                if (buffer & 0x80000000)
                        bcm2835_gpio_write(SPI_MOSI, HIGH);
                else
                        bcm2835_gpio_write(SPI_MOSI, LOW);

                bcm2835_gpio_write(SPI_SCLK, LOW);
                bcm2835_gpio_write(SPI_SCLK, HIGH);

                buffer <<= 1;
        }
        bcm2835_gpio_write(IOE1_CS, HIGH);

}

/* Initialized I/O expansion device MCP23S17 as OUTPUT */
void IOE2_init(uint8_t addr)
{
        uint8_t i;
        uint8_t IOE_cont;
        uint32_t buffer = 0;
```

```
	bcm2835_gpio_write(SPI_SCLK, LOW);

	IOE_cont = 0x40 | ((addr << 1) & 0x0e);
	buffer = ((uint32_t)IOE_cont << 16) | (0x0a08);     // IOCON register -> Hardware address ON

	bcm2835_gpio_write(IOE2_CS, LOW);
	for(i = 0; i < 24; i++) {
		if (buffer & 0x800000)
			bcm2835_gpio_write(SPI_MOSI, HIGH);
		else
			bcm2835_gpio_write(SPI_MOSI, LOW);

		bcm2835_gpio_write(SPI_SCLK, LOW);
		bcm2835_gpio_write(SPI_SCLK, HIGH);

		buffer <<= 1;
	}
	bcm2835_gpio_write(IOE2_CS, HIGH);

	buffer = ((uint32_t)IOE_cont << 24);       // IODIRA register (IO direction) -> PORTA & PORTB as
	                                           output

	bcm2835_gpio_write(IOE2_CS, LOW);
	for(i = 0; i < 32; i++) {
		if (buffer & 0x80000000)
			bcm2835_gpio_write(SPI_MOSI, HIGH);
		else
			bcm2835_gpio_write(SPI_MOSI, LOW);

		bcm2835_gpio_write(SPI_SCLK, LOW);
		bcm2835_gpio_write(SPI_SCLK, HIGH);

		buffer <<= 1;
	}
	bcm2835_gpio_write(IOE2_CS, HIGH);

	buffer = ((uint32_t)IOE_cont << 24) | (0x140000);   // OLATA register (Latch output) -> PORTA & PORTB
	                                                    outputs to 0

	bcm2835_gpio_write(IOE2_CS, LOW);
	for(i = 0; i < 32; i++) {
		if (buffer & 0x80000000)
			bcm2835_gpio_write(SPI_MOSI, HIGH);
		else
			bcm2835_gpio_write(SPI_MOSI, LOW);

		bcm2835_gpio_write(SPI_SCLK, LOW);
		bcm2835_gpio_write(SPI_SCLK, HIGH);

		buffer <<= 1;
	}
	bcm2835_gpio_write(IOE2_CS, HIGH);
}

/* Initialized DAC output to ZERO */
void DAC_init(void)
{
	uint8_t i;
	uint64_t buffer = 0;

	bcm2835_gpio_write(SPI_SCLK, HIGH);

	buffer = 0x00233c00233c;  // RST both DACs

	bcm2835_gpio_write(DAC_CS, LOW);
	for(i = 0; i < 48; i++) {
		if (buffer & 0x800000000000)
			bcm2835_gpio_write(SPI_MOSI, HIGH);
		else
			bcm2835_gpio_write(SPI_MOSI, LOW);

		bcm2835_gpio_write(SPI_SCLK, LOW);
		bcm2835_gpio_write(SPI_SCLK, HIGH);

		buffer <<= 1;
	}
	bcm2835_gpio_write(DAC_CS, HIGH);
}

/* set multiplexer address */
void mux_set_addr(uint8_t addr)
```

```
{
        switch(addr)
        {
                case 0:
                        bcm2835_gpio_write(MUX_A0, LOW);
                        bcm2835_gpio_write(MUX_A1, LOW);
                        bcm2835_gpio_write(MUX_A2, LOW);
                        break;
                case 1:
                        bcm2835_gpio_write(MUX_A0, HIGH);
                        bcm2835_gpio_write(MUX_A1, LOW);
                        bcm2835_gpio_write(MUX_A2, LOW);
                        break;
                case 2:
                        bcm2835_gpio_write(MUX_A0, LOW);
                        bcm2835_gpio_write(MUX_A1, HIGH);
                        bcm2835_gpio_write(MUX_A2, LOW);
                        break;
                case 3:
                        bcm2835_gpio_write(MUX_A0, HIGH);
                        bcm2835_gpio_write(MUX_A1, HIGH);
                        bcm2835_gpio_write(MUX_A2, LOW);
                        break;
                case 4:
                        bcm2835_gpio_write(MUX_A0, LOW);
                        bcm2835_gpio_write(MUX_A1, LOW);
                        bcm2835_gpio_write(MUX_A2, HIGH);
                        break;
                case 5:
                        bcm2835_gpio_write(MUX_A0, HIGH);
                        bcm2835_gpio_write(MUX_A1, LOW);
                        bcm2835_gpio_write(MUX_A2, HIGH);
                        break;
                case 6:
                        bcm2835_gpio_write(MUX_A0, LOW);
                        bcm2835_gpio_write(MUX_A1, HIGH);
                        bcm2835_gpio_write(MUX_A2, HIGH);
                        break;
                case 7:
                        bcm2835_gpio_write(MUX_A0, HIGH);
                        bcm2835_gpio_write(MUX_A1, HIGH);
                        bcm2835_gpio_write(MUX_A2, HIGH);
                        break;
                }

}

/* read adc value */
uint16_t read_adc(void) {

        uint16_t buffer = 0;
        uint8_t i;

        bcm2835_gpio_write(SPI_SCLK, LOW);
        bcm2835_gpio_write(ADC_CS, LOW);
        bcm2835_gpio_write(ADC_RC, LOW);

        while(!bcm2835_gpio_lev(ADC_BUSY));
        bcm2835_delayMicroseconds(10);

        bcm2835_gpio_write(ADC_RC, HIGH);

        for (i = 0; i < 16; i++) {
                buffer <<= 1;
                bcm2835_gpio_write(SPI_SCLK, HIGH);
                bcm2835_gpio_write(SPI_SCLK, LOW);
                if (bcm2835_gpio_lev(SPI_MISO))
                        buffer++;
        }

        bcm2835_gpio_write(ADC_CS, HIGH);

        return buffer;
}

/* set dac value */
void write_dac(DAC_OPERATION_TYPE op_type, uint8_t addr, uint16_t data) {
        uint8_t i, dac_num;
        uint64_t buffer = 0;
        uint64_t data_aux = 0;
        uint64_t data_mask = 0;
```

```
        dac_num = addr / 4;

        buffer &= 0x0000000000000000;
        buffer |= 0x000000037c00037c;      // NOP operations

        addr &= 0x03;
        addr |= ((op_type << 2) & 0x0c);
        data_aux = (((uint64_t)addr) << 16) | ((uint64_t)data);
        data_mask = 0x000000000ffffff;

        for (i = 0; i < dac_num; i++)
        {
                data_aux <<= 24;
                data_mask <<= 24;
        }

        data_mask ^= 0xffffffffffffffff;
        buffer &= data_mask;
        buffer |= data_aux;

        bcm2835_gpio_write(SPI_SCLK, HIGH);

        bcm2835_gpio_write(DAC_CS, LOW);
        for(i = 0; i < 48; i++) {
                if (buffer & 0x800000000000)
                        bcm2835_gpio_write(SPI_MOSI, HIGH);
                else
                        bcm2835_gpio_write(SPI_MOSI, LOW);

                bcm2835_gpio_write(SPI_SCLK, LOW);
                bcm2835_gpio_write(SPI_SCLK, HIGH);

                buffer <<= 1;
        }
        bcm2835_gpio_write(DAC_CS, HIGH);
}

/* Read MCP23S17 */
uint16_t read_ioe(uint8_t addr) {
        uint8_t i;
        uint8_t IOE_cont;
        uint16_t buffer = 0;

        bcm2835_gpio_write(SPI_SCLK, LOW);

        IOE_cont = 0x41 | ((addr << 1) & 0x0e);
        buffer = ((uint32_t)IOE_cont << 8) | (0x12);

        bcm2835_gpio_write(IOE1_CS, LOW);
        for(i = 0; i < 16; i++) {
                if (buffer & 0x8000)
                        bcm2835_gpio_write(SPI_MOSI, HIGH);
                else
                        bcm2835_gpio_write(SPI_MOSI, LOW);

                bcm2835_gpio_write(SPI_SCLK, LOW);
                bcm2835_gpio_write(SPI_SCLK, HIGH);

                buffer <<= 1;
        }

        buffer = 0;

        for (i = 0; i < 16; i++) {
                buffer <<= 1;
                bcm2835_gpio_write(SPI_SCLK, HIGH);
                bcm2835_gpio_write(SPI_SCLK, LOW);
                if (bcm2835_gpio_lev(SPI_MISO))
                        buffer++;
        }

        bcm2835_gpio_write(IOE1_CS, HIGH);

        return buffer;
}

/* Write MCP23S17 */
void write_ioe(uint8_t addr, uint16_t data) {
        uint8_t i;
        uint8_t IOE_cont;
```

```
        uint32_t buffer = 0;

        bcm2835_gpio_write(SPI_SCLK, LOW);

        IOE_cont = 0x40 | ((addr << 1) & 0x0e);
        buffer = ((uint32_t)IOE_cont << 24) | (0x140000) | ((uint32_t)data);

        bcm2835_gpio_write(IOE2_CS, LOW);
        for(i = 0; i < 32; i++) {
                if (buffer & 0x80000000)
                        bcm2835_gpio_write(SPI_MOSI, HIGH);
                else
                        bcm2835_gpio_write(SPI_MOSI, LOW);

                bcm2835_gpio_write(SPI_SCLK, LOW);
                bcm2835_gpio_write(SPI_SCLK, HIGH);

                buffer <<= 1;
        }
        bcm2835_gpio_write(IOE2_CS, HIGH);
}

/* This is a user-defined callback function.
   Whenever a channel has a new value, this function is called. */
static void monitor(struct event_handler_args args)
{
    if (args.status != ECA_NORMAL)
    {
        /* Something went wrong. */
        SEVCHK(args.status, "monitor");
        return;
    }

        const struct dbr_sts_double* data = args.dbr;

    if (data2write_index < DATA2WRITE_BUFFER_SIZE)
    {
                data2write_ch[data2write_index] = args.chid;
                data2write_data[data2write_index++] = (uint16_t)(data->value);
        }

}

uint8_t isData2Write(chid channel, uint16_t *data)
{
        uint8_t data_found = FALSE;
        uint8_t i;

        for (i = 0; i < data2write_index; i++)
        {
                if (data2write_ch[i] == channel)
                {
                        data_found = TRUE;
                        break;
                }
        }

        if (!data_found)
                return FALSE;

        *data = data2write_data[i];

        for (i = i; i < data2write_index; i++)
        {
                data2write_ch[i] = data2write_ch[i + 1];
                data2write_data[i] = data2write_data[i + 1];
        }

        data2write_index--;

        return TRUE;
}

/* print the contents of our PV */
void printDoublePV(const epicsDoublePV* pv)
{
    if (ca_state(pv->channel) != cs_conn)
    {
        printf("%s: <%s>\n",
            ca_name(pv->channel), channel_state_str[ca_state(pv->channel)]);
        return;
```

```
    }
    /* Print channel name, native channel type,
       value, units, severity, range and setrange */
    printf("%s (%s as DOUBLE) = %#.*f %s  %s  range:[%#.*f ... %#.*f]  setrange:[%#.*f ... %#.*f]\n",

        /* Get name and native type from channel ID */
        ca_name(pv->channel), dbf_type_to_text(ca_field_type(pv->channel)),

        /* Get static info 'precision' and 'units', and dynamic data 'value' */
        pv->info.precision, pv->data.value, pv->info.units,

        /* Get dynamic data 'severity' */
        epicsAlarmSeverityStrings[pv->data.severity],

        /* Get more static infos */
        pv->info.precision, pv->info.lower_disp_limit,
        pv->info.precision, pv->info.upper_disp_limit,
        pv->info.precision, pv->info.lower_ctrl_limit,
        pv->info.precision, pv->info.upper_ctrl_limit);
}

int help(char *command)
{
        printf("usage: %s <EPICS VAR FILE> <FILTER SIZE>\n Where:\n  <EPICS VAR FILE> is the file name that
contains the EPICS variable names; and\n  <FILTER_SIZE> is the filter size for analog inputs (>1).\n",
command);
        return 1;
}

////////////////////////////////////////////
////////////////// MAIN ////////////////////
////////////////////////////////////////////
int main(int argc, char **argv)
{
        uint8_t mux_addr;
        uint8_t i;
        uint16_t filter_size, j;
        uint16_t aux_int;
    double ana_val, dig_val;
    double accum;
    char epics_varanain_names[NUM_VARANAIN_MAX][20], epics_vardigin_names[NUM_VARDIGIN_MAX][20];
    char epics_vardigout_names[NUM_VARDIGOUT_MAX][20], epics_varanaout_names[NUM_VARANAOUT_MAX][20] ;
    char* aux_ptr;
    // DAC correction factors
    int8_t gain_code[8] = {92, 81, 87, 91, 127, 127, 127, 127};             //-128 - +127
    int16_t zero_code[8] = {1, 4, 4, 6, 7, 6, 7, 6};                        // -256 - +255
    // ADC correction factors
    float adc_cor_a[8] = {1.005875554, 1.005352639, 1.005255862, 1.005236509, 1.005391355, 1.005333282,
                          1.005275216, 1.005197805};
    float adc_cor_b[8] = {-58.47900212, -41.98484703, -41.35231604, -41.22582446, -42.23789352, -40.85299781, -
                          40.47353728, -39.9676581};
    FILE *fr = NULL;

    epicsDoublePV epicsvaranain[NUM_VARANAIN_MAX];
    epicsDoublePV epicsvardigin[NUM_VARDIGIN_MAX];
    epicsDoublePV epicsvaranaout[NUM_VARANAOUT_MAX];
    epicsDoublePV epicsvardigout[NUM_VARDIGOUT_MAX];
    double search_timeout = 5.0; /* seconds */
    double put_timeout = 1.0; /* seconds */
    int status;
    CA_SYNC_GID gid; /* required for blocking put */

        setlinebuf(stdout);

        if (argc != 3)
                return help(argv[0]);

        filter_size = atoi(argv[2]);

        if (filter_size < 1)
                return help(argv[0]);

        // Read epics variable names
        printf("Opening epics var file %s\n", argv[1]);
        fr = fopen(argv[1], "rt");

        if (fr == NULL)
        {
                printf("Error opening %s\n", argv[1]);
                exit(0);
        }
```

```
    else
    {
            printf("Reading file...\n");

            for (i=0; i<NUM_VARDIGIN_MAX; i++)
                    fscanf(fr, "%s", &epics_vardigin_names[i][0]);

            for (i=0; i<NUM_VARDIGOUT_MAX; i++)
                    fscanf(fr, "%s", &epics_vardigout_names[i][0]);

            for (i=0; i<NUM_VARANAIN_MAX; i++)
                    fscanf(fr, "%s", &epics_varanain_names[i][0]);

            for (i=0; i<NUM_VARANAOUT_MAX; i++)
                    fscanf(fr, "%s", &epics_varanaout_names[i][0]);

            printf("Closing file...\n");
            fclose(fr);
    }

if (!bcm2835_init())
{
            printf("Error during BCM2835_INIT() function\n");
            return 1;
    }

bcm2835_gpio_fsel(ADC_CS, BCM2835_GPIO_FSEL_OUTP);              // Set the pin to be an output
bcm2835_gpio_fsel(ADC_RC, BCM2835_GPIO_FSEL_OUTP);              // Set the pin to be an output
bcm2835_gpio_fsel(ADC_BUSY, BCM2835_GPIO_FSEL_INPT);            // Set the pin to be an input
bcm2835_gpio_fsel(MUX_A0, BCM2835_GPIO_FSEL_OUTP);              // Set the pin to be an output
bcm2835_gpio_fsel(MUX_A1, BCM2835_GPIO_FSEL_OUTP);              // Set the pin to be an output
bcm2835_gpio_fsel(MUX_A2, BCM2835_GPIO_FSEL_OUTP);              // Set the pin to be an output
bcm2835_gpio_fsel(IOE1_CS, BCM2835_GPIO_FSEL_OUTP);             // Set the pin to be an output
bcm2835_gpio_fsel(IOE2_CS, BCM2835_GPIO_FSEL_OUTP);             // Set the pin to be an output
bcm2835_gpio_fsel(DAC_CS, BCM2835_GPIO_FSEL_OUTP);              // Set the pin to be an output
bcm2835_gpio_fsel(SPI_MISO, BCM2835_GPIO_FSEL_INPT);            // Set the pin to be an output
bcm2835_gpio_fsel(SPI_MOSI, BCM2835_GPIO_FSEL_OUTP);            // Set the pin to be an output
bcm2835_gpio_fsel(SPI_SCLK, BCM2835_GPIO_FSEL_OUTP);            // Set the pin to be an output

// PUT CS HIGH
bcm2835_gpio_write(ADC_CS, HIGH);
bcm2835_delay(1);
bcm2835_gpio_write(ADC_RC, HIGH);
bcm2835_delay(1);

    mux_addr = 0;
    for (i=0; i<NUM_VARDIGIN_MAX; i++)
    {
            IOE1_init(i);
            bcm2835_delay(100);
    }
    for (i=0; i < NUM_VARDIGOUT_MAX; i++)
    {
            IOE2_init(i);
            bcm2835_delay(100);
    }

    DAC_init();
    bcm2835_delay(500);

    for (i = 0; i < NUM_VARANAOUT_MAX; i++)
    {
            zero_code[i] &= 0x01ff;                             // Write only 9 bits
            write_dac(set_zero, i, zero_code[i]);
            write_dac(set_gain, i, gain_code[i]);
    }

    /* Start EPICS multi-threaded */
    ca_context_create(ca_enable_preemptive_callback);
    for (i=0; i<NUM_VARDIGIN_MAX; i++)
    {
            aux_ptr = &epics_vardigin_names[i][0];
            puts(aux_ptr);
            ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT, &epicsvardigin[i].channel);
    }
    for (i=0; i<NUM_VARDIGOUT_MAX; i++)
    {
            aux_ptr = &epics_vardigout_names[i][0];
            puts(aux_ptr);
            ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT, &epicsvardigout[i].channel);
    }
```

```
    for (i=0; i<NUM_VARANAIN_MAX; i++)
    {
            aux_ptr = &epics_varanain_names[i][0];
            puts(aux_ptr);
            ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT, &epicsvaranain[i].channel);
    }
    for (i=0; i<NUM_VARANAOUT_MAX; i++)
    {
            aux_ptr = &epics_varanaout_names[i][0];
            puts(aux_ptr);
            ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT, &epicsvaranaout[i].channel);
    }

SEVCHK(status = ca_pend_io(search_timeout), "searching channels");
if (status != ECA_NORMAL) goto end;

    monitorEvent = epicsEventCreate(epicsEventEmpty);

/* Setup a mutex semaphore to make PV access thread-safe */
accessMutex = epicsMutexCreate();
for (i=0; i<NUM_VARANAIN_MAX; i++)
        ca_get(DBR_CTRL_DOUBLE, epicsvaranain[i].channel, &epicsvaranain[i].info);
    for (i=0; i<NUM_VARDIGIN_MAX; i++)
        ca_get(DBR_CTRL_DOUBLE, epicsvardigin[i].channel, &epicsvardigin[i].info);
    for (i=0; i<NUM_VARDIGOUT_MAX; i++)
            ca_get(DBR_CTRL_DOUBLE, epicsvardigout[i].channel, &epicsvardigout[i].info);
    for (i=0; i<NUM_VARANAOUT_MAX; i++)
            ca_get(DBR_CTRL_DOUBLE, epicsvaranaout[i].channel, &epicsvaranaout[i].info);

    for (i=0; i<NUM_VARDIGOUT_MAX; i++)
            ca_create_subscription(DBR_STS_DOUBLE, 1, epicsvardigout[i].channel, DBE_VALUE|DBE_ALARM,
                                   monitor, NULL, NULL);
    for (i=0; i<NUM_VARANAOUT_MAX; i++)
            ca_create_subscription(DBR_STS_DOUBLE, 1, epicsvaranaout[i].channel, DBE_VALUE|DBE_ALARM,
                                   monitor, NULL, NULL);

SEVCHK(status = ca_pend_io(search_timeout), "initializing channels");
if (status != ECA_NORMAL) goto end;

/* Create the "synchronous group id" (gid) used later for put. */
SEVCHK(status = ca_sg_create(&gid), "creating synchronous group");
if (status != ECA_NORMAL) goto end;

/* Init EPICS output variables */
// Digital outputs
    dig_val = 0;
    for (i=0; i<NUM_VARDIGOUT_MAX; i++)
    {
            ca_sg_put(gid, DBR_DOUBLE, epicsvardigout[i].channel, &dig_val);
            SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epicsvaranain[mux_addr].channel));
    }

// Analog outputs
ana_val = 0;
for (i=0; i<NUM_VARANAOUT_MAX; i++)
{
            ca_sg_put(gid, DBR_DOUBLE, epicsvaranaout[i].channel, &ana_val);
            SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epicsvaranain[mux_addr].channel));
    }

    for (i = 0; i < DATA2WRITE_BUFFER_SIZE; i++)
    {
            data2write_ch[i] = 0;
            data2write_data[i] = 0;
    }
    data2write_index = 0;

while(1)
{
            // Read analog inputs
            for (mux_addr=0; mux_addr<NUM_VARANAIN_MAX; mux_addr++)
            {
                    mux_set_addr(mux_addr);
                    bcm2835_delayMicroseconds(10);

                    // Read data
                    accum = 0;
                    for (j=0; j<filter_size; j++)
                            accum += read_adc();

                    //ana_val = accum/filter_size*adc_cor_a[mux_addr]+adc_cor_b[mux_addr];
```

```
                        ana_val =
                                round((((float)accum)/filter_size*adc_cor_a[mux_addr]+adc_cor_b[mux_addr]));
                        //ana_val = accum/filter_size;

                        // Print result
                        //printf("Channel %d = %2.5f Volts\n\r", mux addr, adc val);
                        ca_sg_put(gid, DBR_DOUBLE, epicsvaranain[mux_addr].channel, &ana_val);
                        SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epicsvaranain[mux_addr].channel));
                }

                // Read digital inputs
                for (i=0; i<NUM_VARDIGIN_MAX; i++)
                {
                        aux_int = read_ioe(i);
                        dig_val = SWAP(aux_int);

                        ca_sg_put(gid, DBR_DOUBLE, epicsvardigin[i].channel, &dig_val);
                        SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epicsvardigin[i].channel));
                }


                // Write digital outputs
                for (i = 0; i < NUM_VARDIGOUT_MAX; i++)
                        if (isData2Write(epicsvardigout[i].channel, &aux_int))
                                write_ioe(i, SWAP(aux_int));

                // Write analog outputs
                for (i = 0; i < NUM_VARANAOUT_MAX; i++)
                        if (isData2Write(epicsvaranaout[i].channel, &aux_int))
                                write_dac(set_output, i, aux_int);
        }

        ca_sg_delete(gid);
end:
    ca_context_destroy();
    epicsMutexDestroy(accessMutex);
    epicsEventDestroy(monitorEvent);

    bcm2835_close();
    return 0;
}
```

Raspberry Pi mass separator IOC temperature sensor interface driver

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <cadef.h>
#define epicsAlarmGLOBAL
#include <alarm.h>
#include <epicsEvent.h>
#include <epicsMutex.h>

#define NUM_VAR_MAX                         5
#define EPICS_VAR_NAME_MAX_LENGTH           30
#define SENSOR_NAME_MAX_LENGTH              60
#define FILE_MAX_SIZE                       80
#define TEMP_STR_SIZE                       5

/* Strings describing the connection status of a channel */
const char *channel_state_str[4] = {
    "not found",
    "connection lost",
    "connected",
    "closed"
};

/* Define a "process variable" (PV) */
typedef struct {
    chid channel;
    int status;
    struct dbr_ctrl_double info;
    struct dbr_sts_double data;
} epicsDoublePV;

struct arg_struct {
        int      var_index;
        char*    sensor_name;
};

epicsEventId monitorEvent = NULL;
epicsMutexId accessMutex = NULL;
double temperatures[NUM_VAR_MAX];

/* get full information about a PV */
#define cainfo(pv, type) SEVCHK(\
    (pv).status = (ca state((pv).channel) != cs conn ? ECA DISCONN : \
    ca_get(dbf_type_to_DBR_CTRL(type), (pv).channel, &(pv).info)), ca_name((pv).channel))

/* get only usually changing information about a PV */
#define caget(pv, type) SEVCHK(\
    (pv).status = (ca state((pv).channel) != cs conn ? ECA DISCONN : \
    ca_get(dbf_type_to_DBR_STS(type), (pv).channel, &(pv).data)), ca_name((pv).channel))

int help(char *command)
{
        printf("usage: %s <VARIABLE FILE> <SENSOR FILE>\n Where:\n  <VARIABLE FILE> is the name of the file
with the EPICS variable name; and\n  <SENSOR FILE> is the name of the file with the sensor names.\n", command);
        return 1;
}

void* scanSensor(void *arguments) {

        struct arg_struct *args = arguments;
        char aux_str[FILE_MAX_SIZE];
        int str_size;
        int temp_int;
        FILE *fr = NULL;

        while(1) {
                fr = fopen(args->sensor_name, "rt");
                str_size = fread(aux_str, sizeof(char), FILE_MAX_SIZE, fr);
                fclose(fr);
                if (aux_str[36] == 'Y')           // Consider only reading with correct CRC
                {
                        temp_int = atoi(&aux_str[str_size - TEMP_STR_SIZE - 1]);
                        temperatures[args->var_index] = (float)temp_int / 1000.0;
                }
        }
```

```
}

int main(int argc, char **argv)
{
    FILE *fr = NULL;
    char epics_var_names[NUM_VAR_MAX][EPICS_VAR_NAME_MAX_LENGTH];
    char sensor_names[NUM_VAR_MAX][SENSOR_NAME_MAX_LENGTH];
    unsigned char num_var, i;
    epicsDoublePV epicsvar[NUM_VAR_MAX];
    char *aux_ptr;
    double search_timeout = 5.0; /* seconds */
    double put_timeout = 1.0; /* seconds */
    int status;
    CA_SYNC_GID gid; /* required for blocking put */
    pthread_t *threads;
    int err;
    double temp_copy[NUM_VAR_MAX] = {0, 0};

        setlinebuf(stdout);

        if (argc != 3)
                return help(argv[0]);

        // READ VARIABLE NAME FILE //
        printf("Opening variable name file...\n");
        fr = fopen(argv[1], "rt");

        if (fr == NULL) {
                printf("Error opening file!\n");
                return 1;
        }

        printf("Reading file...\n\n");
        i=0;

        while(fscanf(fr, "%s", &epics_var_names[i][0]) != EOF) {
                if (++i >= NUM_VAR_MAX)
                        break;
        }

        fclose(fr);

        num_var = i;

        // READ SENSOR NAME FILE //
        printf("Opening sensor name file...\n");
        fr = fopen(argv[2], "rt");

        if (fr == NULL) {
                printf("Error opening file!\n");
                return 1;
        }

        printf("Reading file...\n\n");
        i=0;

        while(fscanf(fr, "%s", &sensor_names[i][0]) != EOF) {
                if (++i >= num_var)
                        break;
        }

        fclose(fr);

        printf("Epics var name\t\t\tSensor name\n");
        for (i=0; i<num_var; i++)
                printf("%s\t%s\n", epics_var_names[i], sensor_names[i]);
        printf("\n");

        /* Start EPICS multi-threaded */
        ca_context_create(ca_enable_preemptive_callback);
        for (i=0; i<num_var; i++)
        {
                aux_ptr = &epics_var_names[i][0];
                //puts(aux_ptr);
                ca_create_channel(aux_ptr, NULL, NULL, CA_PRIORITY_DEFAULT, &epicsvar[i].channel);
        }


       SEVCHK(status = ca_pend_io(search_timeout), "searching channels");
       if (status != ECA_NORMAL) goto end;
```

```
        /* Setup a mutex semaphore to make PV access thread-safe */
        accessMutex = epicsMutexCreate();
        for (i=0; i<num_var; i++)
            ca_get(DBR_CTRL_DOUBLE, epicsvar[i].channel, &epicsvar[i].info);

        SEVCHK(status = ca_pend_io(search_timeout), "initializing channels");
        if (status != ECA_NORMAL) goto end;

        /* Create the "synchronous group id" (gid) used later for put. */
        SEVCHK(status = ca_sg_create(&gid), "creating synchronous group");
        if (status != ECA_NORMAL) goto end;

        threads = malloc(num_var * sizeof(*threads));

        for(i=0; i<num_var; i++) {
                struct arg_struct *args;

                if((args = malloc(sizeof(*args))) == NULL){
                        fprintf(stderr,"MALLOC THREAD_PARAM ERROR");
                        return (-1);
                }

                args->sensor_name = sensor_names[i];
                args->var_index = i;

                err = pthread_create(&(threads), NULL, &scanSensor, (void *)args);

                if (err != 0)
                        printf("\ncan't create thread :[%s]", strerror(err));
                else
                        printf("\n Thread created successfully\n");

                sleep(2);
        }

        while(1)
        {
                // Read temperature values
                for (i=0; i<num_var; i++)
                {
                        if (temp_copy[i] != temperatures[i])
                        {
                                temp_copy[i] = temperatures[i];
                                ca_sg_put(gid, DBR_DOUBLE, epicsvar[i].channel, &temperatures[i]);
                                SEVCHK(ca_sg_block(gid, put_timeout), ca_name(epicsvar[i].channel));
                        }
                }
        }

        ca_sg_delete(gid);
end:
    ca_context_destroy();
    epicsMutexDestroy(accessMutex);
    epicsEventDestroy(monitorEvent);

    return 0;

}
```

Stepper motor controller microcontroller program

```
/*
 * File:   main.c
 * Author: Manuel Ramones, Jesus Vasquez
 *
 * Created on March 29, 2013, 3:03 PM
 * Modified on July 6, 2013, 11:51 AM (by Jesus Vasquez)
 */

#include "main.h"
#include <xc.h>
//////////////////////////GLOBAL VARIABLE DECLARATION/////////////////////////////
static char step_index = 0;
static bit ls1_state = FALSE;
static bit ls2_state = FALSE;
static bit ls_error = FALSE;
static bit ls_reach = FALSE;

///////////////////////////////////FUNTIONS/////////////////////////////////////

void mapping_initialization (void) {
//Procedure for mapping the external interruption INT1 to RP11 pin (RC0)
    EECON2 = 0x55;
    EECON2 = 0xAA;
    PPSCON = 0x00;
    RPINR1 = 0x0B;
    EECON2 = 0x55;
    EECON2 = 0xAA;
    PPSCON = 0x01;
}

void initialization (void) {

    MOTA_DIR_TRIS = output;     //PIN RC6 set as output
    MOTA_DIR = FALSE;           //PIN RC6 initialization

    MOTB_DIR_TRIS = output;     //PIN RC7 set as output
    MOTB_DIR = FALSE;           //PIN RC6 initialization

    ADC_EN = 0;                 //disable the ADC module
    PORTB_DIG_PIN1 = 0xFF;      //All the PORTA and PORTB as digital pin
    PORTB_DIG_PIN2 = 0x1F;      //All the PORTA and PORTB as digital pin
    DAC_AB_IN = 0x00;           //Clear all PORTB
    DAC_AB_IN_TRIS = 0x00;      //PORTB set as output

    RPI_PULSE_TRIS = input;     //PIN RC0 set as input
    RPI_DIR_TRSI = input;       //PIN RC1 set as input
    RPI_CS_TRIS = input;        //PIN RC2 set as input

    DIR_LED_TRIS = output;      //PIN RA0 set as input
    PULSE_LED_TRIS = output;    //PIN RA1 set as output
    CS_LED_TRIS = output;       //PIN RA2 set as output

    DIR_LED = LED_OFF;
    PULSE_LED = LED_OFF;
    CS_LED = LED_OFF;

    LS1_TRIS = input;
    LS2_TRIS = input;

    //enable the interruption//
    INT_PE_EN = FALSE;           //Enable the peripheral interrupts
    INT_PRIO_EN = FALSE;        //disable the priority

    #if defined(OP MODE 1)  // Using external interrput 1
    INT_EXT_EN = TRUE;          // Enable the external 1 interrupts
    INT_EXT_EDG = TRUE;         //selection rinsing edge
    INT_EXT_FLAG = FALSE;       //clear the external interrupt flag
    #else                   // Using timer 0 interrput
    T0CON = 0x08;               // Settinf Timer0: Off, 16bits, l-h transition, no preescaler, internal clock.
    INT_TIMER0_FLAG = FALSE;
    INT_TIMER0_EN = TRUE;       // Enable the timer 0 interrupt
    #endif

    INT_GLO_EN = TRUE;          //enable the global interrupts
    ei();
}
```

```
////////////////////////////INTERRUPT ATTENTION FUNCTION//////////////////////

//#pragma interrupt_level 1
void interrupt isr(void) {

    di();   // disable interrupts

    #if defined(OP_MODE_1)
    INT_EXT_FLAG = FALSE;
    #else
    TMR0H = TMR_H_VAL;
    TMR0L = TMR_L_VAL;
    INT_TIMER0_FLAG = FALSE;
    #endif

    if ((!RPI_CS) & (!ls_error)) {
        if (RPI_DIR) {
            if (!ls1_state) {
                step_index++;
                if (step_index == DAC_CODE_SIZE)
                    step_index = 0;
            }
        } else {
            if (!ls2_state) {
                step_index--;
                if (step_index == 255)
                    step_index = (DAC_CODE_SIZE - 1);
            }
        }

        MOTA_DIR = DIR_A[step_index];
        MOTB_DIR = DIR_B[step_index];
        DAC_AB_IN = DAC_CODE[step_index];
    }

    ei();   // Enable interrupts
}

void main(void) {

    unsigned int antibounce_count = 0;
    unsigned int antibounce1_count = 0;
    unsigned int antibounce2_count = 0;
    #ifndef OP_MODE_1
    static bit pulse_prev, pulse_cpy = 0;  // Used to detect edges on pulse signal
    #endif

    initialization();
    mapping_initialization();

    while (1) {

        // Update LED status
        CS_LED = (!RPI_CS);
        DIR_LED = RPI_DIR;
        PULSE_LED = RPI_PULSE;

        // When timer0 version is used, here the PULSE input is checked
        #ifndef OP MODE 1
        pulse_cpy = RPI_PULSE;  // Copy current state of the PULSE signal

        if ((!pulse_prev) & pulse_cpy) {    // Rising edge
            TMR0H = TMR_H_VAL;
            TMR0L = TMR_L_VAL;
            TIMER0_ON = TRUE;
        }
        if (pulse_prev & (!pulse_cpy))  // Falling edge
            TIMER0_ON = FALSE;

        pulse_prev = pulse_cpy; //Save current state for the next iteration

        if (TIMER0_ON & ((RPI_DIR & ls1_state) | ((!RPI_DIR) & ls2_state)))   // Stop the timer when a LS is
                                                                                reached
            TIMER0_ON = FALSE;
        #endif

        // LS1 auntibounce routine
        if (!ls1_state) {
            if (LS1) {
                if ((antibounce1_count++) == ANTIBOUNCE_MAX) {
                    antibounce1_count = 0;
```

```
                ls1_state = TRUE;
            }
        } else {
            antibounce1_count = 0;
        }
    } else {
        if (!LS1) {
            if ((antibounce1_count++) == ANTIBOUNCE_MAX) {
                antibounce1_count = 0;
                ls1_state = FALSE;
            }
        } else {
            antibounce1_count = 0;
        }
    }

    // LS2 auntibounce routine
    if (!ls2_state) {
        if (LS2) {
            if ((antibounce2_count++) == ANTIBOUNCE_MAX) {
                antibounce2_count = 0;
                ls2_state = TRUE;
            }
        } else {
            antibounce2_count = 0;
        }
    } else {
        if (!LS2) {
            if ((antibounce2_count++) == ANTIBOUNCE_MAX) {
                antibounce2_count = 0;
                ls2_state = FALSE;
            }
        } else {
            antibounce2_count = 0;
        }
    }

    // Check for LS error
    if ((ls1_state) & (ls2_state))
        ls_error = TRUE;
    else
        ls_error = FALSE;

  }
}
```

Scilab ENOB calculations

```
// Routine for calculating ENOB from acquired data

clear
close

// Read input data (two tab sepatered columns:
// 1: deltaT between sample, 2: adc word)
M = fscanfMat("data.txt");

// Calculate mean acquisition rate
fs = 1/mean(M(:,1));

// Substract DC component from data
M1 = M(:,2) - mean(M(:,2));

// Calculate FFT module
B = abs(fft(M1));

// Get the number of data points
N = size (B);
N = N(1);

// Plot the power spectrum
C = 10*log10(B(1:N/2).^2);
freq = fs*(0:N/2-1)/N;
plot(freq',C);
title('power spectrum density');
xlabel('frequency(Hz)');
ylabel('|H(f)|(dB)');

// Find the signal frequency
ind_fund = find(B(1:N/2) >= max(B(1:N/2)));
freq_fund = freq(ind_fund);

// Get the power at the signal and zero frequencies
comp_fund = abs(B(ind_fund)).^2;
comp_zero = abs(B(1)).^2;

// Total power on all frequencies
total = sum(abs(B).^2);

// SINAD calculation
SINAD = 20*log10(sqrt(N*(N-3))*sqrt(comp_fund) / sqrt((total - comp_fund - comp_zero)));

// ENOB calculation
ENOB = (SINAD -1.76 +20*log10(10/9))/6.02;

// save result on text files
mdelete("c.txt");
mdelete("f.txt");
mdelete("r.txt");
mdelete("spectrum.png");
write("c.txt",C);
write("f.txt",freq');
print("r.txt",fs,freq_fund,SINAD,ENOB);

// save an image of the plot
wh = get("current figure");
wh.figure size = [1936 1056];
wh.figure_position = [-8 -8];
xgrid(0);
xs2png(0, 'spectrum.png');

// show results to the user
fs
freq_fund
SINAD
ENOB
```

PID Implementation on a EPICS aSub Record

```
#include <registryFunction.h>
#include <epicsExport.h>
#include <aSubRecord.h>
#include <math.h>
#include <string.h>

#define TS       ((double)0.1)    // Sampling time

static long pid(aSubRecord *prec)
{
    double *a, *b, *c, *d, *e, *f, *g, *h, *i, *j, *k, *l;                    // input variables
    double en, sp, pv, kp, ki, kd, f_tau, e_dz, out_max, out_min, cv, mode;   // input parameters
    double k1, k2, k3, a1, a2, delta_out, err2, err_abs;                      // auxiliar variables
    double *ppv_f, *perr, *perr1, *pout;                                      // output variables
    double pv_f, err, err1, out;                                              // output parameters

    // Read Inputs
    a = (double *)prec->a;
    b = (double *)prec->b;
    c = (double *)prec->c;
    d = (double *)prec->d;
    e = (double *)prec->e;
    f = (double *)prec->f;
    g = (double *)prec->g;
    h = (double *)prec->h;
    i = (double *)prec->i;
    j = (double *)prec->j;
    k = (double *)prec->k;
    l = (double *)prec->l;

    // Read Outputs
    pout = (double *)prec->vala;
    perr = (double *)prec->valb;
    perr1 = (double *)prec->valc;
    ppv_f = (double *)prec->vale;

    out = *pout;
    err = *perr;
    pv_f = *ppv_f;
    err1 = *perr1;

    // Put input values into variables
    en = *a;
    sp = *b;
    pv = *c;
    kp = *d;
    ki = *e;
    kd = *f;
    f_tau = *g;
    e_dz = *h;
    out_max = *i;
    out_min = *j;
    cv = *k;
    mode = *l;

    if (en != 0) // Calculate the output only if the system in enable
    {
                // Input signal filter constants
                a1 = TS/(TS+f_tau);
                a2 = f_tau/(TS+f_tau);
                // Apply filter to input signal
                pv_f = a1*pv + a2*pv_f;

                // save the past error values
                err2 = err1;
                err1 = err;
                // Calculate current error
                err_abs = sp - pv;        // error without deadzone
                // Apply deadzone to the error (on another variable)
                if (err_abs < e_dz && err_abs > -e_dz)
                        err = 0;
                else if (err_abs >= e_dz)
                        err = err_abs - e_dz;
                else
                        err = err_abs + e_dz;

                if (mode != 0)   // Process PID calculation only if automatic mode is selected
```

```
                {
                        // PID calculation constants
                        k1 = kp + ki*TS + kd/TS;
                        k2 = -kp - 2*kd/TS;
                        k3 = kd/TS;

                        // Calculate the total output chaneg (P+I+D)
                        delta_out = k1 * err + k2 * err1 + k3 * err2;

                        // Calculate the new output
                        out = out + delta_out;
                }
                else    // If manual mode is selected, just copy the input set cv value to the output
                {
                        out = cv;
                }

                // Check limits
                if (out > out_max)
                        out = out_max;

                if(out < out_min)
                        out = out_min;

    }
    else                // If the system is disabled, keep everithing at zero
    {
         out = 0;
         err = 0;
         err1 = 0;
         pv_f = 0;
    }

    // Write outputs
    memcpy(prec->vala, &out,              sizeof(double)); // New output
    memcpy(prec->valb, &err,              sizeof(double)); // Current error
    memcpy(prec->valc, &err1,             sizeof(double)); // Past error
    memcpy(prec->vald, &err_abs,  sizeof(double)); // Error without dead
    memcpy(prec->vale, &pv_f,             sizeof(double)); // Filtered input

    return 0;
}
epicsRegisterFunction(pid);
```

References

[1] "Eurisol design study," [Online]. Available: http://www.eurisol.org. [Accessed 26 12 2014].

[2] M. Manzolaro, "Study, design and test of the Target - Ion Source system for the INFN SPES facility," Università di Padova, Padua, Italy, 2001.

[3] Y. Jonge and C. Lyneis, Electron Cyclotron Resonance Ion Sources in I.G. Brown, The Physics and Technology of Ion Sources, New York, USA: John Wiley & Sons, 1989.

[4] A. Andrighetto, G. Bisoffi, G. Bassato, L. Calabretta, M. Comunian and J. Vasquez, "SPES: The INFN radioactive beam facility for nuclear physics," in *AIP Conf. Proc. 1491, 58*, 2012.

[5] A. Andrighetto, C. Antonucci, S. Cevolani, C. Petrovich and M. S. Leitner, "Multifoil UCx target for the SPES project – An update," *Eur. Phys. J.,* vol. A, no. 30, pp. 591-601, 2006.

[6] A. Andrighetto, S. Cevolani and C. Petrovich, "Fission fragment production from uranium carbide disc targets," *Eur. Phys. J.,* vol. A, no. 25, pp. 41-47, 2005.

[7] M. Manzolaro, "Master's Degree Thesis," University of Padua, Department of Mechanical, Padua, Italy, 2007.

[8] "Control System Studio project," Spallation Neutron Source (SNS), [Online]. Available: https://ics-web.sns.ornl.gov/css/index.html. [Accessed 26 12 2014].

[9] Argone National Laboratory, "Experimental Physics and Industrial Control System," Argone National Laboratory, [Online]. Available: http://www.aps.anl.gov/epics/. [Accessed 26 12 2014].

[10] M. Giacchini, A. Andrighetto, G. Bassato, L. Costa, R. Izsak, G. Prete and J. Vasquez, "The Control System of SPES Target: Current Status and Perspectives," in *ICALEPCS*, Kobe (Japan), 2009.

[11] M. Giacchini, A. Andrighetto, G. Bassato, N. Conforto, L. Costa, L. Giovannini, C. Scudellaro and J. Vasquez, "EPICS Applications in the Control of SPES Target Laboratory," LNL Annual report, Legnaro (Italy), 2010.

[12] "Raspberry Pi," Raspberry Pi, [Online]. Available: http://www.raspberrypi.org/. [Accessed 26 12 2014].

[13] Raspbian, "Raspbian OS," Raspbian, [Online]. Available: http://www.raspbian.org/. [Accessed 26 12 2014].

[14] Texas Instruments, "LOG112 Precision Logarithmic and Log Ratio Amplifier," Texas Instruments, [Online]. Available: http://www.ti.com/product/log112. [Accessed 26 12 2014].

[15] Texas Instruments, "ADS8509 16-Bit 250kHz CMOS Analog-to-Digital Converter w/Serial Interface 2.5V Internal Reference," Texas Instruments, [Online]. Available: http://www.ti.com/product/ads8509. [Accessed 26 12 2014].

[16] Intersil, "DG408 Single 8-Channel/Differential 4-Channel, CMOS Analog Multiplexers," Intersil, [Online]. Available: http://www.intersil.com/en/products/space-and-harsh-environment/harsh-environment/switches-muxs-crosspoints/DG408.html. [Accessed 26 12 2014].

[17] Texas Instruments, "LMD18245 3A, 55V DMOS Full-Bridge Motor Driver," Texas Instruments, [Online]. Available: http://www.ti.com/product/lmd18245. [Accessed 26 12 2014].

[18] Microchip Technology Inc, "PIC18F27J13 8-bit PIC Microcontroller," Microchip Technology Inc, [Online]. Available: http://www.microchip.com/wwwproducts/Devices.aspx?product=PIC18F27J13. [Accessed 26 12 2014].

[19] Microchip Technology Inc., "Stepper Motor Microstepping with PIC18C452," 2002. [Online]. Available: http://ww1.microchip.com/downloads/en/AppNotes/00822a.pdf. [Accessed 26 12 2014].

[20] Microchip Technology Inc., "Stepper Motor Control Using the PIC16F684," 2004. [Online]. Available: http://ww1.microchip.com/downloads/en/appnotes/00906a.pdf. [Accessed 26 12 2014].

[21] Microchip Technology Inc., "Stepper Motor Control with dsPIC® DSCs," 2009. [Online]. Available: http://www.microchip.com/stellent/groups/techpub_sg/documents/appnotes/en546027.pdf. [Accessed 26 12 2014].

[22] Texas Instruments, "DAC 8734 16-bit Quad High Accuracy +/-16.5V output Digital-to-Analog Converter," Texas Instruments, [Online]. Available: http://www.ti.com/product/dac8734. [Accessed 26 12 2014].

[23] Microchip Technology Inc, "MCP23S17 Interface serial peripherals," Microchip Technology Inc, [Online]. Available: http://www.microchip.com/wwwproducts/Devices.aspx?product=MCP23S17. [Accessed 26 12 2014].

[24] Maxim Integrated, "1µA Supply Current, 1Mbps, 3.0V to 5.5V, RS-232 Transceivers with AutoShutdown Plus," Maxim Integrated, [Online]. Available: http://www.maximintegrated.com/en/datasheet/index.mvp/id/1782. [Accessed 26 12 2014].

[25] M. Rivers, "asynDriver: Asynchronous Driver Support," Advanced Photon Source, Argonne National Laboratory, [Online]. Available: http://www.aps.anl.gov/epics/modules/soft/asyn/. [Accessed 26 12 2014].

[26] D. Zimoch, "StreamDevice 2," PSI, 02 05 2014. [Online]. Available: http://epics.web.psi.ch/software/streamdevice/. [Accessed 26 12 2014].

[27] "IEEE Standard for Terminology and Test Methods for Analog-to-Digital," *IEEE Standard,* p. 1241, 2010.

[28] R. G. Lyons, Understanding Digital Signal Processing, Ann Arbor, USA: Prentice Hall, 2004.

[29] Scilab Enterprises S.A.S, "Scilab," Scilab Enterprises S.A.S, 2015. [Online]. Available: http://www.scilab.org/. [Accessed 26 12 2014].

[30] J. Vasquez, M. Poggi and D. Carlucci, "Beam Diagnostics for the SPES Project," LNL Annual Report, Leganro, Italy., 2013.

[31] G. Bassato, A. Andrighetto, N. Conforto, M. Giacchini, J. A.Montano, M. Poggi and J. Vasquez, "A beam profiler and emittance meter for the spes project at infn-lnl," in *ICALEPCS*, Grenoble (France), 2011.

[32] F. Chip, "USB-COM232-PLUS4 datasheet," 12 04 2010. [Online]. Available: http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_USB-COM232-PLUS4.pdf. [Accessed 26 12 2014].

[33] M. Integrated, "DS1822 Econo 1-Wire Digital Thermometer," Maxim Integrated, 2014. [Online]. Available: http://www.maximintegrated.com/en/products/analog/sensors-and-sensor-interface/DS1822.html. [Accessed 26 12 2014].

[34] T. M. Mooney, "The sscan record for sscan module," Advanced Photon Source, Argonne National Laboratory, [Online]. Available: http://www.aps.anl.gov/bcda/synApps/sscan/sscanRecord.html. [Accessed 26 12 2014].

[35] J. Vasquez, A. Andrighetto, G. Bassato, L. Costa and M. Giacchini, "Safety Control System and its Interface to Epics for the Off-Line Front End of the SPES Project," in *ICALEPCS*, Grenoble (France), 2011.

[36] D-Link, "DUB-E100," D-Link, [Online]. Available: http://us.dlink.com/products/connect/high-speed-usb-2-0-fast-ethernet-adapter/. [Accessed 26 12 2014].

[37] M. Rivers, "Driver Support for Modbus Protocol under EPICS," The University of Chicago, Argonne National Laboratory, Advanced Photon Source, 19 08 2014. [Online]. Available: http://cars9.uchicago.edu/software/epics/modbusDoc.html. [Accessed 26 12 2014].

[38] ADLINK, "ADLINK COM Express," ADLINK, [Online]. Available: http://www.adlinktech.com/Computer-on-Module/Com-Express/index.php. [Accessed 26 12 2014].

[39] ADLINK, "ADLINK Express-IB COM Express," ADLINK, [Online]. Available: http://www.adlinktech.com/PD/web/PD_detail.php?cKind=&pid=1122&seq=&id=&sid=&source=&utm_source=&category=Computer-on-Modules_COM-Express-Type-6-Basic. [Accessed 26 12 2014].

[40] "Adlink Computer-on-Module," Adlink, [Online]. Available: http://www.adlinktech.com/Computer-on-Module. [Accessed 26 12 2014].

[41] B. Wolf, Handbook of Ion Sources, CRC Press, 1995.

[42] Cosylab, "Cosylab's microIOC," Cosylab, [Online]. Available: http://www.microioc.com/. [Accessed 26 12 2014].

[43] M. S. Muhammad, H. Mutahira and T.-S. Choi, "Using a low pass filter to recover three-dimentional shape from focus in the presence of noise," *International journal of innovative computing, information and control,* vol. 8, no. 4, pp. 2777-2788, 2012.

[44] M. Gopal, Digital Control and State Variable Methods. COnventional and Intelligent Control Systems, Third ed., New Delhi: Tata McGraw-Hill, 2009.

[45] J. Ziegler and N. Nichols, "Optimum settings for automatic controllers," *Transactions of the ASME,* vol. 64, pp. 759-768, 1942.

[46] M. Cavenago, T. Kulevoy and S. Petrenko, "Operation of the LNL ECR Ion Source," in *EPAC*, Vienna (Austria), 2000.

[47] "Functional Safety and IEC 61508," IEC, [Online]. Available: http://www.iec.ch/functionalsafety/. [Accessed 26 12 2014].

[48] "The 61508 Association," [Online]. Available: http://www.61508.org/. [Accessed 26 12 2014].

[49] "Profinet," PROFIBUS & PROFINET International, [Online]. Available: http://www.profibus.com/technology/profinet/. [Accessed 26 12 2014].

[50] "Profisafe," PROFIBUS & PROFINET International, [Online]. Available: http://www.profibus.com/technology/profisafe/. [Accessed 26 12 214].

[51] J. Vasquez, S. Canella, G. Prete, G. Bassato, G. Bisoffi, D. Zafiropoulos, G. Scarabottolo and C. Buoso, "New Access Control System for the Tandem-Alpi-Piave Accelerators at LNL," *LNL annual report,* p. 235, 2012.

[52] J. Vasquez, S. Canella, G. Prete, G. Bassato, G. Bisoffi, D. Zafiropoulos, G. Scarabottolo and C. Buoso, "Updates for the New Access Control System for the Tandem-Alpi-Piave Accelerators at LNL," *LNL annual report,* p. 2014, 2013.